Higher-Order Representation Predicates in Separation Logic

Arthur Charguéraud

Inria, Université Paris-Saclay, France
LRI, CNRS & Univ. Paris-Sud, Université Paris-Saclay, France
arthur.chargueraud@inria.fr

Abstract

In Separation Logic, representation predicates are used to describe mutable data structures, by establishing a relationship between the entry point of the structure, the piece of heap over which this structure spans, and the logical model associated with the structure. When a data structure is polymorphic, such as in the case of a container, its representation predicate needs to be parameterized not just by the type of the items stored in the structure, but also by the representation predicates associated with these items. Such higher-order representation predicates can be used in particular to control whether containers should own their items. In this paper, we present, through a collection of practical examples, solutions to the challenges associated with reasoning about accesses into data structures that own their elements.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Formal methods

Keywords Separation Logic, Representation Predicates.

1. Introduction

Separation Logic [11, 14] is a Hoare-style program logic that is particularly well-suited for modular verification of mutable data structures. In this logic, specifications are expressed in terms of *representation predicates*, which establish a relationship between the entry point of a structure, the piece of heap over which this structure spans, and the logical model associated with the structure (e.g., a sequence, a set, a map, etc.).

Separation Logic was originally presented as a first-order logic, and was subsequently extended to a higher-order setting [2]. This extension was motivated by two essential features associated with the possibility of quantifying over heap predicates, i.e., predicates over pieces of heap. The first one is *abstraction*: existential quantification over heap predicates gives the ability to hide the pieces of heap that are private to a data structure. The second is *modularity*: universal quantification over representation predicates gives the ability to specify a polymorphic data structure in terms of the representation predicate describing the elements stored in this structure. In this paper, we focus on modularity.

To illustrate the modularity offered by higher-order representation predicates, and to explain the challenges associated with them, we consider the classic example of queues. O'Hearn et al. [12] present a representation predicate for queues that is parameterized by the heap predicate describing the elements stored in the queue. However, this parameterization could only take place at the meta-level, due to their use of a first-order logic. Biering et al. [2] describe the same example, this time in a higher-order Separation Logic, in which the heap predicate describing the elements can be quantified within the logic. Svendsen et al. [15] present a slightly different representation predicate for queues, describing a queue storing program values as a list of mathematical values, establishing a connection between program values and their corresponding mathematical values through the representation predicate provided for describing the elements. We follow their approach.

The specification for push (enqueue) and pop (dequeue) operations on queues can be expressed as shown below. There, $p \leadsto \mathsf{Queueof}\, R\, L$ denotes the representation predicate for a queue object allocated at location p; R denotes the representation predicate for the elements; and L denotes the list of mathematical values describing the queue—L is the model of the queue.

$$\forall xpRXL.\quad \{p \leadsto \mathsf{Queueof}\, R\, L \star x \leadsto R\, X\}\ (\mathtt{push}\, x\, p)\ \{\lambda\_.\ p \leadsto \mathsf{Queueof}\, R\, (L \mathbin{+\!\!+} X :: \mathsf{nil})\}$$

$$\forall pRXL.\quad \{p \leadsto \mathsf{Queueof}\, R\, (X :: L)\}\ (\mathtt{pop}\, p)\ \{\lambda x.\ p \leadsto \mathsf{Queueof}\, R\, L \star x \leadsto R\, X\}$$

Above, '$\lambda\_.$' denotes the unit value returned by `push`, and "$\lambda x.$" denotes the value x returned by `pop`. The heap predicate $x \leadsto R\, X$ establishes a relationship between a program value x and the corresponding mathematical value X, and describes a nonempty piece of heap in the case where x corresponds to an object allocated in memory. In Svendsen et al.'s presentation, all values in the specification language (e.g. x and X) belong to a global type called Val. By contrast, in this paper, we will assign proper types to the values involved (see §4.4).

In the specifications given above, observe the ownership transfer at play. The `push` operation migrates the ownership of an element, described by $x \leadsto R\, X$, from the client to the queue data structure. Reciprocally, the `pop` operation gives back to the client the ownership of the element returned. This form of ownership transfer is very useful for the client of the data structure, who would otherwise be required to separately maintain, in addition to the queue representation predicate, an assertion describing the ownership of the elements stored in the queue.

Yet, the specification of a container data structure using a representation predicate that includes the ownership of the elements raises an important challenge. This challenge comes from the fact that a read in the data structure cannot simply return the ownership of the item read, otherwise multiple reads would lead to duplicating the owernership of this item. To illustrate the problem, consider the function `peek`, which returns the element at the head of the queue, but without taking it out of the queue. On the one hand, if the specification of peek does not transfer ownership of the head element, then the client code will be prevented from inspecting the head element, effectively making peek useless. On the other hand, if the specification of peek transfers ownership of the head element, then the ownership of this element would be duplicated, thus the corresponding specification, shown below, would be incorrect.

$$\textit{(Incorrect)} \quad \forall pRXL. \quad \{p \rightsquigarrow \mathsf{Queueof}\, R\, (X :: L)\}\ (\mathtt{peek}\, p) \\ \{\lambda x.\ p \rightsquigarrow \mathsf{Queueof}\, R\, (X :: L) \star x \rightsquigarrow R\, X\}$$

The magic wand, a traditional tool from Separation Logic [11, 14], brings a solution to the specification of peek. The magic wand, written $H' \mathrel{-\!\!\star} H$, describes a piece of heap, call it H'', such that, if H'' is extended with H', then the result would be exactly H. In other words, the heap predicate $H' \star (H' \mathrel{-\!\!\star} H)$ can be converted into H. Using the magic wand, we can specify peek as follows.

$$\{p \rightsquigarrow \mathsf{Queueof}\, R\, (X :: L)\}\ (\mathtt{peek}\, p) \\ \{\lambda x.\ x \rightsquigarrow R\, X \star (x \rightsquigarrow R\, X \ \mathrel{-\!\!\star}\ p \rightsquigarrow \mathsf{Queueof}\, R\, (X :: L))\}$$

This specification of peek is sufficient for most practical applications. Its only limitation is that, after a call to peek, the queue cannot be acted upon before the ownership of the head element has been given back to the queue. Indeed, the magic wand form does not match the pre-condition of other operations such as push or pop.

The magic wand can be used, like in the example of peek, to reason about containers in which a single piece of ownership is carved out. However, when multiple pieces need to be carved out independently, as it is the case for example with arrays that own their elements, the magic wand falls short of solving the challenge of reasoning about accesses. For the specific case of mutable lists, which have been used extensively in the Separation Logic literature, the introduction of list segments comes to the rescue. Yet, the notion of list segments does not generalize so easily to more complex tree-based and array-based data structures.

In this paper, we describe techniques for detaching the ownership of items from trees and arrays that own their elements, and also for carving out the ownership of entire subtrees from given trees. The main contributions of this paper are the following.

- Mutable lists: using them, we give a gentle introduction to higher-order representation predicate and present the rewriting rules that are useful in practice for reasoning about them. We have not found such material in prior publications.
- Arrays: we present a representation predicates for describing subsets of cells from an array that owns its elements, as well as rules for converting between a description where the array owns its elements and one where it does not. We illustrate such conversions using matrices that either have disjoint rows or possibly-aliased rows. We have not seen this type of conversions described in prior work, although they are needed in practice.
- Records: we present a systematic pattern for introducing higher-order representation predicates for records. These predicates allow us to control, on a field per field basis, whether the items stored in the fields should be owned. Such use of higher-order representation predicates for describing record fields that do not have a polymorphic type appears to be novel.
- Trees with holes and cut subtrees: we present a novel technique that allows us to detach the ownership of items stored in a tree, and also to detach the ownership of entire subtrees. This technique supports reasoning about trees that contain several missing pieces, and reasoning about the reattaching of items and subtrees that may have been modified or permuted arbitrarily.
- Polymorphic recursion: we show that a tree data structures involving polymorphic recursion can be very elegantly described using a higher-order representation predicate that is recursively applied to itself. As far as we know, this is the first formalization of a mutable data structure involving polymorphic recursion.

A majority (although not all) of the definitions presented in this paper have been formalized and put to practice using CFML [4, 5]. This tool takes as input an un-annotated ML program and generates its *characteristic formula*, reflecting the program semantics as a Coq logical formula. The details of characteristic formulae is not relevant to this paper; only matters the fact that CFML provides a Coq library that includes a shallow embedding of Separation Logic in Coq, and that many of the examples presented in this paper have been formalized in that setting.

The paper begins with a short justification of the choice of the logic and the programming language used to present the examples, and with a presentation of the minimal background on Separation Logic required for reading this paper.

2. Logic and Source Language

The definitions of higher-order representation predicates requires a higher-order logic specification language. In this paper, we present the definitions of logical data types and of representation predicate assuming the logic of the Coq proof assistant [6]. Note, however, that these definitions could be similarly formalized in other general-purpose, higher-order logic theorem provers.

The notion of representation predicates is not specific to any particular programming language. The choice of a concrete language for presenting data structures and example programs in this paper was guided by two constraints. First, because polymorphism plays a key role in this paper, we want a language that makes polymorphism explicit in the definitions of data types, and that supports polymorphic recursion. Second, for describing mutable lists and trees without introducing a cumbersome sum type that would clutter definitions, we want a language that includes null pointers.

As we do not know of a programming language that features both well-typed polymorphic recursion and null pointers, we consider an artificial language, the same one as used by CFML [4, 5]. This language inherits the syntax and semantics of OCaml [9] but at the same time offers null pointers. For the purpose of ML type inference, we assume null to be polymorphic, i.e. to have type "$\forall A.\ A$". However, once we have a well-typed ML program, we type-check it again using a different type system, called weak-ML ([3], Section 4.3), in particular to ensure that null can only be used in places where a pointer value is expected.

Weak-ML essentially consists of an erasure of ML in which all pointer values, including null, admit the constant type loc, and in which all functions admit the constant type func. The only role of weak-ML is to reflect program values other than pointers and functions at their corresponding type in the logic. For example, an OCaml value that is an option on a boolean in the program is described as an option on a boolean in Coq (the assertion language). Weak-ML is not meant to enforce any type safety property, it simply keeps track of the types of program values in order to reflect them in the Separation Logic. The fact that functions and dereferencing operations indeed return values of the type claimed by the programmer is verified through the Separation Logic reasoning.

In summary, although the choice of this artificial programming language might be debatable, it is motivated by the existing formalizations conducted using CFML, and by the fact that the use of null pointers simplify many definitions. In any case, the ideas presented in this paper can be adapted to other programming language.

3. Background on Separation Logic

In Separation Logic, a heap predicate has type $\mathsf{Heap} \rightarrow \mathsf{Prop}$ and characterizes a portion of the heap. The fundamental combinators for heap predicates are agnostic to the representation of heaps, that is, to the definition of the type Heap. These combinators can either be axiomatized, or they can be *defined* in a standard higher-order logic. The latter amounts to constructing a shallow embedding of Separation Logic in a standard higher-order logic. This was done for example in Coq (e.g. [5, 10]) and in Isabelle/HOL (e.g. [16]). In this paper, we assume a shallow embedding construction.

The definitions of these combinators are shown below and explained next. There, h denotes a heap, H denotes a heap predicate, and P denotes a proposition.

$$\begin{array}{lcl} [P] & \equiv & \lambda h.\ h = \varnothing \ \wedge\ P \\ H_1 \star H_2 & \equiv & \lambda h.\ \exists h_1 h_2.\ h_1 \perp h_2 \ \wedge\ h = h_1 \uplus h_2 \\ & & \qquad\qquad \wedge\ H_1\, h_1 \ \wedge\ H_2\, h_2 \\ \exists x.\, H & \equiv & \lambda h.\ \exists x.\ H\, h \end{array}$$

The pure heap predicate $[P]$ characterizes an empty heap and at the same time asserts that P holds. The empty predicate, written $[\,]$, is a shorthand for $[\mathsf{True}]$. Existential quantification is lifted to the level of heap predicates, taking the form $\exists x.\ H$. The separating conjunction (a.k.a. star) of two heap predicates takes the form $H_1 \star H_2$ and asserts that the heap can be partitioned in two disjoint parts: one that satisfies H_1 and one that satisfies H_2. Its definition involves two auxiliary notions: $h_1 \perp h_2$ asserts that the heaps h_1 and h_2 have disjoint domains; $h_1 \uplus h_2$ denotes the union of two disjoint heaps. Moreover, we write $\circledast_{i\in I}\, H_i$ for iterating the separating conjunction on a collection of heap predicates $(H_i)_{i\in I}$.

Separation Logic also relies on a primitive heap predicate for describing an atomic piece of heap. The exact form of this predicate depends on the view of memory exposed by the programming language. In a low-level view of memory such as that of the C programming language, the type Heap is interpreted as a map from locations to machine words, and the heap predicate $l \hookrightarrow n$ asserts that the memory cell at location l stores the word n. Building on top of this predicate, one can derive higher-level predicates describing, say, a record object laid out in memory.

In a high-level view of memory such as that of the OCaml programming language, the type Heap may be interpreted as an heterogeneous map from locations to values. We write $p \mapsto_A v$ to describe a singleton heap binding p to a value v of type A. We typically omit the type annotation. For example, we write $p \mapsto \{\!|\mathrm{hd}{=}v;\ \mathrm{tl}{=}p'|\!\}$ to assert that a list-cell record with two fields storing the value v and p' is allocated in memory at location p. Here, p and p' have type loc, and v is a value of some type a, where a is the logical type associated with the program value v.

Throughout the paper, we illustrate the use of heap predicates in the specification of functions. The specification of a function typically takes the form $\forall x.\ \{H\}\ (f\ x)\ \{\lambda y.\ H'\}$, where x denotes the argument, y denotes the output value, H denotes the precondition describing the input heap, and H' denotes the postcondition describing the output heap. The Separation Logic triples are given their standard, total correctness interpretation.[1]

4. Mutable Lists

We begin with the study of mutable lists, a simple data structure that nevertheless suffices to motivate and illustrate a number of challenges associated with higher-order representation predicates.

[1] A triple $\{H\}\ t\ \{Q\}$ is interpreted as shown below, where $\Downarrow$ denotes the big-step evaluation judgment, h_1 describes the heap on which t operates, h_2 describes the framed heap untouched by the evaluation of t, v describes the output value, h_1' describes the modified heap, and h_3 describes the discarded pieces of heap. Explanations can be found in a previous paper [5].

$$\forall h_1 h_2.\ \left\{ \begin{array}{l} H\, h_1 \\ h_1 \perp h_2 \end{array} \right. \Rightarrow\ \exists v h_1' h_3.\ \left\{ \begin{array}{l} t_{/h_1 \uplus h_2} \Downarrow v_{/h_1' \uplus h_2 \uplus h_3} \\ h_1' \perp h_2 \perp h_3 \\ Q\, v\, h_1' \end{array} \right.$$

4.1 Implementation of Null-Terminated Mutable Lists

A mutable list consists of a sequence of cells. Each cell has a pointer to the next cell. The null pointer denotes the end of the list. More precisely, each cell is represented as a two-field record, with one field for the head, storing an element from the list, and one field for the back, storing a pointer to the rest of the list.

The code below shows the record data type, which is parameterized, followed by an example mutable list storing two integers.

```
type 'a cell = { mutable hd : 'a;
                 mutable tl : 'a cell }
let demo = { hd = 8; tl = { hd = 5; tl = null } }
```

4.2 Traditional, First-Order List Representation Predicate

In Separation Logic [14], a mutable list is described by a heap predicate of the form $\mathsf{Mlist}\, L\, p$, asserting that, at pointer p, there exists a list storing the values described by the list L. Here, p is a logical value of type loc that describes a memory location, and L is a logical value of type $\mathsf{list}\, A$, for some type A. For example, the `demo` list defined in §4.1 is described by the heap predicate $\texttt{demo} \leadsto \mathsf{Mlist}\,(8 :: 5 :: \mathsf{nil})$.

To improve the readability of heap descriptions, we introduce an arrow notation that we will use for all heap predicates that have a main entry point. We let $p \leadsto Q$ be a notation for "$Q\, p$", that is, for the application of the predicate Q to the value p. In particular, we write $p \leadsto \mathsf{Mlist}\, L$ in place of $\mathsf{Mlist}\, L\, p$, to better suggest the intuition that "at p, we find a list described by L".

The heap predicate $p \leadsto \mathsf{Mlist}\, L$ is defined by structural recursion over the list L, as shown below. When the list is null, the heap predicate simply asserts that p must be null, in the empty heap. When the list L is of the form $x :: L'$, the heap predicate asserts the existence of a list cell described by the predicate $p \mapsto \{\!|\mathrm{hd}{=}x;\ \mathrm{tl}{=}p'|\!\}$, where p' is the pointer to the tail of the list. Thus, $p' \leadsto \mathsf{Mlist}\, L'$ describes the tail of the list.

$$\begin{array}{lcl} p \leadsto \mathsf{Mlist}\, L & \equiv & \mathsf{match}\ L\ \mathsf{with} \\ & & \mid \mathsf{nil}\ \Rightarrow\ [p = \mathsf{null}] \\ & & \mid x :: L'\ \Rightarrow\ \exists p'.\quad p \mapsto \{\!|\mathrm{hd}{=}x;\ \mathrm{tl}{=}p'|\!\} \\ & & \qquad\qquad\qquad\qquad \star\ p' \leadsto \mathsf{Mlist}\, L' \end{array}$$

Note that the star operator that appears in the above definition ensures that all list cells involved are disjoint from each other. Note also the use of the existential quantification on p', which introduces abstraction by hiding the locations of the intermediate list cells.

To illustrate a function specification involving Mlist, consider the function `copy`, which copies a mutable list. This function admits the specification shown below, where p is the pointer to the input list and p' is the pointer on the output list.

$$\forall pL.\ \{p \leadsto \mathsf{Mlist}\, L\}\ (\texttt{copy}\, p)\ \{\lambda p'.\ p \leadsto \mathsf{Mlist}\, L \star p' \leadsto \mathsf{Mlist}\, L\}$$

The specification above asserts in particular that the cells of the output list are disjoint from the cells of the input list, and that the input list was left unaltered by the copy process.

4.3 Towards Higher-Order List Representation Predicate

Consider the case of a program that manipulates a mutable list of disjoint mutable lists. To reason about such a data structure, it can be convenient to establish a direct relationship between the entry pointer to the outer list, and the logical representation of the data structure, which is a list of lists, that is, a logical value of type $\mathsf{list}\,(\mathsf{list}\, A)$. We are thus seeking for a heap predicate, written MlistofMlist, to relate a pointer p with a list of lists. For example, a concrete instance of such a predicate might be:

$$p \leadsto \mathsf{MlistofMlist}\ \left((5 :: 7 :: \mathsf{nil}) :: (8 :: 3 :: \mathsf{nil}) :: (\mathsf{nil}) :: (4 :: \mathsf{nil})\ ::\ \mathsf{nil}\right).$$

There are two main approaches to defining the heap predicate $\mathsf{MlistofMlist}$. The first approach consists of describing first the outer list, and then describing the iterated separating conjunction of the inner lists. In the formal definition shown below, K has type $\mathsf{list\,loc}$ and describes the list of the pointers stored in the outer list, whereas L has type $\mathsf{list}\,(\mathsf{list}\,A)$, for some A, and describes the list of the inner lists. In the definition shown below, $|L|$ denotes the length of L, and the iterated star asserts that the mutable list found at address $K[i]$ describes the list $L[i]$.

$$p \rightsquigarrow \mathsf{MlistofMlist}\,L \;\equiv\; \exists K.\; [\,|K| = |L|\,] \star p \rightsquigarrow \mathsf{Mlist}\,K \\ \star \circledast_{i \,\in\, [0,\,|L|)}\; (K[i]) \rightsquigarrow \mathsf{Mlist}\,(L[i])$$

A second approach to defining $\mathsf{MlistofMlist}$ is to generalize the definition of the predicate Mlist, to account for the fact that each element from the outer list is itself an entry point into a mutable list. In the definition shown below, x has type loc and describes a pointer to one of the inner lists, whereas X has type $\mathsf{list}\,A$ and describes the logical representation of the corresponding list.

$$\begin{array}{lll} p \rightsquigarrow \mathsf{MlistofMlist}\,L & \equiv & \mathsf{match}\,L\,\mathsf{with} \\ & & \mid \mathsf{nil} \;\Rightarrow\; [p = \mathsf{null}] \\ & & \mid X :: L' \;\Rightarrow\; \exists x p'.\;\; p \mapsto \{\!|\mathrm{hd}{=}x;\ \mathrm{tl}{=}p'|\!\} \\ & & \qquad\qquad \star\, p' \rightsquigarrow \mathsf{MlistofMlist}\,L' \\ & & \qquad\qquad \star\, x \rightsquigarrow \mathsf{Mlist}\,X \end{array}$$

The two definitions presented above for $\mathsf{MlistofMlist}$ can be easily proved equivalent (by induction on L). Compared with the first definition, the second one generalizes better to the case of tree-shaped data structures (see §7). Hence, we thereafter follow the second approach. We will nevertheless exploit the equivalence between the two definitions on a few occasions.

In the example above, we considered the case of a mutable list of mutable lists. In general, we could have mutable lists of any kind of mutable objects, for example mutable lists of references, mutable lists of arrays, mutable lists of mutable queues, etc. In what follows, we generalize the definition of $\mathsf{MlistofMlist}$ to a higher-order predicate of the form $\mathsf{Mlistof}\,R$, that is, a higher-order representation predicate for lists that takes as argument the representation predicate that applies to the elements.

4.4 Higher-Order List Representation Predicate

We define a higher-order representation predicate for lists, written $p \rightsquigarrow \mathsf{Mlistof}\,R\,L$. This definition, which generalizes that of $\mathsf{MlistofMlist}$ simply by replacing Mlist with an abstract representation predicate R, is as follows.

$$\begin{array}{lll} p \rightsquigarrow \mathsf{Mlistof}\,R\,L & \equiv & \mathsf{match}\,L\,\mathsf{with} \\ & & \mid \mathsf{nil} \;\Rightarrow\; [p = \mathsf{null}] \\ & & \mid X :: L' \;\Rightarrow\; \exists x p'.\;\; p \mapsto \{\!|\mathrm{hd}{=}x;\ \mathrm{tl}{=}p'|\!\} \\ & & \qquad\qquad \star\, p' \rightsquigarrow \mathsf{Mlistof}\,R\,L' \\ & & \qquad\qquad \star\, x \rightsquigarrow R\,X \end{array}$$

This definition is highly polymorphic, so it is worth making explicit all the types involved. Recall that $p \rightsquigarrow \mathsf{Mlistof}\,R\,L$ is a notation for $\mathsf{Mlistof}\,R\,L\,p$, and that, similarly, $x \rightsquigarrow R\,X$ is a notation for $R\,X\,x$. Recall also that both these heap predicates have type Hprop, that is $\mathsf{Heap} \rightarrow \mathsf{Prop}$. Let a denote the type of the concrete values stored in the mutable list beginning at address p. The variable x that appears in the definition has type a. Let A denote the logical type associated with these concrete values. The variable X that appears in the definition has type A and L has type $\mathsf{list}\,A$. The abstract representation predicate R thus has type $A \rightarrow a \rightarrow \mathsf{Hprop}$, and $\mathsf{Mlistof}$ admits the following type:

$$\mathsf{Mlistof} \;:\; \forall A a.\; (A \rightarrow a \rightarrow \mathsf{Hprop}) \rightarrow \mathsf{list}\,A \rightarrow \mathsf{loc} \rightarrow \mathsf{Hprop}.$$

Importantly, $\mathsf{Mlistof}$ is a strict generalization of Mlist, in the sense that we can define a heap predicate Id such that $\mathsf{Mlistof\,Id}$ yields exactly Mlist. The predicate Id is defined as follows:

$$\mathsf{Id} \quad\equiv\quad \lambda X.\,\lambda x.\; [x = X].$$

Note that Id is polymorphic and has type $\forall A.\; A \rightarrow A \rightarrow \mathsf{Hprop}$. For this definition of Id, we can prove the intended equality[2]:

$$p \rightsquigarrow \mathsf{Mlistof\,Id}\,L \;=\; p \rightsquigarrow \mathsf{Mlist}\,L.$$

Now that we have defined $\mathsf{Mlistof}$, it remains to see how we use it in practice to state specifications of functions manipulating mutable lists of possibly-mutable objects.

4.5 Example: a Higher-Order Copy Function

In §4.2, we considered a specification for a `copy` function that duplicates the structure of a list. A naive generalization of the specification, shown below, would be incorrect, because it would duplicate the ownership of the list items.

(Incorrect) $\forall pL.\ \{p \rightsquigarrow \mathsf{Mlistof}\,R\,L\}\ (\mathtt{copy}\,p)$
$\{\lambda p'.\ p \rightsquigarrow \mathsf{Mlistof}\,R\,L \star p' \rightsquigarrow \mathsf{Mlistof}\,R\,L\}$

The problem is that the `copy` function considered only performs a shallow copy of the list, not a deep copy. If we really want to specify the behavior of the `copy` function on a list described by $p \rightsquigarrow \mathsf{Mlistof}\,R\,L$, because of the sharing induced we first need to change the heap description to $p \rightsquigarrow \mathsf{Mlist}\,K \star \circledast_i\ K[i] \rightsquigarrow \mathsf{Mlist}\,L[i]$ for some K, exploiting the equivalence stated in §4.3.

If, however, we consider a function that performs a deep copy of the list, then we can give an interesting specification in terms of $p \rightsquigarrow \mathsf{Mlistof}\,R\,L$. Let `deepcopy` be a function that, while duplicating the list structure, makes calls to a given function f for constructing copies of the elements. We can specify the higher-order function `deepcopy` as shown below. The first part of the specification describes the hypothesis made about the behavior of f as a copy function for the elements, whereas the remaining part describes the pre- and post-condition associated with applications of `deepcopy` to f and to a pointer p on a mutable list.

$$\begin{array}{l} \forall f p R L. \quad \big(\forall x X.\ \{x \rightsquigarrow R\,X\}\ (f\,x) \\ \qquad\qquad\qquad\qquad \{\lambda x'.\ x \rightsquigarrow R\,X \star x' \rightsquigarrow R\,X\}\big) \\ \quad \Rightarrow\ \{p \rightsquigarrow \mathsf{Mlistof}\,R\,L\}\ (\mathtt{deepcopy}\,f\,p) \\ \qquad\qquad \{\lambda p'.\ p \rightsquigarrow \mathsf{Mlistof}\,R\,L \star p' \rightsquigarrow \mathsf{Mlistof}\,R\,L\} \end{array}$$

4.6 Example: Mutable List of Functions with Local State

In this example, we consider a mutable list of counter functions. These counter functions have an internal state implemented using a reference, but this reference is not directly accessible. We then traverse the list using a higher-order iterator, making a call to each of the counters to update their internal states.

A *counter* function is a function that, each time it is called, returns the next natural number. A fresh counter function can be constructed as shown below, by allocating a reference cell with initial value 0 and then returning a function that increments the counter and returns its current value.

```
let mkcounter () =
  let r = ref 0 in (fun () -> incr r; get r)
```

We define a predicate, written $f \rightsquigarrow \mathsf{Count}\,n$, to assert that the function closure f denotes a counter whose current state is n. This heap predicate describes in particular the ownership of the local

[2] The equality $p \rightsquigarrow \mathsf{Mlistof\,Id}\,L \;=\; p \rightsquigarrow \mathsf{Mlist}\,L$ relates two heap predicates. It should be interpreted according to predicate extensionality, as the logical equivalence between the two predicates on all heaps. Formally:

$$(Q_1 = Q_2) \;\equiv\; (\forall h.\ Q_1\,h \Leftrightarrow Q_2\,h).$$

In fact, by further exploiting predicate extensionality, we could even prove the equality: $\mathsf{Mlistof\,Id} = \mathsf{Mlist}$.

state of the function, that is, the reference cell associated with it. The predicate $f \rightsquigarrow \mathsf{Count}\, n$ is defined as shown below. Note that f is a value of type func (recall §2). Observe in particular how the concrete address of the reference cell implementing the state of the counter is existentially quantified.

$$\begin{aligned}&f \rightsquigarrow \mathsf{Count}\, n \;\equiv\\ &\quad \exists r.\,(r \hookrightarrow n)\\ &\quad\quad \star\, [\forall i.\ \{r \hookrightarrow i\}\ (f\ ())\ \{\lambda x.\,[x = i+1] \star (r \hookrightarrow i+1)\}]\end{aligned}$$

Instances of the predicate Count can be introduced for example by exploiting the specification of `mkcounter`, which is as follows.

$$\{[\,]\}\ (\texttt{mkcounter ()})\ \{\lambda f.\ f \rightsquigarrow \mathsf{Count}\, 0\}$$

The definition of $f \rightsquigarrow \mathsf{Count}\, n$ is hidden from the end-user; only the specification shown below is exposed. This specification asserts that a call to a counter f increments its internal state, denoted as i, and returns the value of the new internal state.

$$\{f \rightsquigarrow \mathsf{Count}\, i\}\ (f\ ())\ \{\lambda x.\, f \rightsquigarrow \mathsf{Count}\,(i+1) \star [x = i+1]\}$$

Remark: our specification of calls to the counter function is similar to that presented by Svendsen et al. [15], although they do not attempt to hide the implementation of the inner state.

We are now ready to verify the following program, which allocates two counters, increments the first one, then creates a mutable list with the two counters, and iterates over the list a function that increments every counter. Below, `miter` denotes a higher-order iterator on mutable lists (its definition is not shown).

```
let a = mkcounter() in
let b = mkcounter() in
let n = a() in
let p = { hd = a; tl = { hd = b; tl = null }} in
miter (fun f -> ignore(f())) p
```

The state before the creation of the list p is described by: $a \rightsquigarrow \mathsf{Count}\, 1 \star b \rightsquigarrow \mathsf{Count}\, 0$. The state after the creation of the list p is described by: $p \rightsquigarrow \mathsf{Mlistof\, Count}\,(1 :: 0 :: \mathsf{nil})$. The state after the iteration of the function `(fun f -> f())` on each of the elements the list p is described by: $p \rightsquigarrow \mathsf{Mlistof\, Count}\,(2 :: 1 :: \mathsf{nil})$.

It remains to present the proof steps involved for reasoning about the call to `miter`. We show below the specification of `miter`. It is similar in spirit to the specification of a `fold` function presented by Svendsen et al. [15]. Below, L denotes the input (logical) list, L' denotes the output (logical) list, and I denotes the loop invariant which relates the current state with the already-processed prefixes K and K' of the lists L and L'. The variable X denotes an element from L, and f denotes a counter, i.e. a concrete element stored in the mutable list. We use the notation $(K\&X)$ as a shorthand for $K +\!\!+ (X :: \mathsf{nil})$.

$$\begin{aligned}&\forall g p I R R' L.\\ &\quad\big(\forall f X K K' T.\ \ L = K +\!\!+ X :: T \Rightarrow\\ &\quad\quad \{I\, K\, K' \star f \rightsquigarrow R\, X\}\ (g\, f)\\ &\quad\quad\quad \{\lambda\_.\ \exists X'.\ I\,(K\&X)\,(K'\&X') \star f \rightsquigarrow R'\, X'\}\big)\\ &\Rightarrow\ \{p \rightsquigarrow \mathsf{Mlistof}\, R\, L \star I\,\mathsf{nil}\,\mathsf{nil}\}\ (\texttt{miter}\ g\, p)\\ &\quad\quad\quad\quad \{\lambda\_.\ \exists L'.\ p \rightsquigarrow \mathsf{Mlistof}\, R'\, L' \star I\, L\, L'\}\end{aligned}$$

In our example, f is a pointer to a function closure, and "$g\, f$" reduces to an invocation of the counter, that is, to "$\mathsf{ignore}f()$". To reason about the call to `miter`, we exploit the above specification by taking $L \equiv 1 :: 0 :: \mathsf{nil}$ and $L' \equiv 2 :: 1 :: \mathsf{nil}$. We instantiate R as Count and define I such that: $I\, K\, K' \equiv [K' = \mathsf{map}\,(+1)\, K]$. The proof obligation arising from the premise, and associated with reasoning about the call to the function `(fun f -> f())`, is:

$$\begin{aligned}&\forall x K K' X.\ \{[K' = \mathsf{map}\,(+1)\, K] \star f \rightsquigarrow \mathsf{Count}\, X\}\ (\mathsf{ignore}(f()))\\ &\quad \{\lambda\_.\ \exists X'.\ [(K\&X) = \mathsf{map}\,(+1)\,(K'\&X')] \star f \rightsquigarrow \mathsf{Count}\, X'\}\end{aligned}$$

One can check that, if we instantiate X' as $X+1$, and ignore the return value, then specification above is derivable from the exposed specification for counter functions stated earlier, that is:

$$\{f \rightsquigarrow \mathsf{Count}\, i\}\ (f\ ())\ \{\lambda x.\,[x = i+1] \star f \rightsquigarrow \mathsf{Count}\,(i+1)\}$$

In conclusion, the predicate Mlistof can be used to describe mutable lists, not only those storing plain pointer structures, but also those storing functions with local state. Moreover, as this example shows, a higher-order representation predicate such as Mlistof can be smoothly integrated into the specification of higher-order iterators.

5. List Segments

In this section, we adapt the traditional list segment heap predicate to a higher-order representation predicate, following the same approach as for null-terminated lists.

5.1 Representation of List Segments

The heap predicate $p \rightsquigarrow \mathsf{Mlistsegof}\, R\, q\, L$ asserts that, starting at pointer p, the heap contains the beginning of a mutable list segment, such that the elements contained between p (inclusive) and q (exclusive) are described by the list L . The formal definition, shown below, differs from the definition of $p \rightsquigarrow \mathsf{Mlistof}\, R\, L$ in that $[p = \mathsf{null}]$ gets replaced with [p = q].

$$\begin{aligned}p \rightsquigarrow \mathsf{Mlistsegof}\, R\, q\, L \equiv\ & \mathsf{match}\ L\ \mathsf{with}\\ & \mid \mathsf{nil}\ \Rightarrow\ [p = q]\\ & \mid X :: L'\ \Rightarrow \exists x p'.\ p \mapsto \{\!|\mathrm{hd}{=}x;\ \mathrm{tl}{=}p'|\!\}\\ &\quad\quad \star\ p' \rightsquigarrow \mathsf{Mlistsegof}\, R\, q\, L'\\ &\quad\quad \star\ x \rightsquigarrow R\, X\end{aligned}$$

Note that null-terminated lists are a special case of list segments:

$$p \rightsquigarrow \mathsf{Mlistof}\, R\, L\ \ =\ \ p \rightsquigarrow \mathsf{Mlistsegof}\, R\,\mathsf{null}\, L.$$

We have experienced that the following equalities are very useful for reasoning about programs manipulating list segments.

$$\begin{aligned}\mathsf{null} \rightsquigarrow \mathsf{Mlistsegof}\, R\, q\, L\ &= [L = \mathsf{nil}\ \wedge\ q = \mathsf{null}]\\ p \rightsquigarrow \mathsf{Mlistsegof}\, R\, q\,(L_1 +\!\!+ L_2)\ &= \exists p'.\ \ p \rightsquigarrow \mathsf{Mlistsegof}\, R\, p'\, L_1\\ &\quad\quad \star\, p' \rightsquigarrow \mathsf{Mlistsegof}\, R\, q\, L_2\\ p \rightsquigarrow \mathsf{Mlistsegof}\, R\, q\,(L\&X)\ &= \exists x p'.\ \ p \rightsquigarrow \mathsf{Mlistsegof}\, R\, p'\, L\\ &\quad\quad \star\, p' \mapsto \{\!|\mathrm{hd}{=}x;\ \mathrm{tl}{=}q|\!\}\\ &\quad\quad \star\, x \rightsquigarrow R\, X\\ p \rightsquigarrow \mathsf{Mlistsegof}\, R\, L \star [p \neq \mathsf{null}] &= \exists x p' X L'.\ [L = X :: L']\\ &\quad\quad \star\, p \mapsto \{\!|\mathrm{hd}{=}x;\ \mathrm{tl}{=}p'|\!\}\\ &\quad\quad \star\, p' \rightsquigarrow \mathsf{Mlistsegof}\, R\, L'\\ &\quad\quad \star\ x \rightsquigarrow R\, X\end{aligned}$$

5.2 Example: Getting and Reading the n-th Cell of a List

The function `nth`, whose code appears next, returns the i-th cell of a mutable list.

```
let rec nth (i:int) (p:'a cell) =
  if i = 0 then p else nth (i-1) (p.tl)
```

If the pre-condition for this function describes a full list of the form $p \rightsquigarrow \mathsf{Mlistof}\, R\, L$, what could the post-condition be? The challenge is that the location returned by `nth` corresponds to one of the locations existentially quantified by Mlistof. For describing the post-condition, it appears that we have little choice but to introduce a list segment that splits the list at the location returned by `nth`. This can be done in two ways: either the list can be logically split between the pre- and the post-condition, or this split can be anticipated and be performed beforehand in the reasoning.

In the first case, the specification of `nth` takes the following form. Below, q denotes the return value, and the list gets split into a prefix segment $p \rightsquigarrow \mathsf{Mlistsegof}\, R\, q\, L_1$ and a suffix $q \rightsquigarrow \mathsf{Mlistof}\, R\, L_2$.

$$\begin{array}{l} \forall p L i.\ \{p \rightsquigarrow \mathsf{Mlistof}\, R\, L \star [0 \leqslant i < |L|]\} \\ \quad (\texttt{nth i p}) \\ \quad \{\lambda q.\ \exists L_1 L_2.\ p \rightsquigarrow \mathsf{Mlistsegof}\, R\, q\, L_1 \star q \rightsquigarrow \mathsf{Mlistof}\, R\, L_2 \\ \qquad \star [L = L_1 +\!\!+ L_2 \wedge |L_1| = i]\} \end{array}$$

Note that the pieces from the post-condition can be merged back into the form $p \rightsquigarrow \mathsf{Mlistof}\, R\, L$ by using the rewriting rule for concatenation of list segments presented near the end of 5.1.

In the second case, the specification of `nth` has a smaller footprint: it only describes the prefix segment $p \rightsquigarrow \mathsf{Mlistsegof}\, R\, q\, L$ actually traversed by the function.

$$\begin{array}{l} \forall i p L.\ \{p \rightsquigarrow \mathsf{Mlistsegof}\, R\, q\, L \star [|L| = i]\}\ (\texttt{nth i p}) \\ \qquad \{\lambda r.\ p \rightsquigarrow \mathsf{Mlistsegof}\, R\, q\, L \star [r = q]\} \end{array}$$

Although it requires the user to split ahead of time the list into list segments (which can be achieved using the same rewriting rule for concatenation of list segments), this second specification is much simpler to state.

Consider now the program "`(nth i p).hd <- v`", which calls the function `nth` to obtain the pointer to the cell at index i, and then updates the value stored in this cell. Starting from a heap described by $p \rightsquigarrow \mathsf{Mlistof}\, R\, L$, we first argue that, if i is a valid index in L, then L can be decomposed as $L_1 +\!\!+ X :: L_2$. Using the rewriting rules from §5.1, we then change the heap predicate to:

$$\begin{array}{l} p \rightsquigarrow \mathsf{Mlistsegof}\, R\, q\, L_1 \star q \mapsto \{\!|\mathrm{hd}{=}x;\ \mathrm{tl}{=}r|\!\} \\ \star\, r \rightsquigarrow \mathsf{Mlistof}\, R\, L_2 \star x \rightsquigarrow R\, X. \end{array}$$

for some q, r and x. Exploiting the second specification of `nth` shown above, we establish that the call to `nth i p` returns exactly the address q. The write operation on this cell thus involves the predicate $q \mapsto \{\!|\mathrm{hd}{=}x;\ \mathrm{tl}{=}r|\!\}$. After reasoning about the write operation, the heap predicate may be folded back into an instance of $\mathsf{Mlistof}$ using the same rewriting rules in the other direction.

5.3 Example: Mutable Queues with an Abstract Interface

Through this example, we illustrate the use of list segments in the implementation of a mutable queue data structure. We also present an abstract interface for a mutable queue module, with an interface involving a higher-order representation predicate.

We implement a mutable queue as a list segment, maintaining two pointers: one on the head of the list, and one on the tail of the list. To avoid the need to treat specially the empty queue, we ensure that the underlying list is always made of at least one cell. To that end, we consider that the value stored in the last cell of the list is irrelevant. The record storing the head and tail pointers admits the following signature, where the type `cell` is that defined in §4.1.

```
type 'a queue = { mutable front : 'a cell;
                  mutable back : 'a cell; }
```

We introduce the representation predicate $p \rightsquigarrow \mathsf{Queueof}\, R\, L$, to assert that, at address p in memory, there exists a mutable queue represented by the list L, when the elements stored are represented by the representation predicate R. The definition, shown below, asserts the existence of the queue record, the existence of the list segment that begins on the front pointer and reaches the back pointer, and the existence of the last cell, at the back of the queue, whose contents is ignored.

$$\begin{array}{lll} p \rightsquigarrow \mathsf{Queueof}\, R\, L & \equiv \exists f b. & p \mapsto \{\!|\mathrm{front}{=}f;\ \mathrm{back}{=}b|\!\} \\ & & \star\, f \rightsquigarrow \mathsf{Mlistsegof}\, R\, b\, L \\ & & \star\, \exists x y.\ b \mapsto \{\!|\mathrm{hd}{=}x;\ \mathrm{tl}{=}y|\!\} \end{array}$$

Figure 1 shows an abstract specification for a queue module. The function `create` produces a queue represented by the empty list. The function `is_empty` returns a boolean b indicating whether the queue is empty. This function leaves the queue unaltered. The function `push` adds an element described by $x \rightsquigarrow R\, X$ to the back of the queue. Note that `push` effectively transfers the ownership of x to the queue.

The function `pop` extracts an element from the front of the queue. Symmetrical to `push`, `pop` transfers back to the client the ownership of the front element. A second specification is also provided for `pop`. Although this second specification is more verbose than the former, it is slightly more convenient to use in practice—it saves the need to perform, beforehand, a case analysis on the list.

The function `peek` returns the element at the front of the queue, without removing it from the queue. As explained in the introduction, this function can be elegantly specified using the magic wand. The magic wand, written $H' \mathrel{-\!\!\star} H$, describes a piece of heap, call it H'', such that, if H'' is extended with H', then the result would be exactly H. It is defined as follows. (Remark: the equality that appears below could also be weakened into an entailment relation.)

$$H' \mathrel{-\!\!\star} H \quad \equiv \quad \exists H''.\ H'' \star [H' \star H'' = H]$$

Last, the function `transfer` migrates all the elements from the second queue at the back of the first queue. It thus leaves the second queue empty.

Although we do not show the details here, we are able to prove that the particular queue implementation based on list segments described previously does satisfy the abstract specification for queues.

5.4 From First-Order to Higher-Order Specifications

The specifications of a mutable queue presented in Figure 1 account for the fact that the queue structure owns its elements. This ownership of the elements is very convenient for the client of the queue module. Indeed, if the specifications were not accounting for the ownership of the elements stored in the queue, then, during the verification process, we would need to separately maintain the ownership of a queue of pointer values, and a set of the elements contained in the queue. In practice, the use of a higher-order representation predicate typically saves a lot of proof effort to the client of a queue data structure.

However, for the implementer of queue data structures, the verification with respect to a higher-order representation predicate involves a little bit more work than the verification with respect to a first-order representation predicate. Indeed, with a first-order representation predicate, invariants and specifications are a slightly simpler, as they need not mention representation predicates for the elements, such as $x \rightsquigarrow R\, X$.

Interestingly, though, we can prove once and for all that any specification of a queue expressed using a first-order representation predicate can be turned into a specification expressed using a higher-order representation predicate. The idea is that, given any heap predicate of the form $p \rightsquigarrow \mathsf{Queue}\, L$, we can derive a definition of $p \rightsquigarrow \mathsf{Queueof}\, R\, L$ using a iterated star over the elements stored in the queue (recall §4.3), and we can then lift the specifications expressed in terms of $p \rightsquigarrow \mathsf{Queue}\, L$ into corresponding specifications expressed in terms of $p \rightsquigarrow \mathsf{Queueof}\, R\, L$. We give below an example below illustrating the lifting of the specifications of `push`. Recall that $L \& X$ is a shorthand for $L +\!\!+ (X :: \mathsf{nil})$.

$$\begin{array}{l} \forall\, \mathsf{Queue}. \\ \left(\begin{array}{ll} & p \rightsquigarrow \mathsf{Queueof}\, R\, L\ = \\ \forall p R L. & \quad \exists K.\ [\,|K| = |L|\,] \star p \rightsquigarrow \mathsf{Queue}\, K \\ & \qquad \star \circledast_{i \in [0, |L|)} (K[i]) \rightsquigarrow R\, (L[i]) \end{array}\right) \\ \wedge\ \big(\forall x p L.\, \{p \rightsquigarrow \mathsf{Queue}\, L\}\ (\texttt{push}\, x\, p)\ \{\lambda\_.\, p \rightsquigarrow \mathsf{Queue}\, (L \& x)\}\big) \\ \Rightarrow\ \big(\forall x p X L.\, \{p \rightsquigarrow \mathsf{Queueof}\, R\, L \star x \rightsquigarrow R\, X\}\ (\texttt{push}\, x\, p) \\ \qquad\qquad \{\lambda\_.\ p \rightsquigarrow \mathsf{Queueof}\, R\, (L \& X)\}\big) \end{array}$$

$$\begin{array}{lll}
\{[\,] & \}\ (\texttt{create()})\ \{ & \lambda p.\ p \rightsquigarrow \mathsf{Queueof}\, R\, \mathsf{nil}\} \\
\{p \rightsquigarrow \mathsf{Queueof}\, R\, L & \}\ (\texttt{is\_empty}\, p)\ \{ & \lambda b.\ [b = \mathsf{true} \Leftrightarrow L = \mathsf{nil}] \star p \rightsquigarrow \mathsf{Queueof}\, R\, L\} \\
\{p \rightsquigarrow \mathsf{Queueof}\, R\, L \star x \rightsquigarrow R\, X & \}\ (\texttt{push}\, x\, p)\ \{ & \lambda\_.\ p \rightsquigarrow \mathsf{Queueof}\, R\, (L \& X)\} \\
\{p \rightsquigarrow \mathsf{Queueof}\, R\, (X :: L) & \}\ (\texttt{pop}\, p)\ \{ & \lambda x.\ p \rightsquigarrow \mathsf{Queueof}\, R\, L \star x \rightsquigarrow R\, X\} \\
\{p \rightsquigarrow \mathsf{Queueof}\, R\, L \star [L \neq \mathsf{nil}] & \}\ (\texttt{pop}\, p)\ \{ & \lambda x.\ \exists X L'.\ [L = X :: L'] \star p \rightsquigarrow \mathsf{Queueof}\, R\, L' \star x \rightsquigarrow R\, X\} \\
\{p \rightsquigarrow \mathsf{Queueof}\, R\, (X :: L) & \}\ (\texttt{peek}\, p)\ \{ & \lambda x.\ x \rightsquigarrow R\, X \star (x \rightsquigarrow R\, X \ \mathrel{-\!\!\star}\ p \rightsquigarrow \mathsf{Queueof}\, R\, (X :: L))\} \\
\{p_1 \rightsquigarrow \mathsf{Queueof}\, R\, L_1 \star p_2 \rightsquigarrow \mathsf{Queueof}\, R\, L_2 & \}\ (\texttt{transfer}\, p_1\, p_2)\ \{ & \lambda\_.\ p_1 \rightsquigarrow \mathsf{Queueof}\, R\, (L_1 +\!\!+ L_2) \star p_2 \rightsquigarrow \mathsf{Queueof}\, R\, \mathsf{nil}\}
\end{array}$$

Figure 1. Mutable queue specification that includes ownership transfer for the elements.

Note that the same kind of lifting lemma could be derived just as well for any other specification from Figure 1.

In summary, specifications of a data structure with higher-order representation predicates are very useful for reasoning about client code, and there are two ways to establish them: either by verifying an implementation of the data structure directly with respect to the higher-order representation predicate, or by verifying it only with respect to a first-order representation predicate and then lifting the specifications to their higher-order counterparts. The second approach typically pays off when verifying several implementations that satisfy a same interface, e.g. several queue data structures.

6. Records

In the previous section, we introduced a higher-order representation predicate called Mlistof to describe mutable lists. In this section, we apply a similar idea to records. For each record type, we generate a higher-order representation predicate that controls, on a per-field basis, whether the contents of each field should be owned.

6.1 Representation Predicate for List Cells

Recall the definition of Mlistof from §4.4. This definition involves, for the case of nonempty lists, the following heap predicate:

$$\exists x p'.\, p \mapsto \{\!|\mathrm{hd}{=}x;\ \mathrm{tl}{=}p'|\!\} \star x \rightsquigarrow R\, X \star p' \rightsquigarrow \mathsf{Mlistof}\, R\, L'.$$

This predicate is made of three parts: (1) a heap predicate describing a single list cell made of two fields, (2) a heap predicate describing the contents of the first field of this record, and (3) a heap predicate describing the contents of the second field of this record.

We introduce a higher-order representation predicate for list cells, written $p \rightsquigarrow \mathsf{Mcellof}\, R_1\, V_1\, R_2\, V_2$. This predicate describes a list cell at location p, and relies on the representation predicates R_1 and R_2 for describing the content of the two fields. We thereby establish a direct relationship with the corresponding logical values V_1 and V_2 describing the logical content of the two fields. The predicate Mcellof is defined as follows.

$$\begin{array}{lll}
p \rightsquigarrow \mathsf{Mcellof}\, R_1\, V_1\, R_2\, V_2 & \equiv & \exists v_1 v_2.\quad p \mapsto \{\!|\mathrm{hd}{=}v_1;\ \mathrm{tl}{=}v_2|\!\} \\
& & \qquad\quad \star\ v_1 \rightsquigarrow R_1\, V_1 \\
& & \qquad\quad \star\ v_2 \rightsquigarrow R_2\, V_2
\end{array}$$

Remark: when writing $p \rightsquigarrow \mathsf{Mcellof}\, R_1\, V_1\, R_2\, V_2$, the names of the fields hd and tl no longer appear explicitly. One could nevertheless set up a custom notation to make these names appear explicitly.

Using the predicate Mcellof, we revisit the definition of Mlistof and express it in a much more concise manner, as shown below.

$$\begin{array}{lll}
p \rightsquigarrow \mathsf{Mlistof}\, R\, L & \equiv & \mathsf{match}\, L\, \mathsf{with} \\
& & \mid \mathsf{nil} \Rightarrow [p = \mathsf{null}] \\
& & \mid X :: L' \Rightarrow \\
& & \quad p \rightsquigarrow \mathsf{Mcellof}\, R\, X\, (\mathsf{Mlistof}\, R)\, L'
\end{array}$$

Technical remark: it is no longer obvious that the definition of Mlistof shown above is structurally recursive on its argument L. Nevertheless, Coq accepts it thanks to its ability to unfold on-the-fly the definition of Mcellof when checking the guard condition.

In §6.3, we will explain how to specify, in terms of Mcellof, read and write operations in record fields. But first, we consider the simpler case of references, which consist of single-field records.

6.2 Representation Predicate for References

A reference is a record with a single field named contents. The higher-order representation predicate for references is as follows.

$$p \rightsquigarrow \mathsf{Refof}\, R\, V \;\equiv\; \exists v.\ p \mapsto \{\!|\mathrm{contents}{=}v|\!\} \star v \rightsquigarrow R\, V$$

In general, given a heap described by $p \rightsquigarrow \mathsf{Refof}\, R\, V$, it is not possible to directly specify the behavior of a read operation into the reference without first making the existentially-quantified value v appear explicitly. To make v appear, it would suffice to unfold the definition of Refof. However, doing so would mean that the manipulation of references would involve two different predicates, one of the form $p \rightsquigarrow \mathsf{Refof}\, R\, V$ and one of the form $p \mapsto \{\!|\mathrm{contents}{=}v|\!\}$. We find it more effective to specify read and write operations on references only in terms of the predicate Refof. Moreover, specifications expressed in terms of the representation predicate generalize better to the case of records with several fields.

To that end, we rely on the identity representation predicate Id. Recall from §4.4 that Id is defined as $\lambda X x.\ [x = X]$. We have:

$$p \rightsquigarrow \mathsf{Refof}\, \mathsf{Id}\, v \;=\; p \mapsto \{\!|\mathrm{contents}{=}v|\!\}.$$

Thus, when the heap is described by $p \rightsquigarrow \mathsf{Refof}\, \mathsf{Id}\, v$, we can specify that a read in p returns exactly the value v. In summary, to read a reference described by a heap predicate $p \rightsquigarrow \mathsf{Refof}\, R\, v$, we can first *focus* on its content field, by exploiting the equality:

$$p \rightsquigarrow \mathsf{Refof}\, R\, V \;=\; \exists v.\ p \rightsquigarrow \mathsf{Refof}\, \mathsf{Id}\, v \star v \rightsquigarrow R\, V$$

then we can read in the reference using the predicate $p \rightsquigarrow \mathsf{Refof}\, \mathsf{Id}\, v$, obtaining the value v. Once we are done with manipulating v, and possibly after modifying the structure that v might point to, we may fold back to the form $p \rightsquigarrow \mathsf{Refof}\, R\, V'$, for some V', by exploiting the same equality in the reverse direction.

The operations for creating, reading, and writing in *focused* reference are specified as follows. Note that these specifications are simply an alternative presentation of the axiomatic specifications of record operations, as provided by the program logic.

$$\begin{array}{l}
\{[\,]\}\ (\mathsf{ref}\, v)\ \{\lambda p.\ p \rightsquigarrow \mathsf{Refof}\, \mathsf{Id}\, v\} \\
\{p \rightsquigarrow \mathsf{Refof}\, \mathsf{Id}\, v\}\ (\mathsf{get}\, p)\ \{\lambda x.\ p \rightsquigarrow \mathsf{Refof}\, \mathsf{Id}\, v \star [x = v]\} \\
\{p \rightsquigarrow \mathsf{Refof}\, \mathsf{Id}\, v\}\ (\mathsf{set}\, p\, w)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Refof}\, \mathsf{Id}\, w\}
\end{array}$$

It is also possible to derive specifications that, to some extent, operate on *unfocused* references, described by a predicate of the form $p \rightsquigarrow \mathsf{Refof}\, R\, V$ for an arbitrary R. These specifications, shown below, combine the specifications shown above with ownership transfer of the contents of the reference.

$$\{v \rightsquigarrow R\,V\}\ (\mathsf{ref}\,v)\ \{\lambda p.\ p \rightsquigarrow \mathsf{Refof}\,R\,V\}$$

$$\{p \rightsquigarrow \mathsf{Refof}\,R\,V\}\ (\mathsf{get}\,p)\ \{\lambda v.\ p \rightsquigarrow \mathsf{Refof}\,\mathsf{Id}\,v \star v \rightsquigarrow R\,V\}$$

$$\{p \rightsquigarrow \mathsf{Refof}\,R\,V \star w \rightsquigarrow R'\,V'\}\ (\mathsf{set}\,p\,w)$$
$$\{\lambda\_.\ p \rightsquigarrow \mathsf{Refof}\,R'\,V' \star (\exists v.\ v \rightsquigarrow R\,V)\}$$

In the specification of set, the post-condition includes the description of a piece of heap $\exists v.\ v \rightsquigarrow R\,V$, which typically corresponds to an object that has become inaccessible. On the one hand, if the program logic is affine, pieces of heaps can be safely discarded, so it is safe to drop $\exists v.\ v \rightsquigarrow R\,V$ from the post-condition of set. On the other hand, if the program logic is linear, then the heap predicate $\exists v.\ v \rightsquigarrow R\,V$ likely indicates a memory leak whenever R is not equal to Id.

6.3 Access to Record Fields

We now come back to the study of the list predicate Mcellof, which we use to illustrate how to specify focus operations as well as read and write operations on records with multiple fields.

Unfolding the definition of Mcellof (given in §6.1) allows one to convert from a view where all fields are owned to a view where none of the fields is owned. If, however, instead of unfolding the definition of Mcellof, we use rewriting rules to introduce the identity representation predicate Id, then we are able to control, on a per-field basis, which fields should be *focused* (i.e. ready for read and write operations) and which fields should remain *unfocused* (i.e. owned). Having a fine-grained control over which fields should be focused is particularly useful when a record has many fields but only a few of them need to be accessed, as it greatly reduces the number of variables that need to be introduced.

The rewriting rule shown below allows one to focus on the first field when it is applied from left to right, and, symmetrically, to unfocus on this field when it is applied from right to left.

$$p \rightsquigarrow \mathsf{Mcellof}\,R_1\,V_1\,R_2\,V_2 \;=\; \exists v_1.\ \ p \rightsquigarrow \mathsf{Mcellof}\,\mathsf{Id}\,v_1\,R_2\,V_2 \star v_1 \rightsquigarrow R_1\,V_1$$

Above, observe that, when operating on the first field, the second field can be either focused or unfocused: R_2 may be any representation predicate, including Id. A symmetrical rule (not shown) allows for focusing or unfocusing on the second field of the record.

Like we did previously for references (in §6.2), we provide specifications for read and write operations stated in terms of the predicate Mcellof. They are restricted to fields that are focused, i.e. that are represented using Id. For example, to read and write in the head field, we exploit the following specifications.

$$\{p \rightsquigarrow \mathsf{Mcellof}\,\mathsf{Id}\,v_1\,R_2\,V_2\}\ (p.\mathsf{hd})$$
$$\{\lambda x.\,p \rightsquigarrow \mathsf{Mcellof}\,\mathsf{Id}\,v_1\,R_2\,V_2 \star [x = v_1]\}$$

$$\{p \rightsquigarrow \mathsf{Mcellof}\,\mathsf{Id}\,v_1\,R_2\,V_2\}\ (p.\mathsf{hd}\texttt{<-}w)$$
$$\{\lambda\_.\ p \rightsquigarrow \mathsf{Mcellof}\,\mathsf{Id}\,w\,R_2\,V_2\}$$

Note that the focus/unfocus rewriting rules and the specifications above are automatically generated by the tool CFML given the signature of the record that appears in the source code.

Here again, as done previously for references (in §6.2), we derive specifications that account for ownership transfer. For example, we show next the specifications of allocation and read operations.

$$\{v_1 \rightsquigarrow R_1\,V_1 \star v_2 \rightsquigarrow R_2\,V_2\}\ (\{\!|\text{hd}{=}v_1;\ \text{tl}{=}v_2|\!\})$$
$$\{\lambda p.\ p \rightsquigarrow \mathsf{Mcellof}\,R_1\,V_1\,R_2\,V_2\}$$

$$\{p \rightsquigarrow \mathsf{Mcellof}\,R_1\,V_1\,R_2\,V_2\}\ (p.\mathsf{hd})$$
$$\{\lambda v_1.\ \exists R_1.\ p \rightsquigarrow \mathsf{Mcellof}\,\mathsf{Id}\,v_1\,R_2\,V_2 \star\ v_1 \rightsquigarrow R_1\,V_1\}$$

In summary, read and write operations on records can be assigned (at least) three different types of specifications: (1) specifications that operate on the low-level heap predicate $p \mapsto \{\!|\text{hd}{=}v_1;\ \text{tl}{=}v_2|\!\}$, (2) specifications that operate on a focused predicate of the form $p \rightsquigarrow \mathsf{Mcellof}\,\mathsf{Id}\,v_1\,R_2\,V_2$, and (3) specifications that perform on-the-fly focus operations starting from an unfocused predicate of the form $p \rightsquigarrow \mathsf{Mcellof}\,R_1\,V_1\,R_2\,V_2$. As we argued, in order to limit the number of heap predicates, we choose to never manipulate the low-level heap predicate as in (1), but to only work in terms of Mcellof, using specifications of type (2) and (3).

We have found, when carrying proofs in CFML, that having access to both specifications of type (2) and (3) helps shorten the proof scripts significantly. That said, in a system with a sufficiently-high degree of automation for focus and unfocus operations, it might be possible for specifications of type (3) to be automatically derived from specifications of type (2), in which case only specifications of type (2) would need to be stated explicitly.

7. Trees

We next generalize our results from lists to trees. We begin with binary trees, in particular binary search trees, then cover n-ary trees, and bootstrapped trees, which involve polymorphic recursion. We also present a technique for specifying trees in which items or subtrees have been carved out.

7.1 Implementation of Binary Trees

We consider binary trees with items stored in the nodes, implemented as records with three fields, as shown below. Leaves are represented with the null value.

```
type 'a node = { mutable item : 'a;
                 mutable left : 'a node;
                 mutable right : 'a node; }
```

To describe such trees in the logic, we introduce a type of purely functional trees. In Coq, this datatype is defined as follows.

```
Inductive tree (A:Type) : Type :=
  | Leaf : tree A
  | Node : A → tree A → tree A → tree A.
```

7.2 Specification of Binary Trees

The representation predicate for binary trees follows the exact same construction as that of the predicate Mlistof introduced in §4.4.

$$\begin{aligned}
p \rightsquigarrow \mathsf{Mtreeof}\,R\,T \;\equiv\; & \mathsf{match}\,T\,\mathsf{with}\\
& \mid \mathsf{Leaf} \Rightarrow [p = \mathsf{null}]\\
& \mid \mathsf{Node}\,X\,T_1\,T_2 \Rightarrow \exists x p_1 p_2.\\
& \qquad p \mapsto \{\!|\text{item}{=}x;\ \text{left}{=}p_1;\ \text{right}{=}p_2|\!\}\\
& \qquad \star\ x \rightsquigarrow R\,X\\
& \qquad \star\ p_1 \rightsquigarrow \mathsf{Mtreeof}\,R\,T_1\\
& \qquad \star\ p_2 \rightsquigarrow \mathsf{Mtreeof}\,R\,T_2
\end{aligned}$$

Following the introduction of the higher-order representation predicate Mcellof for list cells (recall §6.1), we introduce a predicate Nodeof for describing node cells.

$$\begin{aligned}
& p \rightsquigarrow \mathsf{Nodeof}\,R_1\,V_1\,R_2\,V_2\,R_3\,V_3\\
\equiv\ & \exists v_1 v_2 v_3.\quad p \mapsto \{\!|\text{item}{=}v_1;\ \text{left}{=}v_2;\ \text{right}{=}v_3|\!\}\\
& \star\ v_1 \rightsquigarrow R_1\,V_1 \star v_2 \rightsquigarrow R_2\,V_2 \star v_3 \rightsquigarrow R_3\,V_3
\end{aligned}$$

Using Nodeof, the body of the definition of $p \rightsquigarrow \mathsf{Mtreeof}\,R\,T$ can be simplified as follows.

$$\begin{aligned}
p \rightsquigarrow \mathsf{Mtreeof}\,R\,T \;\equiv\; & \mathsf{match}\,T\,\mathsf{with}\\
& \mid \mathsf{Leaf} \Rightarrow [p = \mathsf{null}]\\
& \mid \mathsf{Node}\,X\,T_1\,T_2 \Rightarrow\\
& \qquad p \rightsquigarrow \mathsf{Nodeof}\,R\,X\,(\mathsf{Mtreeof}\,R)\,T_1\\
& \qquad\qquad (\mathsf{Mtreeof}\,R)\,T_2
\end{aligned}$$

7.3 Binary Search Trees

A binary search tree is a binary tree satisfying specific invariants: smaller values in the left subtree, greater values in the right subtree. There are two approaches to enforcing those invariants. The first approach consists of extending the predicate $\mathsf{Mtreeof}$ to augment it with additional invariants. The second approach consists of reusing the predicate $\mathsf{Mtreeof}$ unchanged, and separately enforcing invariants on the logical tree (i.e. the tree written T above). The second approach has two major benefits over the former: it enables us to share the definition of $\mathsf{Mtreeof}$ between all binary trees implementation, and it allows us to state and prove lemmas about tree operations completely outside of the Separation Logic fragment, that is, through reasoning that only involves pure predicates over logical trees. In what follows, we apply the second approach described above, first to the case of a binary search tree storing integers, and then to a binary search tree storing arbitrary mutable objects.

For the case of integers, we introduce a heap predicate, written $p \rightsquigarrow \mathsf{Msearchtree}\, E$, to assert that the binary search tree rooted at location p represents a set of integers E. This predicate asserts the existence of a pure tree T, such that the tree rooted at p describes T, and such that T satisfies the invariants of being a search tree representing E. The latter property is characterized by an auxiliary inductively-defined predicate, called search, which is parameterized by the order relation on the items, written $\prec$. Note that, the representation predicate $\mathsf{Mtreeof}$ is applied to Id because in this first example the items are just integers and thus do not point to other pieces of heap. The definitions are as follows.

$$p \rightsquigarrow \mathsf{Msearchtree}\, E \quad \equiv \quad \exists T.\ p \rightsquigarrow \mathsf{Mtreeof}\, \mathsf{Id}\, T \star [\mathsf{search}_{\prec}\, T\, E]$$

$$\frac{}{\mathsf{search}_{\prec}\ \mathsf{Leaf}\, \varnothing} \qquad \frac{\mathsf{search}\, T_1\, E_1 \quad \mathsf{search}\, T_2\, E_2 \quad \forall y \in E_1.\ y \prec x \quad \forall y \in E_2.\ x \prec y}{\mathsf{search}_{\prec}\ (\mathsf{Node}\, x\, T_1\, T_2)\ (\{x\} \cup E_1 \cup E_2)}$$

For the case of trees with items that are arbitrary mutable objects, we need to assume a representation predicate R for the items. We also need to specify the comparison function used for comparing items. It is specified as follows.

$$\begin{array}{l} \forall v_1 v_2 V_1 V_2.\ \{v_1 \rightsquigarrow R\, V_1 \ \star\ v_2 \rightsquigarrow R\, V_2\} \\ \qquad (\texttt{compare}\ v_1\ v_2) \\ \qquad \{\lambda n.\ v_1 \rightsquigarrow R\, V_1 \ \star\ v_2 \rightsquigarrow R\, V_2 \star \qquad\qquad \} \\ \qquad\quad [\mathsf{if}\ n = 0\ \mathsf{then}\ V_1 = V_2\ \mathsf{else} \\ \qquad\quad\ \mathsf{if}\ n < 0\ \mathsf{then}\ V_1 \prec V_2\ \mathsf{else}\ V_2 \prec V_1] \end{array}$$

Remark: for simplicity, we assume here the existence of a total order over the items, although in general two items might be considered equivalent even when the items are not logically equal, and we moreover assume that comparison function to only depend on the items being compared, and not on a global state.

The higher-order version of the binary search tree representation predicate, parameterized by $\prec$ and R, appears next.

$$\begin{array}{l} p \rightsquigarrow \mathsf{Msearchtreeof}_{\prec}\, R\, E \\ \equiv\ \exists T.\ p \rightsquigarrow \mathsf{Mtreeof}\, R\, T\ \star\ [\mathsf{search}_{\prec}\, T\, E] \end{array}$$

Note that the above definition is polymorphic in both a, the type of items in the program, and A, the type at which items are described in the logic. The representation predicate R has type $A \to a \to \mathsf{Hprop}$, the order relation $(\prec)$ has type $A \to A \to \mathsf{Prop}$, the tree T has type $\mathsf{tree}\, A$, and the set E has type $\mathsf{set}\, A$, where set corresponds to the built-in sets from the logic.

7.4 Trees with List of Subtrees

We next consider a slightly more challenging tree structure, in which each tree node consists of an item and a mutable list of subtrees. In the OCaml type definition shown below, `cell` is the type of list cells introduced in §4.1.

```
type 'a node = {
  mutable item : 'a;
  mutable children : ('a node) cell }
```

The higher-order representation predicate for nodes follows the exact same pattern as for other records (recall §6.1).

$$\begin{array}{rl} p \rightsquigarrow \mathsf{Nodeof}\, R_1\, V_1\, R_2\, V_2 \ \equiv & \exists v_1 v_2. \\ & p \mapsto \{\!|\mathsf{item}{=}v_1;\ \mathsf{children}{=}v_2|\!\} \\ & \star\ v_1 \rightsquigarrow R_1\, V_1\ \star\ v_2 \rightsquigarrow R_2\, V_2 \end{array}$$

The trees are represented in the logic by the following datatype.

```
Inductive tree (A:Type) : Type :=
  | Leaf : tree A
  | Node : A → list (tree A) → tree A.
```

The higher-order representation predicate $p \rightsquigarrow \mathsf{Narytreeof}\, R\, T$ describes trees with list of subtrees. In the definition shown below, the key ingredient is the use of the representation predicate $\mathsf{Mlistof}\,(\mathsf{Narytreeof}\, R)$ for describing the mutable list of subtrees.

$$\begin{array}{l} p \rightsquigarrow \mathsf{Narytreeof}\, R\, T \equiv \\ \quad \mathsf{match}\, T\, \mathsf{with} \\ \quad |\, \mathsf{Leaf}\ \Rightarrow\ [p = \mathsf{null}] \\ \quad |\, \mathsf{Node}\, X\, L\ \Rightarrow\ p \rightsquigarrow \mathsf{Nodeof}\, R\, X\, (\mathsf{Mlistof}\,(\mathsf{Narytreeof}\, R))\, L \end{array}$$

In summary, by composing higher-order representation predicates to obtain $\mathsf{Mlistof}\,(\mathsf{Narytreeof}\, R)$, we are able to describe in a very concise manner the ownership of a mutable list and of all the subtrees whose roots are stored in that list.

7.5 Bootstrapped Trees

In this section, we consider a data structure called *bootstrapped chunked bags* to illustrate the design of higher-order representation predicates for data structures involving polymorphic recursion. This structure represents a bag (i.e. an unordered multiset of values) that supports push and pop operations in constant time, and supports merge and split operations in logarithmic time.[3] These operations are particularly useful in the design of parallel algorithms.

Bootstrapped chunked bags are parameterized by a constant `k`, and are constructed using chunks. A chunk is an array of capacity `k`, and thus can be used to represent bags storing up to `k` items. A bootstrapped chunked bag consists of several *layers*. The first layer contains a chunk of items. The second layer contains a chunk of chunks of items. The third layer contains a chunk of chunks of chunks of items, and so on.

For example, assume `k` = 10. Then, a bag storing 162 items could be represented with a first-layer chunk storing 2 items, a second-layer chunk storing 6 chunks of 10 items each, and a third layer chunk that stores a single chunk, which itself contains 10 chunks of 10 items each.

In what follows, we omit the details of the representation of chunks. We simply assume a type `'a chunk` for chunks, and a corresponding representation predicate, written $c \rightsquigarrow \mathsf{Chunkof}\, R\, E$, to assert that at location c there exists a chunk that represents a multiset of items E, when these items are reflected in the logic using the representation predicate R.

An empty bag is represented as the null pointer. A nonempty bag is represented as a record made of a chunk of elements (the current layer) and a bag of chunks of elements (the next layers).

[3] Remark: the bootstrapped chunked bags presented here are a simplification of *bootstrapped chunked sequences* [1], which were recently proposed as an alternative to finger trees [7] and to an earlier structure by Tarjan et al [8], for improving constant factors and space consumption. Bootstrapped chunked bags may also be viewed as a generalization of the bootstrapped catenable lists presented by Okasaki [13].

Note that elements stored at the first layer are base items, whereas elements stored in the deeper layers are pointers on chunks.

```
  type 'a bag = { mutable head : 'a chunk;
                  mutable next : ('a chunk) bag }
```

The higher-order representation predicate associated with this record type is defined following the same pattern as before.

$$p \rightsquigarrow \mathsf{Nodeof}\, R_1\, V_1\, R_2\, V_2 \;\equiv\; \exists v_1 v_2.\; p \mapsto \{\!|\text{head}{=}v_1;\ \text{next}{=}v_2|\!\} \star v_1 \rightsquigarrow R_1\, V_1 \star v_2 \rightsquigarrow R_2\, V_2$$

The layer structure is represented in the logic as shown below.

```
Inductive layer : Type → Type :=
  | Empty : ∀A, layer A
  | Layer : ∀A, multiset A → layer (multiset A) → layer A.
```

To relate the memory layout of bootstrapped chunked bag with its logical model, we introduce the predicate $p \rightsquigarrow \mathsf{Layersof}\, R\, T$. In the definition shown below, a nonempty layer consists of two parts: a chunk of items, represented by a multiset E', and a bag of chunks, described by the representation predicate $\mathsf{Layersof}\,(\mathsf{Chunkof}\, R)$.

$$\begin{aligned} &p \rightsquigarrow \mathsf{Layersof}\, R\, T \equiv\\ &\quad \mathsf{match}\ T\ \mathsf{with}\\ &\quad\ |\ \mathsf{Empty} \Rightarrow [p = \mathsf{null}]\\ &\quad\ |\ \mathsf{Layer}\, E'\, T' \Rightarrow\\ &\qquad p \rightsquigarrow \mathsf{Nodeof}\,(\mathsf{Chunkof}\, R)\, E'\, (\mathsf{Layersof}\,(\mathsf{Chunkof} R))\, T' \end{aligned}$$

Observe the polymorphic recursion at play here: the argument R of Layersof becomes instantiated at the next level with $\mathsf{Chunkof}\, R$. Thus, when unfolding the recursive definitions, the representation predicates associated with deeper layers take the form: $\mathsf{Chunkof}\,(\mathsf{Chunkof} \ldots (\mathsf{Chunkof}\, R)\ldots)$.

It remains to define a heap predicate, written $p \rightsquigarrow \mathsf{Bagof}\, R\, E$, to assert that, at the memory location p, there exists a chunked bag data structure that represents the multiset E, when items are represented using the representation predicate R. The definition, shown below, involves an auxiliary inductively-defined predicate $\mathsf{Layerbag}\, T\, E$, which asserts that E is the multiset obtained by collecting all the elements stored in the leaves of the tree T.

$$p \rightsquigarrow \mathsf{Bagof}\, R\, E \;\equiv\; \exists T.\; p \rightsquigarrow \mathsf{Layersof}\, R\, T \star [\mathsf{Layerbag}\, T\, E]$$

$$\frac{}{\mathsf{Layerbag}\ \mathsf{Empty}\ \varnothing} \qquad \frac{\mathsf{Layerbag}\, T'\, G}{\mathsf{Layerbag}\,(\mathsf{Layer}\, E'\, T')\,(E' \uplus \mathsf{flatten}\, G)}$$

Above, flatten is an operation that takes as argument a multiset of multisets of items, and returns the union of these multisets.

The predicate $p \rightsquigarrow \mathsf{Bagof}\, R\, E$ is involved in the specification of bag operations. For example, the specification of `push` is shown below. It is very similar to that for mutable queues (from Figure 1).

$$\{p \rightsquigarrow \mathsf{Bagof}\, R\, E \star x \rightsquigarrow R\, X\}\ (\mathtt{push}\, x\, p)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Bagof}\, R\,(E \uplus \{X\})\}$$

In conclusion, data structures involving polymorphic recursion can be naturally described using a higher-order representation predicate in which the argument R gets instantiated on recursive calls with the representation predicate itself, here $\mathsf{Chunkof}\, R$.

7.6 Trees with Holes and Cut Subtrees

In this section, we present a technique for allowing one to detach the ownership of some items from a tree that owns its items (tree with holes), and/or to detach one or more entire subtrees from the tree (tree with cut subtrees). We present this technique using as running example the binary trees defined in §7.1. The predicate Nodeof, which we use thereafter for describing tree nodes, is that introduced in §7.2.

The central idea for describing trees with holes is to extend the type `tree` that is used to describe binary trees in the logic with two additional constructors, called `Hole` and `Cut`. First, the constructor `Hole` is similar to the constructor `Node` describing nodes, except that the item of the node considered is not owned: it is described using Id instead of R, where R is the representation predicate R normally describing the items. Second, the constructor `Cut` takes one location as argument. The intention is that `Cut` q describes the fact that the subtree rooted at q has been cut out from the main tree. Note that the contents of the cut out subtree is not described, in the sense that the constructor `Cut` appears like a leaf in the logical tree.

```
Inductive tree (a:Type) (A:Type) : Type :=
  | Leaf : tree A
  | Node : A → tree A → tree A → tree A
  | Hole : a → tree A → tree A → tree A
  | Cut : loc → tree A.
```

We then generalize the representation predicate $p \rightsquigarrow \mathsf{Mtreeof}\, R\, T$ to account for the two new constructors. In particular, when T is of the form $\mathsf{Cut}\, q$, we simply enforce that q be equal to p, somewhat like we did for list segments. The updated definition appears next.

$$\begin{aligned} &p \rightsquigarrow \mathsf{Mtreeof}\, R\, T\\ &\equiv \mathsf{match}\ T\ \mathsf{with}\\ &\quad |\ \mathsf{Leaf} \Rightarrow [p = \mathsf{null}]\\ &\quad |\ \mathsf{Node}\, X\, T_1\, T_2 \Rightarrow\\ &\qquad p \rightsquigarrow \mathsf{Nodeof}\, R\, X\, (\mathsf{Mtreeof}\, R)\, T_1\, (\mathsf{Mtreeof}\, R)\, T_2\\ &\quad |\ \mathsf{Hole}\, x\, T_1\, T_2 \Rightarrow\\ &\qquad p \rightsquigarrow \mathsf{Nodeof}\, \mathsf{Id}\, x\, (\mathsf{Mtreeof}\, R)\, T_1\, (\mathsf{Mtreeof}\, R)\, T_2\\ &\quad |\ \mathsf{Cut}\, q \Rightarrow [p = q] \end{aligned}$$

We may convert between a Node and a Hole as follows.

$$\begin{aligned} &p \rightsquigarrow \mathsf{Mtreeof}\, R\,(\mathsf{Node}\, X\, T_1\, T_2)\\ =\;& \exists x.\ p \rightsquigarrow \mathsf{Mtreeof}\, R\,(\mathsf{Hole}\, x\, T_1\, T_2) \star x \rightsquigarrow R\, X \end{aligned}$$

For creating Hole and Cut nodes in-depth in the tree, we need tree surgery operations. To that end, we define a notion of path and substitution in trees. A path, written i, is represented as a list of booleans, with each boolean indicating whether to go down a left branch or a right branch. Note that a valid path would never go down through a Leaf or a Cut constructor. A predicate (whose definition is not shown), written $\mathsf{path}\, i\, T\, T'$, asserts that i is a valid path in T, and that T' is the subtree reached by i. A substitution operation (whose definition is not shown), written $\mathsf{subst}\, i\, T'\, T$, replaces the subtree at path i in T with T'. Note that this operation is undefined if i is not a valid path in T.

With these definitions, we are equipped to state the definitions of the central tree surgery lemma, which asserts that the subtree found at a valid path can be cut out from the main tree, or, reciprocally, merged back into the tree.

$$\begin{aligned} &\mathsf{path}\, i\, T\, T' \Rightarrow\\ &p \rightsquigarrow \mathsf{Mtreeof}\, R\, T \;=\; \exists q.\ \ p \rightsquigarrow \mathsf{Mtreeof}\, R\,(\mathsf{subst}\, i\,(\mathsf{Cut}\, q)\, T) \star q \rightsquigarrow \mathsf{Mtreeof}\, R\, T' \end{aligned}$$

It remains to give the specification of the function `find`, which takes a path as argument and returns a pointer to the corresponding subtree. If we assume that the targeted subtree has already been isolated in the logic, then we can easily specify the return value of `find`, as shown below.

$$\{p \rightsquigarrow \mathsf{Mtreeof}\, R\, T \star [\mathsf{path}\, i\, T\,(\mathsf{Cut}\, q)]\}\ (\mathtt{find}\, i\, p)\ \{\lambda x.\ p \rightsquigarrow \mathsf{Mtreeof}\, R\, T \star [x = q]\}$$

Note that this specification allows for the tree T to have arbitrary many holes in it, as long as these holes are not on the path i followed by the function `find`. Also, observe how close this specification is from that of function `nth` provided at the end for §5.2.

It is also possible to combine the specification above with the tree surgery lemma. This yields the following specification.

$$\{p \leadsto \mathsf{Mtreeof}\, R\, T \,\star\, [\mathsf{path}\, i\, T\, T']\}\ (\mathtt{find}\, i\, p)$$
$$\{\lambda q.\ p \leadsto \mathsf{Mtreeof}\, R\, (\mathsf{subst}\, i\, (\mathsf{Cut}\, q)\, T) \,\star\, q \leadsto \mathsf{Mtreeof}\, R\, T'\}$$

If the intention is to fold back the unaltered subtree into the main tree, then the post-condition in the above specification may be simplified using the magic wand (recall §5.3). The alternative post-condition shown below describes the ownership of the subtree and provides an instance of the magic wand asserting that when this subtree is given back, then the original tree is recovered.

$$\lambda q.\, q \leadsto \mathsf{Mtreeof}\, R\, T' \star (q \leadsto \mathsf{Mtreeof}\, R\, T' \mathrel{-\!\!\star} p \leadsto \mathsf{Mtreeof}\, R\, T)$$

Although the magic wand can be handy for some specific applications, tree with holes and cut subtrees are much more general. In particular, they support reasoning about trees that contain several holes, and support merging back subtrees that may have been modified or permuted in arbitrary ways.

8. Arrays

8.1 Individual Cells

In Separation Logic, the ownership of a single cell of an array can be described by a predicate of the form $\mathsf{Cell}\, i\, v\, p$, where p is the location of the array, i is the index of the cell, and v is the value stored in this cell. With our arrow notation, this predicate can be written $p \leadsto \mathsf{Cell}\, i\, v$. One may also introduce a specific notation for arrays, e.g. $p[i] \leadsto v$, yet we will not need such a notation in this paper. Remark: the ownership of two distinct cells of an array takes the form $p \leadsto \mathsf{Cell}\, i\, v \star p \leadsto \mathsf{Cell}\, i'\, v'$. It may be surprizing at first that p appears to the left of two arrows separated by a star operator, but there is nothing wrong here.

Under a low-level view of memory such as in the C programming language, the predicate $p \leadsto \mathsf{Cell}\, i\, v$ can be *defined* as $(p + i) \hookrightarrow v$, by relying on pointer arithmetic in order to build on top of the heap predicate describing the ownership of a single memory cell. This presentation was followed in particular in the original paper on Separation Logic [14]. However, in higher-level programming languages that do not expose pointer arithmetic, we simply treat Cell as an abstract, primitive heap predicate.

We next define Cellof, the higher-order version of the representation predicate Cell, following the same pattern as for Refof (§6.2).

$$p \leadsto \mathsf{Cellof}\, R\, i\, V \;\equiv\; \exists v.\ p \leadsto \mathsf{Cell}\, i\, v \,\star\, v \leadsto R\, V$$

The specifications of read and write operations also closely resemble their counterparts on references. They are restricted to focused cells, represented using Id, as shown below.

$$\{p \leadsto \mathsf{Cellof}\, \mathsf{Id}\, i\, v\}\ (p[i])\ \{\lambda x.\ p \leadsto \mathsf{Cellof}\, \mathsf{Id}\, i\, v \,\star\, [x = v]\}$$
$$\{p \leadsto \mathsf{Cellof}\, \mathsf{Id}\, i\, v\}\ (p[i]\,\texttt{<-}\,w)\ \{\lambda\_.\ p \leadsto \mathsf{Cellof}\, \mathsf{Id}\, i\, w\}$$

8.2 Groups of Cells

We next define the predicate $p \leadsto \mathsf{Cellsof}\, R\, M$ to describe a subset of the cells from a same array—either a strict subset, or all the cells from the array. Here, M denotes a finite map whose domain corresponds to the indices of the cells considered, and whose values correspond to the logical descriptions of the values stored in these cells. The definition is as follows.

$$p \leadsto \mathsf{Cellsof}\, R\, M \;=\; \mathop{\circledast}_{i \in \mathsf{dom}\, M} p \leadsto \mathsf{Cellof}\, R\, i\, (M[i])$$

The equality state below allows may be exploited to focus at once on all cells covered by M, introducing a map T describing the concrete values stored in these cells.

$$p \leadsto \mathsf{Cellsof}\, R\, M \;=\; \exists T.\ [\mathsf{dom}\, M = \mathsf{dom}\, T] \,\star\, p \leadsto \mathsf{Cellsof}\, \mathsf{Id}\, T$$
$$\star\ \circledast_{i \in \mathsf{dom}\, M}\, (T[i]) \leadsto R\, (M[i])$$

For convenience, read and write operations can be specified directly on a group of focused cells. Below, we write $M[i := w]$ to denote the update of M with a binding from i to w.

$$i \in \mathsf{dom}\, M \;\Rightarrow$$
$$\{p \leadsto \mathsf{Cellsof}\, \mathsf{Id}\, M\}\ (p[i])\ \{\lambda x.\ p \leadsto \mathsf{Cellsof}\, \mathsf{Id}\, M \,\star\, [x = M[i]]\}$$
$$\{p \leadsto \mathsf{Cellsof}\, \mathsf{Id}\, M\}\ (p[i]\,\texttt{<-}\,w)\ \{\lambda\_.\ p \leadsto \mathsf{Cellsof}\, \mathsf{Id}\, (M[i := w])\}$$

For accessing a particular cell of the array, an alternative to focusing on all cells consists of first isolating the cell targeted, and then focusing only on this cell. The isolation operation is described below, where we write $M \backslash i$ for the map M with the key i removed.

$$\begin{array}{ccl} p \leadsto \mathsf{Cellsof}\, R\, M & = & p \leadsto \mathsf{Cellsof}\, R\, (M \backslash i) \\ \text{when } i \in \mathsf{dom}\, M & & \star\ p \leadsto \mathsf{Cellof}\, R\, i\, (M[i]) \end{array}$$

The following operation is also very useful for rearranging groups.

$$\begin{array}{ccl} p \leadsto \mathsf{Cellsof}\, R\, (M_1 \uplus M_2) & = & p \leadsto \mathsf{Cellsof}\, R\, M_1 \\ & & \star\ p \leadsto \mathsf{Cellsof}\, R\, M_2 \end{array}$$

For example, to specify a recursive quicksort function which takes as argument an array p, and two indices i and j in between which to sort the array, we would request in the precondition a predicate of the form $p \leadsto \mathsf{Cellsof}\, R\, M$ with the hypothesis that the domain of M matches the interval from i to j. For reasoning about the recursive calls after the pivot phase, we split the (updated) map describing the array into three parts, corresponding to the cells storing values that are smaller than, equal to, or greater than the pivot value. Thanks to the frame rule, we automatically obtain, for each recursive call, the fact that the cells outside the range of the recursive call are not modified. Once the two recursive calls are completed, we may then merge back the updated maps in order to obtain a post-condition mentioning a map whose domain is the same as in the pre-condition.

8.3 Length Predicate

So far, we have only described the cells from an array, but not the length of the array. In many languages, such as OCaml, the length of the array in stored in the header, and is accessible to the programmer using a function called `length`. To specify this function, we need a predicate to keep track of the length of the array. In other languages, such as C, there is no array header, yet we still need to keep track of its length in order to correctly specify the deallocation operation.

To specify the length, we introduce another primitive heap predicate, written $p \leadsto \mathsf{Arraysize}\, n$ (or, equivalently $\mathsf{Arraysize}\, n\, p$), to assert that an array of size n is allocated at location p. This predicate suffices, in particular, to specify the length function, as follows.

$$\{p \leadsto \mathsf{Arraysize}\, n\}\ (\mathtt{length}\, p)\ \{\lambda x.\, p \leadsto \mathsf{Arraysize}\, n \,\star\, [x = n]\}$$

We next introduce a predicate, written $p \leadsto \mathsf{Arrayof}\, R\, M$ to package the ownership of all the cells of the array together with its Arraysize predicate. The formal definition is as follows.

$$\begin{array}{ccl} p \leadsto \mathsf{Arrayof}\, R\, M & \equiv & \exists n.\ \ [\mathsf{dom}\, M = [0, n)] \\ & & \quad \star\ p \leadsto \mathsf{Arraysize}\, n \\ & & \quad \star\ p \leadsto \mathsf{Cellsof}\, R\, M \end{array}$$

It is convenient in practice to state derived specifications in terms of Arrayof for the length function, for read and write operations in focused arrays, as well as for focus operations. Due to lack of space, we only include present two of these specifications.

$$\{p \leadsto \mathsf{Arrayof}\, R\, M\}\ (\mathtt{length}\, p)$$
$$\{\lambda x.\, p \leadsto \mathsf{Arrayof}\, R\, M \star [x = |M|]\}$$
$$i \in \mathsf{dom}\, M \;\Rightarrow\; \{p \leadsto \mathsf{Arrayof}\, \mathsf{Id}\, M\}\ (p[i])$$
$$\{\lambda x.\ p \leadsto \mathsf{Arrayof}\, \mathsf{Id}\, M \star [x = M[i]]\}$$

We also assign concise specifications to the function `alloc`, which allocates an array of given size without initializing its cells, and to the function `free`, which deallocates an array. In the specifications shown below, $|M|$ denotes the number of bindings in M.

$$\{[\,]\}\ (\texttt{alloc}\,n)\ \{\lambda p.\ \exists M.\ p \rightsquigarrow \mathsf{Arrayof\,Id}\,M \ \star\ [|M| = n]\}$$
$$\{p \rightsquigarrow \mathsf{Arrayof\,Id}\,M\}\ (\texttt{free}\,p)\ \{\lambda\_.\,[\,]\}$$

Observe in particular how the pre-condition of `free` ensures two important properties: first, that the ownership of all the cells from the array is given back; and second, that the cells must be focused, so as to ensure the absence of memory leaks.

8.4 Example: Representation of Matrices

We consider matrices represented as arrays of arrays, first in the case where the rows consists of disjoint arrays, and second in the case where the rows might be shared. In both case, we assume the elements of the matrix to be represented by a predicate R of type $a \to A \to \mathsf{Hprop}$, for some a and A.

For the case of a matrix with non-aliased rows, we introduce the predicate $p \rightsquigarrow \mathsf{Matrixof}\,R\,M$, where M describes a mathematical matrix storing elements of type A. In the definition shown below, $\mathsf{dims}\,M$ returns the width and the height of the matrix, and K, of type $\mathsf{map\,int}\,(\mathsf{map\,int}\,A)$, denotes the logical value associated with the array of arrays.

$$\begin{aligned}
&p \rightsquigarrow \mathsf{Matrixof}\,R\,M \quad \equiv\\
&\quad \exists K.\ p \rightsquigarrow \mathsf{Arrayof}\,(\mathsf{Arrayof}\,R)\,K\\
&\quad \star \begin{bmatrix} \exists nm.\ (n,m) = \mathsf{dims}\,M \ \wedge\ \mathsf{dom}\,K = [0,n) \\ \wedge\ \forall i \in [0,n).\ \ \mathsf{dom}\,(K[i]) = [0,m) \\ \wedge\ \forall j \in [0,m).\ \ K[i][j] = M_{i,j} \end{bmatrix}
\end{aligned}$$

For the case of a matrix with possibly-aliased rows, we need to separately describe three parts. First, we describe the main array, which stores the locations of the arrays describing the rows. Assume this array is at location p, and represented with a map T of type $\mathsf{map\,int\,loc}$. The corresponding heap predicate is $p \rightsquigarrow \mathsf{Arrayof\,Id}\,T$. Second, we describe a group of arrays representing the possibly-shared rows. To that end, we make use of a map G that binds locations to maps describing the rows, each of type $\mathsf{map\,int}\,A$. In other words, G has type $\mathsf{map\,loc}\,(\mathsf{map\,int}\,A)$. The group of arrays is described by an iterated star over the bindings in G, more precisely: $\circledast_{(u,U)\in G}\ u \rightsquigarrow \mathsf{Arrayof}\,R\,U$. Third, we may establish a connection between the outer array T, the inner arrays described by G, and the matrix M being represented by the whole data structure, as follows.

$$\begin{aligned}
&p \rightsquigarrow \mathsf{AliasedRowsMatrixof}\,R\,M \quad \equiv\\
&\quad \exists TG.\ p \rightsquigarrow \mathsf{Arrayof\,Id}\,T \ \star\ \circledast_{(u,U)\in G}\ u \rightsquigarrow \mathsf{Arrayof}\,R\,U\\
&\quad \star \begin{bmatrix} \exists nm.\ (n,m) = \mathsf{dims}\,M \ \wedge\ \mathsf{dom}\,T = [0,n) \\ \wedge\ \forall i \in [0,n).\ T[i] \in \mathsf{dom}\,G \\ \wedge\ \forall j \in [0,m).\ G[T[i]][j] = M_{i,j} \end{bmatrix}
\end{aligned}$$

Interestingly, thanks to the rule that allows to focus at once on all cells from an array, we can convert, at the logical level, from a matrix made of disjoint rows (Matrixof) to a matrix made a possibly-aliased rows (AliasedRowsMatrixof). Even more interestingly, if we are able to prove that a matrix with possibly-aliased rows is actually represented by an outer array that contains pointers that are all distinct from each other, then the reciprocal conversion (from AliasedRowsMatrixof to Matrixof) is possible.

In summary, the representation predicate $\mathsf{Arrayof}\,(\mathsf{Arrayof}\,R)$ describes an outer array that owns the inner arrays, whereas the predicate $\mathsf{Arrayof\,Id}$ describes an outer array that does not own the inner arrays, the latter being described using a group of arrays each described using the predicate $\mathsf{Arrayof}\,R$.

9. Conclusion

Separation Logic is a powerful approach to the modular specification and verification of mutable data structures. Higher-order representation predicates further increase modularity by offering the possibility to control, a posteriori, whether a container should own or not the items that it stores. This possibility has been identified by previous work [2, 12, 15]. In this paper, we investigate this idea further, considering a number of practical applications. In particular, we present a uniform pattern for reasoning about an access operation in a structure that owns its elements: first focus on the piece of data considered in order to detach the ownership of this item from the ownership of the structure, then reason about the access operation, and finally unfocus in order to fold back to the original form. One remaining challenge is to investigate to what extent the focus and unfocus operations can be automated in proof scripts.

Acknowledgements

I wish to thank François Pottier for fruitful discussions. This research was partly supported by the French national research organization ANR (grant ANR-15-CE25-0008).

References

[1] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Theory and practice of chunked sequences. In Andreas S. Schulz and Dorothea Wagner, editors, *Algorithms - 22th Annual European Symposium (ESA)*, volume 8737 of *LNCS*, pages 25–36. Springer, 2014.

[2] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Transactions on Programming Languages and Systems*, 29(5):24–es, aug 2007.

[3] Arthur Charguéraud. *Characteristic Formulae for Mechanized Program Verification*. PhD thesis, Université Paris-Diderot, 2010.

[4] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *ICFP*, pages 418–430. ACM, 2011.

[5] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs, 2015. Submitted for publication. `http://arthur.chargueraud.org/research/2013/cf/cf.pdf`.

[6] The Coq Development Team. The Coq proof assistant reference manual, version 8.1. At `http://coq.inria.fr/`, 2007.

[7] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *JFP*, 16(2):197–218, 2006.

[8] Haim Kaplan and Robert E Tarjan. Persistent lists with catenation via recursive slow-down. In *TOC'95*, pages 93–102. ACM, 1995.

[9] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system*, October 2005.

[10] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. In *ICFP*, pages 62–73, 2006.

[11] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, pages 1–19. Springer, 2001.

[12] Peter O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *POPL*, pages 268–280, January 2004.

[13] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[14] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[15] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. Verifying generics and delegates. In *ECOOP 2010 – Object-Oriented Programming*, pages 175–199. Springer, 2010.

[16] Carsten Varming and Lars Birkedal. Higher-order separation logic in isabelle/HOLCF. *Electronic Notes in Theoretical Computer Science*, 218:371–389, oct 2008.