Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 108 (2004) 99–112
www.elsevier.com/locate/entcs

Formally Specifying Dynamic Data Structures for Embedded Software Design: an Initial Approach

Edgar G. Daylight[a,b,1] Bart Demoen[b,2] Francky Catthoor[a,c,1]

[a] *DESICS Division, IMEC vzw, Kapeldreef 75, B-3001 Heverlee, Belgium*

[b] *Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium*

[c] *Department of Electrical Engineering, Katholieke Universiteit Leuven, Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium*

Abstract

In the embedded, multimedia community, designers deal with data management at different levels of abstraction ranging from abstract data types and dynamic memory management to physical data organisations. In order to achieve large reductions in energy consumption, memory footprint, and/or execution time, data structure related optimizations are a must. However, the complexity of describing and implementing such optimized implementations is immense. Hence, a strong, practical need is present to unambiguously (i.e. mathematically) describe these complicated dynamic data organisations.
The objective of this article is to formally describe data structures and access operations -or dynamic data structures for short- that we have implemented in prior, application-related work. We do this by (a) extending the syntax and semantics of Separation Logic -a logic developed recently in the program verification community- and (b) using it as a specification language for our applications. The short-term benefit of this work is that it allows the embedded software designer to unambiguously express and hence more easily explore low cost, dynamic data structures. In practice this means that the designer can clearly reason and consequently implement nontrivial but optimal dynamic data structures. The benefit in the long term is that it provides an avenue for future optimizing compilers to increase the global scope of optimizations that are related to dynamic data management.

Keywords: Embedded system design, Separation logic, Dynamic data structures

[1] Email: `{voudheus,catthoor}@imec.be`
[2] Email: `{bmd}@cs.kuleuven.ac.be`

1 Introduction

Designing and implementing low cost data structure implementations on an embedded platform is a tedious (i.e. time consuming and error-prone) task. But in the context of embedded, multimedia applications, this task is of extreme importance. Due to the data dominance in this application domain, very efficient code can be produced if the data structures of the application are taken into account early during the design trajectory. This implies that not only verification of the implementation (i.e. the output of the design trajectory) is required. Also the ability to systematically explore various data organisations [3,14] (while traversing the design trajectory) is to the benefit of the designer. It is this latter issue that concerns us.

1.1 Modelling Low Cost, Dynamic Data Structures

We motivate our formal work by presenting one example of a data structure transformation. In Figure 1(a), the initial data structure 1 is transformed into data structure 2 by adding links to data structure 1. These links allow easy traversal through data structure 2 which usually results in a decrease of the average amount of data accesses. The memory footprint on the other hand has increased due to the additional links. A clear trade-off is present between data accesses and memory footprint. This trade-off can span a large range when the size of the data set grows.

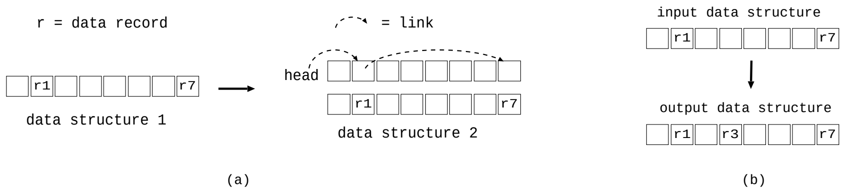

Fig. 1. (a) Adding an array of links on top of an array of records. (b) An example of dynamics: inserting a record in data structure 1.

The traversal operation, used in the above example, is an access operation in which all stored records (r) in the data structure need to be consulted. Due to the dynamic behaviour of the multimedia application, it is not a priori known which records are stored in the data structure. Traversal is a very dominant access operation in multimedia applications as is demonstrated in various case studies in [4]. This implies that memory-related energy consumption can decrease drastically for data structure 2 (in comparison to data structure 1) due to the decrease in number of data accesses, even though the memory footprint has increased.

In our previous work [4] we have explored various data structures and corresponding transformations. We have only presented one transformation here but in practice, we apply various transformations in sequence and hence obtain low cost but difficult-to-understand data structures and corresponding access operations. The problem we faced during our explorations was the inability to exactly state what data structure 2 (or data structure 1) means without having to implement the data structure in code. Two factors contribute to this problem. (i) The process of implementing complicated, dynamic data structures is a time-consuming and error-prone process. (ii) Even if the implementation phase is successfully completed, other designers and even the same designer have difficulty extracting and understanding from the code the data structure and its access operations.

The exact problem we address in this paper is the development of a formal model that allows a designer to compactly and unambiguously express the exact behaviour of nontrivial but low cost dynamic data structures. Our contribution is the extension of Separation Logic [9,10,11] in terms of syntax and semantics and using it as a specification language for dynamic data structures.

In the rest of this section we demonstrate our two main contributions in terms of formalisation: (i) sparseness and (ii) access operations. We do so by using data structure 1 as an example. We use the term data structure and heap interchangeably. For instance, data structure 1 is a synonym for heap 1.

Sparseness

To describe the spatial orientation of data structure 1, we can use the $\star$ connective of Separation Logic in the following specification.

$$\exists l.\, \exists r_1.\, \exists r_7.\ (l \mapsto \_) \star (l+1 \mapsto r_1) \star (l+2 \mapsto \_) \star \cdots \star (l+6 \mapsto \_) \star (l+7 \mapsto r_7)$$

The specification states that the data structure contains eight consecutive heap cells starting at location l. The sparseness of the data structure is two (out of a total of eight elements). However, the specification is too strict for it specifies exactly which two heap cells contain a record (i.e. r_1 and r_7) as opposed to expressing that *any* two of the eight heap cells can contain a record. It is exactly this characteristic which needs to be modelled because it is the main source of complexity in dynamic data structures for embedded, multimedia applications. We will later show how to deal with sparseness formally.

Access Operations

Another, more fundamental problem is the inability to model change (of a data structure) in Separation Logic. Consider for instance the insertion of a record r_3 in data structure 1 as is shown in Figure 1(b). To specify heap changes (or data structure changes) such as these, we need to extend the original syntax and semantics to incorporate notions of input heap and output heap. The input heap corresponds to the original data structure (i.e. before the insertion has taken place). The output heap corresponds to the data structure that contains the record r_3. Insertion is only one of the access operations that we define formally in this paper.

1.2 Overview

We present related work in Section 2 and re-introduce all relevant, initial work on Separation Logic in Section 3. We extend Separation Logic in two different ways in Section 4 in order to handle sparseness and access operations. In this section we model data structure 1 of Figure 1(a). Due to lack of space, we refer to a technical report [5] in which we present all the syntax and semantics of our specification language and completely model data structure 2 of Figure 1(a). We state our conclusions in Section 5.

2 Related Work

Besides the references given in the introduction, other sources of inspiration from the verification literature are [15,1]. Yang [15] discusses soundness and completeness of Separation Logic. Ahmed et al. [1] use type systems in conjunction with a variant of Separation Logic in order to describe hierarchical memory layouts.

The difference between the program verification community and our (embedded systems') community is that we do not develop a logic intended for verification of an imperative program. We use and extend the syntax and semantics of the original logic in order to model data structures and access operations. The goal is to aid the embedded systems' designer in expressing complex but low cost, dynamic, data organisations. Soundness, completeness and other logical properties are of no concern for our (current) objective.

When designing and implementing an embedded (e.g. hand-held) device, reduction of energy consumption for a given task is the main objective [2,3,6]. In the subdomain of multimedia, embedded systems, data management is a main contributor to power consumption [3,6,12,13]. Polyhedral models [8,3] are often used in this community (e.g. in optimization tools) to mathematically model data storage and data accesses.

Our specification language differs in two ways from currently used formal approaches for data organisations in the embedded systems' community. First, we describe the correlations between subparts of a compound data structure as opposed to only modelling the compound data structure as a set of uncorrelated arrays. For example, the two arrays that together form data structure 2 of Figure 1(a) are correlated in a specific way. We model this explicitly (see [5]). Second, we distinguish between the different kinds of data accesses that are applied on the data structure under investigation. For example, we distinguish between data accesses that correspond to the insertion of a record and data accesses that correspond to the removal of a record as opposed to simply modelling both as physical accesses without further distinction. These two additional sources of information are explicitly taken into account in our formalism. If exploited correctly, this valuable information allows significant reductions in energy consumption, memory footprint, and/or execution speed (as is demonstrated in our previous work [4]).

3 Separation Logic

All fundamental concepts in this section are originally from O'Hearn, Reynolds, and co. [7,11,15]. We re-introduce the concepts for didactic purposes only. Readers who are unfamiliar with Separation Logic are recommended to read [11].

3.1 Stack vs. Heap

To describe a data organisation, we use a stack and a heap. An element $s \in S$ is a stack and $dom(s)$ denotes the domain of the stack s. An element $h \in H$ is a heap and $dom(h)$ denotes the domain of the heap h. A stack maps variables onto values. A heap maps locations onto values. Values are either integers, atoms, or locations.

$$(1) \qquad Val = Int \cup Atoms \cup Loc \qquad S = Var \rightharpoonup_{fin} Val \qquad H = Loc \rightharpoonup_{fin} Val$$

Loc= $\{l, \ldots\}$ is a set of locations and $\forall l \in Loc.\ l + 1 \in Loc$ and $(l + 1) - 1 = l$. Var $= \{x, y, \ldots\}$ is a set of variables. Atoms $= \{nil, a, \ldots\}$ is the set of atoms. We use the notation $\rightharpoonup_{fin}$ for finite partial functions.

3.2 Syntax

Expressions E are presented in Table 1 where E_1 and E_2 are either both integers or locations. In the latter case, addition and subtraction on locations still needs to be defined.

Separation Logic is an extension of classical (predicate) logic. The empty heap, spatial conjunction, and spatial implication[3] constitute this extension. The nonatomic formulae β are presented in Table 1 where P and Q are nonatomic formulae. The atomic formulae α are presented in Table 1 where

[3] We do not use spatial implication in this paper.

Table 1
Syntax

E	$::=$	x	variable	β	$::=$	α	Atomic Formulae
	$\mid$	42	integer		$\mid$	$false$	Falsity
	$\mid$	nil	nil		$\mid$	$P \Rightarrow Q$	Classical Implication
	$\mid$	a	atom		$\mid$	emp	Empty Heap
	$\mid$	l	location		$\mid$	$P \star Q$	Spatial Conjunction
	$\mid$	$E_1 + E_2$	addition		$\mid$	$P \mathrel{-\!\!\star} Q$	Spatial Implication
	$\mid$	$E_1 - E_2$	subtraction		$\mid$	$\exists x.P$	Existential Quantification
	$\mid$	$\ldots$					
α	$::=$	$E_1 = E_2$	Equality	$\neg P$	$=$		$P \Rightarrow false$
	$\mid$	$E_1 < E_2$	Smaller than	$true$	$=$		$\neg(false)$
	$\mid$	$E_1 \leq E_2$	Smaller than or equal to	$P \vee Q$	$=$		$(\neg P) \Rightarrow Q$
	$\mid$	$E_1 \mapsto E_2$	Points to	$P \wedge Q$	$=$		$\neg(\neg P \vee \neg Q)$
	$\mid$	$\ldots$		$\forall x.P$	$=$		$\neg\exists x.\neg P$

E_1 and E_2 are expressions. The other connectives (see Table 1) are defined in terms of those presented previously. We define the set $free(P)$ of free variables of a formula as usual.

3.3 Semantics

The relation of the form $s, h \models P$ asserts that P is true of stack $s \in S$ and heap $h \in H$. It is required that $free(P) \subseteq dom(S)$. We use the notation $h \perp h'$ to denote that heap h and heap h' are disjoint: $dom(h) \cap dom(h') = \varnothing$. Also, $h \centerdot h'$ denotes the union of disjoint heaps (i.e. the union of functions with disjoint domains).

An expression E is interpreted as a heap-independent value $[[E]]\,s \in Val$ where the $dom(s)$ includes the free variables of E. Examples are $[[x]]\,s = s\ x$ where $x \in Var$ and $[[3]]\,s = 3$ and $[[l]]\,s = l$ where $l \in Loc$. Also, $[[E_1 + E_2]]\,s = [[E_1]]\,s + [[E_2]]\,s$ and $[[E_1 - E_2]]\,s = [[E_1]]\,s - [[E_2]]\,s$. The semantic clauses are presented below. (See [7] for the motivation of these definitions.)

$s, h \models E_1 = E_2$	iff	$[[E_1]]\,s = [[E_2]]\,s$	$s, h \models P \mathrel{-\!\!\star} Q$	iff	$\forall h'$. if $h' \perp h$
$s, h \models E_1 \mapsto E_2$	iff	$\{[[E_1]]\,s\} = dom(h)$ and			and $s, h' \models P$
		$h([[E_1]]\,s) = [[E_2]]\,s$			then $s, h \centerdot h' \models Q$
$s, h \models emp$	iff	$h = \varnothing$ (h is the empty heap)	$s, h \models false$		never
$s, h \models P \star Q$	iff	$\exists h_0, h_1.\ \ h_0 \perp h_1,\ \ h_0 \centerdot h_1 = h,$	$s, h \models P \Rightarrow Q$	iff	if $s, h \models P$ then $s, h \models Q$
		$s, h_0 \models P$ and $s, h_1 \models Q$	$s, h \models \exists x.P$	iff	$\exists v \in Val.\ \ [s \mid x \mapsto v], h \models P$

The first clause implicitly assumes that equality is defined for values since $[[E]]\,s \in Val$ with E being an expression. This amounts to assuming that equality is defined for atoms and locations. Similar assumptions hold for the additional clause: $s, h \models E_1 < E_2$ iff $[[E_1]]\,s < [[E_2]]\,s$. Also, we interpret $E_1 \leq E_2$ as the abbreviation for $(E_1 < E_2) \vee (E_1 = E_2)$ and $E_1 \leq E_2 < E_3$ as the abbreviation for $(E_1 \leq E_2) \wedge (E_2 < E_3)$, etc. We also use the underscore in a points-to relation to mean the following:

$$e \mapsto \_ \quad \triangleq \quad \exists x.\, e \mapsto x \text{ where } x \text{ is not free in } e.$$

4 Extending the Specification Language

In each of the following two sections, we start off with the original syntax of Section 3.2 and the original semantics of Section 3.3 and extend both syntax and semantics in order to model realistic problems that we have encountered in our applications (see [4]).

In Section 4.1 we extend the logic in order to model the spatial (e.g. sparse) characteristics of data structure 1 of Figure 1(a). In Section 4.2 we extend the logic in order to model the insertion and removal access operations of data structure 1. Based on the extensions presented in this paper, we completely specify data structure 2 in [5].

4.1 Array: Spatial Orientation

In order to mathematically describe an array, we (re)use the Iterated Separating Conjunction operator and define a similar operator in order to model sparseness. We explain how to subtract one heap from another and distinguish between active and passive records. Finally we obtain an exact mathematical description of data structure 1 of Figure 1(a) as intended.

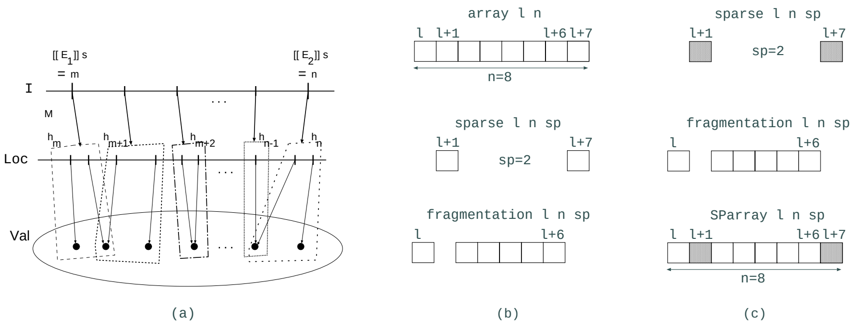

Fig. 2. (a) Mathematical objects that are used to define ISC. (b) A heap of length 8 and two disjoint subheaps. (c) Composing the sparse array from two disjoint subheaps.

4.1.1 Iterated Separating Conjunction

The $\bigodot_{v=E_1}^{E_2}$ operator [11] is called Iterated Separating Conjunction (ISC) and is defined below. In Figure 2(a) we graphically represent the mathematical objects that are used in the definition. The integers m and n are defined in terms of E_1 and E_2 respectively. The interval I is represented as a line segment starting at m and ending at n. The function M maps each integer on the line segment onto a heap $h \in H$. Each heap is a function from locations to values. The figure also shows that each heap is disjoint from all other heaps. This means that the locations that belong to the domain of a specific heap do not belong to the domain of any other heap.

$$\begin{array}{ll}
 & s, h \models \bigodot_{v=E_1}^{E_2} P \\
 & \quad\text{iff} \\
 & v \notin \mathit{dom}\,(s) \text{ and} \\
\text{let} & m = [[E_1]]\, s \in \mathit{Int}, \\
 & n = [[E_2]]\, s \in \mathit{Int}, \\
 & I = \{i \mid m \leq i \leq n\}
\end{array}
\qquad
\begin{array}{ll}
 & \quad <\!\text{continued}\!> \\
\text{in} & \exists M \in I \to H. \\
 & \quad \forall i, j \in I.\, i \neq j \text{ implies} \\
 & \quad Mi \perp Mj \text{ and} \\
 & \quad h = \bigcup \{Mi \mid i \in I\} \text{ and} \\
 & \quad \forall i \in I.\ [s \mid v \mapsto i]\,, Mi \models P
\end{array}$$

This allows us to define an array.

$$(2) \qquad \mathit{array}\, l\, n \quad \triangleq \quad \bigodot_{k=0}^{n-1} (l + k \mapsto \_)$$

This is equivalent to: $(l \mapsto \_) \star (l + 1 \mapsto \_) \star \ldots \star (l + n - 1 \mapsto \_)$. This array is depicted at the top in Figure 2(b) for $n = 8$.

Note that the right hand side of (2) is a formula. Whenever the notation $\mathit{array}\ l\ n$ is used in a specification, it should be substituted with the formula $\bigodot_{k=0}^{n-1} (l + k \mapsto \_)$. E.g.: $\exists l.\, \exists n.\ (\mathit{array}\, l\, n)$ is syntactically equivalent to $\exists l.\, \exists n.\ \bigodot_{k=0}^{n-1} (l + k \mapsto \_)$. For convenience we sometimes refer to (2) as Formula (2).

We can state properties of the array. For instance, the lookup property:

$$(3) \qquad \mathit{lookup}\, l\, n\, \mathit{index}\, \mathit{val} \quad \triangleq \quad (0 \leq \mathit{index} < n) \wedge (l + \mathit{index} \mapsto \mathit{val})$$

expresses that (i) the value index lies in the range defined by 0 and n and (ii) the current heap contains the value val at location $l + \mathit{index}$. To state that the data organisation under investigation is an array with the *lookup* property we assert that: $\exists l. \exists n. \forall \mathit{index}. \exists \mathit{val}.\ (\mathit{array}\, l\, n) \wedge (\mathit{lookup}\, l\, n\, \mathit{index}\, \mathit{val})$. This formula will only evaluate to true, for a given stack s and heap h, if there exists an l and an n such that for all index values that lie in between the range defined by 0 and n, there exists a val such that both $\bigodot_{k=0}^{n-1} (l + k \mapsto \_)$ and $(l + \mathit{index} \mapsto \mathit{val})$ hold.

Even though Formula (3) follows naturally from Formula (2), we still state Formula (3) explicitly. Recall that our objective is to describe specific properties of a data structure regardless of whether these properties can be derived from other logical formulae (cf. Section 2 in which we relate our work to the program verification community).

Another property of the array is presented in Formula (4) which is an inductive definition[4]. It describes how to traverse an array. The boundary of the traversal is defined by l and n, the current heap cell (of the traversal) that is being consulted is characterized by i, function f is applied to the value v that is stored in the current heap cell, and res is the result which represents the sum of all the function applications during the traversal.

$$(4)\qquad \mathit{traverse}\ l\ n\ i\ f\ \mathit{res} \ \triangleq\ (\mathit{emp} \wedge (\mathit{res} = 0))\ \vee$$
$$((0 \le i < n) \wedge (\exists v.\ (l + i \mapsto v) \star (\mathit{traverse}\ l\ n\ (i+1)\ f\ (\mathit{res} - (f\ v)))))$$

For instance, $\exists l.\exists n.\exists f.\exists res.\ (array\ l\ n) \wedge (traverse\ l\ n\ 0\ f\ res)$ evaluates to true if the data organisation under investigation, defined by the given stack s and heap h, has the structure described by Formula (2) and Formula (4) for a given l, n, function f, and result res.

Formulae (3) and (4) describe access operations but, since they do not change the heap, we have been able to specify them in this section. In general, access operations change the heap. This issue is dealt with in Section 4.2.

In the rest of this article, we will state formulae -such as Formulae (2),(3), and (4)- in isolation.

4.1.2 Sparse Iterated Separating Conjunction

In order to specify sparseness, we introduce the $Sparse_{sp} \bigodot_{v=E_1}^{E_2}$ operator which we call Sparse Iterated Separating Conjunction (Sparse ISC).

$$\begin{array}{ll}
s, h \models Sparse_{sp} \bigodot_{v=E_1}^{E_2} P & \text{in}\quad 0 < sp < n - m + 1, \\
\qquad\qquad \text{iff} & \qquad \exists J \subseteq I \text{ and } |J| = sp, \\
\quad v \notin dom\,(s) \text{ and} & \qquad \exists M \in J \to H. \\
\text{let}\quad m = [[E_1]]\, s \in Int, & \qquad\quad \forall i, j \in J.\, i \neq j \text{ implies} \\
\qquad\ n = [[E_2]]\, s \in Int, & \qquad\quad Mi \perp Mj \text{ and} \\
\qquad\ I = \{i \mid m \le i \le n\} & \qquad\quad h = \bigcup \{Mj \mid j \in J\} \text{ and} \\
\qquad\ \text{<continued>} & \qquad\quad \forall j \in J.\ [s \mid v \mapsto j]\,, Mj \models P
\end{array}$$

A specific example of sparseness is presented in the middle of Figure 2(b) in which $sp = 2$. The mathematical formulation is:

$$sparse\ l\ n\ sp \ \triangleq\ Sparse_{sp} \bigodot_{k=0}^{n-1} (l + k \mapsto \_)$$

which describes a subheap of the array. The subheap consists of sp heap cells.

[4] We do not analyze the *termination* property of this inductive definition. Neither do we do so for other definitions in this paper.

4.1.3 Spatial Subtraction

Subtracting the sparse array from the array results in a fragmented array (see bottom data structure in Figure 2(b)). We express this as follows.

$$\mathit{fragmentation}\; l\; n\; sp \triangleq (\mathit{array}\; l\; n) \ominus (\mathit{sparse}\; l\; n\; sp)$$

The $\ominus$ connective, which we call spatial subtraction, is defined as follows.

$$s, h \models P \ominus Q \quad \text{iff} \quad \exists h'.\exists h''. \quad h = h'' - h' \quad \text{and} \quad s, h'' \models P \quad \text{and} \quad s, h' \models Q$$

It is possible to define $\ominus$ in terms of the original syntax (of Section 3.2) but we omit this for brevity.

4.1.4 Active vs. Passive Records

The heap described by *sparse l n sp* is not equivalent to data structure 1 in Figure 1(a) because this heap only contains *sp* heap cells while data structure 1 contains all n heap cells. In the case of data structure 1, a total of *sp* heap cells contain an *active* record (i.e. r1 and r7) and all other heap cells contain passive records. In order to specify data structure 1, we introduce a new notation: $\mathit{Active}\,(E_1 \mapsto E_2)$. This notation asserts that the corresponding heap cell of E_2 is an active record. We apply the following changes to our initial semantics.

$$H = \mathit{Loc} \rightharpoonup_{fin} \mathit{Val} \times \{\mathit{Active}, \mathit{Passive}\}$$

$$\begin{array}{lll} s, h \models E_1 \mapsto E_2 & \text{iff} & \{[[E_1]]\, s\} = \mathit{dom}(h) \quad \text{and} \quad h([[E_1]]\, s) = \langle [[E_2]]\, s, \_ \rangle \\ s, h \models \mathit{Active}\,(E_1 \mapsto E_2) & \text{iff} & \{[[E_1]]\, s\} = \mathit{dom}(h) \quad \text{and} \quad h([[E_1]]\, s) = \langle [[E_2]]\, s, \mathit{Active} \rangle \end{array}$$

Using *Active* allows us to define exactly a sparse array: $SParray\; l\; n\; sp$. This is data structure 1 of Figure 1(a).

$$\begin{array}{lll} \mathit{array}\, l\, n & \triangleq & \bigodot_{k=0}^{n-1} (l + k \mapsto \_) \\ \mathit{sparse}\, l\, n\, sp & \triangleq & \mathit{Sparse}_{sp} \bigodot_{k=0}^{n-1} \mathit{Active}\,(l + k \mapsto \_) \\ \mathit{fragmentation}\, l\, n\, sp & \triangleq & (\mathit{array}\, l\, n) \ominus (\mathit{sparse}\, l\, n\, sp) \\ SParray\, l\, n\, sp & \triangleq & (\mathit{sparse}\, l\, n\, sp) \star (\mathit{fragmentation}\, l\, n\, sp) \end{array}$$

See Figure 2(c) for the corresponding graphical representations.

4.2 Array: Access Operations

We extend the original syntax (Section 3.2) and semantics (Section 3.3) to model change. We do this by changing the relation $s, h \models P$ to $s, h_i, h_o \models P$. We use h_i to denote the input heap (i.e. the heap before the change has occurred) and h_o to denote the output heap (i.e. the heap after the change). We do not split the stack s into an input stack s_i and an output stack s_o.

We present extended syntax, additional notation, and corresponding semantics. Based on these extensions, we specify various access operations of data structure 1.

4.2.1 Syntax

The additional, nonatomic formulae β are:

$$\begin{array}{rll} \beta ::= & emp_i & \text{Empty Input Heap} \\ \mid & emp_o & \text{Empty Output Heap} \\ \mid & Same & \text{IIOH} \\ \mid & Same\,(R) & \text{IIOH and both model } R \\ \mid & P;Q & \text{Sequential Composition} \end{array}$$

where P and Q are nonatomic formulae and R is a nonatomic formula that describes only one heap. In other words, R is a nonatomic formula of Section 3.2. IIOH is an abbreviation for Identical Input and Output Heaps.

The additional atomic formulae α are:

$$\begin{array}{rll} \alpha ::= & E_1 \mapsto_i E_2 & \text{Points to Relation in Input Heap} \\ \mid & E_1 \mapsto_o E_2 & \text{Points to Relation in Output Heap} \end{array}$$

Instead of having emp to denote that the (one and only) heap h is empty, we use emp_i to denote that input heap h_i is empty and emp_o to denote that output heap h_o is empty. We use $Same$ to describe that h_i and h_o are identical. Similarly, $Same\,(R)$ is used when both h_i and h_o adhere to the description R. For instance, $s, h_i, h_o \models Same\,((5 \mapsto 3) \star true)$ is semantically equivalent to $s, h_i, h_o \models ((5 \mapsto_i 3) \wedge (5 \mapsto_o 3)) \star Same$. Note that we use two different points-to relations: $\mapsto_i$ for the input heap h_i and $\mapsto_o$ for the output heap h_o.

The sequential composition $P;Q$ denotes a heap change that is composed of two consecutive heap changes; i.e. heap change P followed by heap change Q.

4.2.2 Notation

Since we are dealing with an input heap h_i and an output heap h_o in order to model heap changes, we extend the concept of disjointness of heaps (cf. Section 3.3) to disjointness of heap changes. Similarly, we extend the concept of the union of disjoint heaps to the union of disjoint heap changes and the same holds for set inclusion. Finally, we define the projections of a couple of heaps to one heap.

We use $(h_i, h_o) \perp \left(h_i^{'}, h_o^{'}\right)$ to denote that $h_i \perp h_i^{'}$ and $h_o \perp h_o^{'}$. Similarly, $(h_i, h_o) \centerdot \left(h_i^{'}, h_o^{'}\right)$ denotes $\left(h_i \centerdot h_i^{'},\ h_o \centerdot h_o^{'}\right)$. Similarly, $(h_i, h_o) \sqsubseteq \left(h_i^{'}, h_o^{'}\right)$ denotes $h_i \sqsubseteq h_i^{'}$ and $h_o \sqsubseteq h_o^{'}$. Also, $\Pi_1\,(h_1, h_2) = h_1$ and $\Pi_2\,(h_1, h_2) = h_2$.

4.2.3 Semantics

The semantics of assertions are given by:

$$s, h_i, h_o \models P \text{ with } free(P) \subseteq dom(s)$$

thb

Table 2
Semantics

$s, h_i, h_o \models E_1 = E_2$	iff	$[[E_1]]\, s = [[E_2]]\, s$	$s, h_i, h_o \models P \twoheadrightarrow\!\!\star Q$	iff	$\forall h_i', h_o'.$ if $\left(h_i', h_o'\right) \perp (h_i, h_o)$
$s, h_i, h_o \models E_1 < E_2$	iff	$[[E_1]]\, s < [[E_2]]\, s$			and $s, h_i', h_o' \models P$ then
$s, h_i, h_o \models E_1 \mapsto_i E_2$	iff	$\{[[E_1]]\, s\} = dom(h_i)$			$s, \left(h_i \cdot h_i'\right), \left(h_o \cdot h_o'\right) \models Q$
	and	$h_i([[E_1]]\, s) = \langle [[E_2]]\, s \rangle$	$s, h_i, h_o \models false$	iff	never
$s, h_i, h_o \models E_1 \mapsto_o E_2$	iff	$\{[[E_1]]\, s\} = dom(h_o)$	$s, h_i, h_o \models P \Rightarrow Q$	iff	if $s, h_i, h_o \models P$
	and	$h_o([[E_1]]\, s) = \langle [[E_2]]\, s \rangle$			then $s, h_i, h_o \models Q$
$s, h_i, h_o \models emp_i$	iff	$h_i = \varnothing$	$s, h_i, h_o \models \exists x.P$	iff	$\exists v \in Val.\ [s \mid x \mapsto v], h_i, h_o \models P$
$s, h_i, h_o \models emp_o$	iff	$h_o = \varnothing$	$s, h_i, h_o \models P; Q$	iff	$\exists h_{tmp}.\ s, h_i, h_{tmp} \models P$
$s, h_i, h_o \models P \star Q$	iff	$\exists h_{i,1}, h_{o,1}, h_{i,2}, h_{o,2}.$			and $s, h_{tmp}, h_o \models Q$
$(h_i, h_o) = \left(h_{i,1}, h_{o,1}\right) \cdot \left(h_{i,2}, h_{o,2}\right)$,			$s, h_i, h_o \models Same$	iff	$h_i = h_o$
$s, h_{i,1}, h_{o,1} \models P$	and	$s, h_{i,2}, h_{o,2} \models Q$	$s, h_i, h_o \models Same\,(R)$	iff	$s, h_i \models R$ and $s, h_o \models R$
					and $s, h_i, h_o \models Same$

Table 3
The array and its access operations.

$array\ l\ n$	$\triangleq$	$Same\left(\bigodot_{k=0}^{n-1} (l + k \mapsto \_)\right)$	$remove\ l\ n\ index\ v$	$\triangleq$	$(0 \le index < n)\ \wedge$
$insert\ l\ n\ index\ v$	$\triangleq$	$(0 \le index < n)\ \wedge$	$((l + index \mapsto_i v) \wedge \neg (l + index \mapsto_o v)) \star Same$		
$(\exists w.\ (l + index \mapsto_i w) \wedge (l + index \mapsto_o v)) \star Same$			$modify\ l\ n\ index\ f$	$\triangleq$	$\exists v.$
			$(remove\ l\ n\ index\ v)\,; (insert\ l\ n\ index\ (f\ v))$		

The basic domains of Section 3.1 remain unchanged. The semantic clauses of the original and extended syntax are defined in Table 2. Note that in the last semantic clause we use the relation $s, h \models R$ of Section 3.3. In addition to the semantic clauses of Table 2, we add the semantical interpretation of the Iterated Separating Conjunction but we omit the (almost identical) Sparse ISC for brevity:

$$s, h_i, h_o \models \bigodot_{v=E_1}^{E_2} P$$

iff

$v \notin dom\,(s)$ and

let $m = [[E_1]]\, s \in Int,$
$n = [[E_2]]\, s \in Int,$
$J = \{j \mid m \le j \le n\}$

<continued>

in $\exists M \in J \rightarrow H \times H.$
$\forall k, l \in J.\ k \neq l$ implies
$Mk \perp Ml$ and
$h_i = \bigcup \{\Pi_1 (Mk) \mid k \in J\}$ and
$h_o = \bigcup \{\Pi_2 (Mk) \mid k \in J\}$ and
$\forall k \in J.\ [s \mid v \mapsto k], \Pi_1 (Mk), \Pi_2 (Mk) \models P$

4.2.4 Specifying Access Operations

In Table 3 we specify an array, insertion, removal, and modification of a record in an array. As a simple application of sequential composition, we specify the modification of an array element as the sequential composition of the removal of a value v followed by the insertion of the modified value $(f\ v)$.

Note that for the insertion of a record in the array, we specify that value w is overwritten by value v regardless of whether values w and v are equal.

5 Conclusions

In this article we have addressed the need of formally describing dynamic data structures as they occur in real-life, multimedia applications. Two relevant characteristics that we have modelled are (i) sparseness and (ii) access operations. While doing so, we have extended the original syntax and semantics of Separation Logic -a logic used in the program verification community- in various ways of which the introduction of input heap vs. output heap is the most profound. As an application of our syntactical and semantical extensions, we have specified data structure 1 of Figure 1(a) in this article and data structure 2 in [5], both in terms of spatial orientation and in terms of access operations.

Acknowledgement

The first author thanks P. O'Hearn and C. Calcagno for helping him specify access patterns.

References

[1] A. Ahmed, L. Jia, D. Walker, "Reasoning about Hierarchical Storage" *18th Annual IEEE Symposium on Logic in Computer Science,* June 22 - 25, 2003 Ottawa, Canada.

[2] L. Benini, G. DeMicheli, *"Dynamic Power Management Design Techniques and CAD Tools"*, 1998, Kluwer Academic Publishers, ISBN 0-7923-8086-X.

[3] F. Catthoor, et al., *"Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design"*Kluwer Academic Publishers, 1998.

[4] E.G. Daylight, D. Atienza, et al., *"Memory-Access-Aware Data Structure Transformations for Embedded Software with Dynamic Data Accesses"*To appear in: Special Issue on IEEE Transactions on VLSI Systems, 2003/2004. http://www.imec.be/design/ddte

[5] E.G. Daylight, B. Demoen, F. Catthoor, "Formally Specifying Dnyamic Data Structures for Embedded Software Design: a More Thorough Initial Approach", DESICS Division, IMEC vzw, *technical report* B-3001 Heverlee, Belgium, January 2004. http://www.imec.be/design/ddte

[6] D. Fisher, J. Lin, *"Profiling Energy Consumption on the Palm III Personal Digital Assistant"*, UC Berkeley Computer Science Division, Berkeley, CA 94720-1776

[7] S. Ishtiaq, P.W. O'Hearn, "BI as an Assertion Language for Mutable Data Structures", *Proc. of the 28th ACM-SIGPLAN*London, Jan. 2001.

[8] W. Kelly, W. Pugh, "A framework for unifying reordering transformations", Dept of CS, Univ. of Maryland, *technical report CS-TR-3193*, College Park MD, USA, April 1993.

[9] P. O'Hearn, J. Reynolds, H. Yang, "Local Reasoning about Programs that Alter Data Structures", *Proceedings of CSL'01*, Paris, 2001. Pages 1-19, LNCS 2142 ©Springer-Verlag.

[10] D.J. Pym, *"The Semantics and Proof Theory of the Logic of Bunched Implications"*Kluwer Academic Publishers, 2002, ISBN 1-4020-0745-0.

[11] J.C. Reynolds, “Separation Logic: A Logic for Shared Mutable Data Structures”, *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, July 22-25, 2002 in Copenhagen, Denmark.

[12] D. Singh, J. Rabaey, et al., *“Power conscious CAD tools and methodologies: a perspective”*, special issue on “Low power electronics” of the Proc. of the IEEE, Vol. 83, No.4, pp. 570-594, April 1995.

[13] N. Vijaykrishnan, M. Kandemir, et al., *“Evaluating integrated hardware-software optimisations using a unified energy estimation framework”*, IEEE Transactions on Computers, Vol. 5.2, No.1, pp 59-75, Jan. 2003.

[14] S. Wuytack, et al., “Transforming Set Data Types to Power Optimal Data Structures”, *Proc. IEEE Intnl. Workshop on Low Power Design*, Laguna Beach CA, pp.51-56, April 1995.

[15] H. Yang, *“Local Reasoning for Stateful Programs”*, Ph.D. dissertation, UIUC, July 2001.