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

[Abstract]

Edgar G. Daylight[*]
DESICS Division, IMEC,
Kapeldreef 75, B-3001
Heverlee, Belgium
voudheus@imec.be

Bart Demoen
Dept. of Comp. Sc.
K.U.Leuven,
Celestijnenlaan 200 A, B-3001
Heverlee, Belgium
bmd@cs.kuleuven.ac.be

Francky Catthoor[†]
DESICS Division, IMEC,
Kapeldreef 75, B-3001
Heverlee, Belgium
catthoor@imec.be

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 power consumption, memory footprint, and/or execution time, data structure related optimizations are a must [1, 6, 7]. 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 our work (in progress) is to formally describe data structures and access operations –or dynamic data structures for short– that we have implemented in prior, application-related work [2]. We do this by (a) extending the syntax and semantics of Separation Logic [3, 4, 5] –a logic developed recently in the program verification community– and (b) using it as a specification language for our applications.

In this paper we specify a singly linked list (SLL) in terms of structure and access operations. To do the latter, we extend Separation Logic's syntax and semantics in order to model access operations as heap changes.

1. SPECIFY A SINGLY LINKED LIST (SLL) AS A HEAP

We reuse the initial spec of an SLL from Reynolds [5]:

$$\begin{aligned} list\,\epsilon(i) &\triangleq emp \wedge (i = nil) \\ list\,(a \mathbin{\bullet} \alpha)\,(i) &\triangleq \exists j.(i \mapsto a, j) \star list\,\alpha(j) \end{aligned}$$

The spec –which is graphically represented in Figure 1– uses Separation Logic's syntax. The first line of the spec refers to the base case of the inductive definition: the list is empty (denoted by emp) and the head pointer i is nil. The second line corresponds to the inductive case: the element a and the pointer j are stored in the first record of the list and j points to the rest of the list (which is itself a list). The empty list is denoted by ϵ. A list containing the elements a, b, $c, \ldots$ is denoted as $a \mathbin{\bullet} \alpha$ in which a denotes the first element of the list and α denotes the rest of the list; i.e. the elements $b, c, \ldots$ The notation $i \mapsto a, j$ is an abbreviation for $(i \mapsto a) \star (i + 1 \mapsto j)$: the heap cell with address i contains the element a and the adjacent heap cell with address $i + 1$ contains the pointer j.

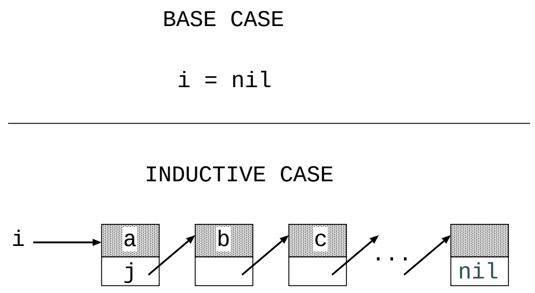

Figure 1: Singly Linked List (SLL).

As an example, we specify the *calculate* operation of an SLL as follows.

$$\begin{aligned} calculate\, i\, f\, res \;\triangleq\; & (emp \wedge (i = nil) \wedge (res = 0)) \\ & \vee \\ & (\exists v. \exists j.\ (i \mapsto v, j) \star (calculate\, j\, f\ (res - (f\ v)))) \end{aligned}$$

If $calculate\, i\, f\, res$ holds true, then the result $res = f(a) + f(b) + f(c) + \ldots$ where the list contains the elements $a, b, c, \ldots$ The beginning of the list is characterized by the head pointer i.

[*]Also at Dept. of Comp. Sc. K.U.Leuven, Belgium.
[†]Also at K.U.Leuven, Belgium.

2. SPECIFY SLL AS HEAP CHANGE

How can we specify the appending of an element to an SLL? Is Separation Logic's original syntax appropriate for this? The answer is no: we need to distinguish between an input heap h_i and an output heap h_o in order to model a heap change as opposed to just modelling one heap.

The following spec (see also Figure 2) uses additional syntax and semantics which is formalized in Section 4. In order to *append* an element a to an SLL, we distinguish between two cases. The first line in the spec corresponds to the first case and the other lines correspond to the second case of Figure 2.

$$\begin{aligned} append\, i\, a\, j \quad &\triangleq \quad ((emp_i \wedge (i = nil)) \wedge (j \mapsto_o a, nil)) \\ &\qquad\qquad \vee \\ \exists m.\exists m'.\exists b\ ((i = j) \wedge &\left((i \mapsto_i b, m) \wedge \left(j \mapsto_o b, m'\right)\right) \\ &\star \left(append\, m\, a\, m'\right)\right) \end{aligned}$$

In the first case, the initial (or input) list is empty (denoted by emp_i where i stands for input), the head pointer i of the list is nil, and the output list contains a record $< a, nil >$. In other words, the change described here corresponds to appending an element a to an empty list.

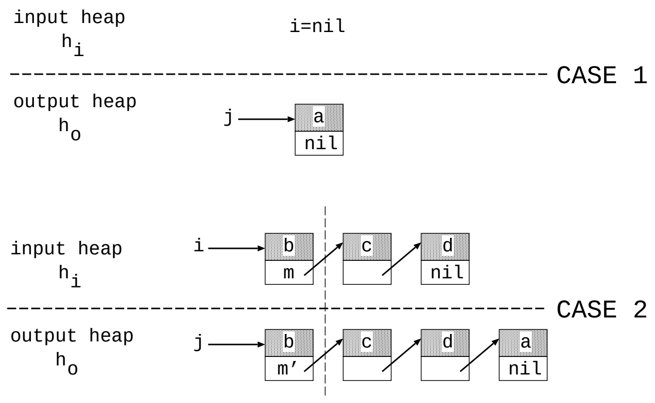

Figure 2: *append i a j*: Appending a record to an SLL.

In the second case, the input list is not empty: the head pointer i points to the first record $< b, m >$ in the input list. Since (in this example) the element a is appended to the end of the list, the head pointer j of the output list also points to b. This is denoted by $j \mapsto_o b, m'$ (where o stands for output). The pointer m' is the new pointer that serves as the head pointer of the newly obtained rest of the list (i.e. after the element a has been appended to the rest of the input list). This is expressed by the recursive call to $append\, m\, a\, m'$.

Similarly, removing an element b from the (singly linked) list is specified as follows.

$$\begin{aligned} remove\, i\, b\, j \quad &\triangleq \quad ((i \mapsto_i b, j) \wedge emp_o) \star Same \\ &\qquad\qquad \vee \\ &\left(\exists m.\exists m'.\exists a.\ (a \neq b)\ \wedge\ (i = j)\ \wedge\right. \\ &\left.\left((i \mapsto_i a, m) \wedge \left(i \mapsto_o a, m'\right)\right) \star \left(remove\, m\, b\, m'\right)\right) \end{aligned}$$

To be complete, we also respecify the structure of the SLL (cfr. Figure 1) in the context of heap changes:

$$\begin{aligned} list\, \epsilon(i) \quad &\triangleq \quad Same\,(emp) \wedge (i = nil) \\ list\ (a \centerdot \alpha)\ (i) \quad &\triangleq \quad \exists j.\ Same(i \mapsto a, j) \star list\, \alpha(j) \end{aligned}$$

We use $Same(emp)$ to express that both the input heap and the output heap are empty. Similarly, $Same(i \mapsto a, j)$ is equivalent to $(i \mapsto_i a, j) \wedge (i \mapsto_o a, j)$.

3. REINTRO TO SEPARATION LOGIC

We briefly reintroduce Separation Logic. All fundamental concepts are borrowed from [3, 4, 5]. In the next section we extend the logic in order to model heap changes.

Stack vs. Heap

To describe a data organisation, we use a stack $s \in S$ and a heap $h \in H$.

$$\begin{aligned} Val &= Int \cup Atoms \cup Loc \\ S &= Var \rightharpoonup_{fin} Val \\ H &= Loc \rightharpoonup_{fin} Val \end{aligned}$$

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.

Expressions

Expressions are:
$E ::= x$ (variable) | 42 (integer) | nil (nil) | a (atom) | l (location) | $E_1 + E_2$ (addition) | $E_1 - E_2$ (subtraction) | ...
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.

Syntax

Separation Logic is an extension of classical (predicate) logic. The empty heap, spatial conjunction, and spatial implication[1] constitute this extension.

The non-atomic formulae β are:

β	$::=$	α	Atomic Formulae
	$\mid$	$false$	Falsity
	$\mid$	$P \Rightarrow Q$	Classical Implication
	$\mid$	emp	Empty Heap
	$\mid$	$P \star Q$	Spatial Conjunction
	$\mid$	$P \mathrel{-\!\!\star} Q$	Spatial Implication
	$\mid$	$\exists x.P$	Existential Quantification

where P and Q are non-atomic formulae.

The atomic formulae α are:

α	$::=$	$E_1 = E_2$	Equality
	$\mid$	$E_1 < E_2$	Smaller than
	$\mid$	$E_1 \leq E_2$	Smaller than or equal to
	$\mid$	$E_1 \mapsto E_2$	Points to
	$\mid$	$\ldots$	

where E_1 and E_2 are expressions. We use $E_1 \leq E_2$ as an abbreviation for $(E_1 = E_2) \vee (E_1 < E_2)$.

The other connectives (such as $\neg$, $\vee$, $\wedge$) are defined in terms of those presented above. For instance: $\neg P = P \Rightarrow false$. We define the set $free(P)$ of free variables of a formula as usual.

[1]We do not use spatial implication in this paper and therefore omit it when discussing the semantics.

Notation

We use $h \perp h'$ to denote that heaps h and h' are disjoint: $dom(h) \cap dom(h') = \varnothing$. Also, $h \bullet h'$ denotes the union of disjoint heaps (i.e. the union of functions with disjoint domains).

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 $free(P) \subseteq dom(s)$. An expression E is interpreted as a heap-independent value $[[E]]\, s \in Val$ where the $dom(s)$ includes the free variables of E. We present two semantic clauses below; the rest can be found in [3].

$s, h \models E_1 \mapsto E_2$	iff	$\{[[E_1]]\, s\} = dom(h)$ and $h([[E_1]]\, s) = [[E_2]]\, s$
$s, h \models P \star Q$	iff	$\exists h_0, h_1.\ h_0 \bullet h_1 = h,$ $s, h_0 \models P$ and $s, h_1 \models Q$

4. HEAP CHANGES

We extend the original syntax and semantics of Section 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.

Syntax

The additional, non-atomic formulae β are:

β	$::=$	emp_i	Empty Input Heap
	$\mid$	emp_o	Empty Output Heap
	$\mid$	$Same$	Identical Input & Output Heaps (IIOH)
	$\mid$	$Same\,(R)$	IIOH and both model R
	$\mid$	$P; Q$	Sequential Composition

where P and Q are non-atomic formulae and R is a non-atomic formula that describes only one heap. In other words, R is a non-atomic formula of Section 3.

The additional atomic formulae α are:

α	$::=$	$E_1 \mapsto_i E_2$	Points to Relation in Input Heap
	$\mid$	$E_1 \mapsto_o E_2$	Points to Relation in Output Heap

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.

Notation

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, $(h_i, h_o) \bullet \left(h_i', h_o'\right)$ denotes $\left(h_i \bullet h_i',\ h_o \bullet h_o'\right)$.

Semantics

The semantics of assertions are given by:

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

The basic domains of Section 3 remain unchanged. All semantic clauses are defined below. Note that in the last clause we use the relation $s, h \models R$ of Section 3.

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

5. ACKNOWLEDGEMENTS

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

6. REFERENCES

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

[2] E.G. Daylight, et al., *"Memory-Access-Aware Data Structure Transformations for Embedded Software with Dynamic Data Accesses"*, To appear in: Special Issue on Low Power Electronics, 2003.

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

[4] 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.

[5] J.C. Reynolds, "Separation Logic: A Logic for Shared Mutable Data Structures", *Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science*, July 22-25, 2002 in Denmark.

[6] D. Singh, et al., *"Power conscious CAD tools and methodologies: a perspective"*, Special Issue on Low Power Electronics, IEEE, Vol.83, No.4, pp.570-594, April 1995.

[7] N. Vijaykrishnan, et al., *"Evaluating integrated HW-SW optimisations using a unified energy estimation framework"*, IEEE Trans on Computers, Vol.5.2, No.1, pp59-75, Jan. 2003.