Transitive Separation Logic

Han-Hing Dang, Bernhard Möller

RAMiCS 2012

1 Introduction

- purpose of separation logic (SL): reasoning about linked object/record structures
- original sequential version has also been extended to concurrent contexts
- this talk: concentrate on the data structure level
- algebraic description of control structure level
  has been given elsewhere

phenomena to be treated:

- reachability (garbage collection)
- (absence of) sharing
- cycle detection
- preservation of substructures under destructive assignments

we sketch some algebraic tools for and some useful extensions of SL

2 Brief Recap of SL

extends Hoare logic

formulas talk not only about program variables, but also about heap portions (*heaplets*) i.e., partial functions from addresses to values (which include addresses)

central new connective: separating conjunction

$$P_1 * P_2$$

holds for a given heaplet h iff

- h can be partitioned into heaplets h_1, h_2
- i.e., $h = h_1 \cup h_2$ and $\lceil h_1 \cap \lceil h_2 = \emptyset$ ($\lceil$ is the *domain* operator)
- and P_i holds for h_i

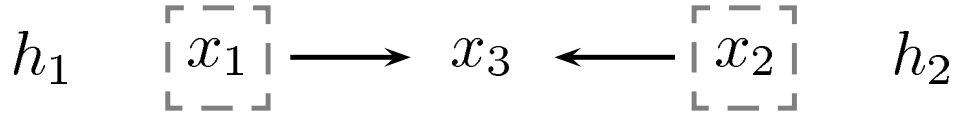
3 A Limitation

in many cases the separation expressed by $\lceil h_1 \cap \lceil h_2 = \emptyset$ is too weak

Example immediate sharing

assume addresses x_1, x_2, x_3

$$h_1 : x_1 \mapsto x_3 \qquad h_2 : x_2 \mapsto x_3$$



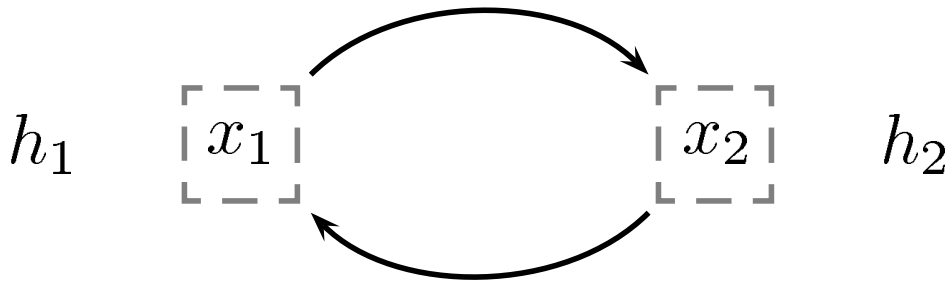
satisfy the above separation property, but $h = h_1 \cup h_2$

does not appear very separated □

Example a simple cycle

addresses x_1, x_2

$$h_1 : x_1 \mapsto x_2 \qquad h_2 : x_2 \mapsto x_1$$



again, h_1, h_2 satisfy the separation property □

we want to find a stronger separation condition that takes such phenomena into account

4 Image and Reachability

central notion: reachability (in one or more steps)

abstraction: forget about non-pointer attributes of objects and about multiple links from one object to another

then a linked object structure corresponds to an *access relation* a between objects

more abstract view: take a to be an element of a modal Kleene algebra $(S, +, 0, \cdot, 1, ^*, \lceil, \rceil)$

let $p \leq 1$ be a test representing a set of objects (identified by pointers)

the modal diamond operator $\langle a|\, p$ yields the *image* of p under a:, i.e., the set of immediate successors of p-objects under a:

$$\langle a|\, p =_{df} (p \cdot a)^{\urcorner}$$

where $p \cdot a$ is the restriction of a to start objects in p and $^{\urcorner}$ is the *codomain* operator

as usual the reflexive transitive closure of an access element a is a^*

the reachability function can now be defined as

$$\mathit{reach}(p, a) =_{df} \langle a^*|\, p$$

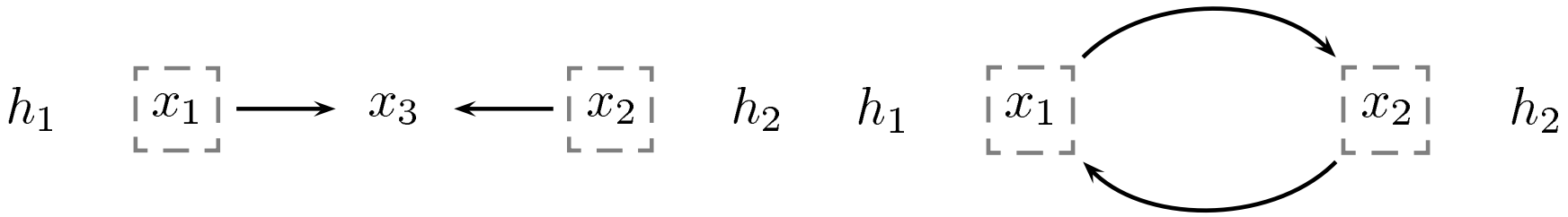
5 Strong Separation

now we can formulate a stronger separation relation $\circledast$

$$a_1 \circledast a_2 \Leftrightarrow_{df} \mathit{reach}(\ulcorner a_1, a) \cdot \mathit{reach}(\ulcorner a_2, a) = 0$$

where $a = a_1 + a_2$

this rules out the above two examples



by $p \leq \langle b^* | p$ for all p, b, the new separation condition indeed implies the analogue of the old one:

$$a_1 \mathbin{⊛} a_2 \;\Rightarrow\; \lceil a_1 \cdot \lceil a_2 = 0$$

moreover, $⊛$ is downward closed by isotony of *reach*

central property:

$$a \mathbin{⊛} b \;\Leftrightarrow\; \overline{\lceil a} \cdot \overline{\lceil b} = 0$$

where $\overline{\lceil a} =_{df} \lceil a + a \rceil$ is the set of *nodes* of access element a

using $\circledast$ we define a stronger variant of $*$

$$P_1 \circledast P_2$$

holds for an access relation a iff

- a can be strongly partitioned into some a_1, a_2
- i.e., $a = a_1 + a_2$ and $a_1 \circledast a_2$,
- and P_i holds for a_i

Lemma $\circledast$ is commutative and associative

why does "classical" SL get along without this stronger notion?

some aspects of it can be welded implicitly into recursive data type predicates

Example singly linked lists (x, y addresses)

$$h \models \mathit{list}(\mathsf{nil}) \quad \Leftrightarrow \quad h = \emptyset$$

$$x \neq \mathsf{nil} \Rightarrow$$

$$(h \models \mathit{list}(x) \Leftrightarrow \exists y : h \models [x \mapsto y] * \mathit{list}(y))$$

we will elaborate on this in the next section

note that using $\circledast$ instead of $*$ would not work, because the heaplets used are not strongly separate

6 Nil and Closed Access Relations

$\square$ is a special element characterising the pseudo-pointer nil/`null`

call a *proper* if $\square \sqcap \lceil a = 0$ (equivalently $\square \cdot a = 0$) and
closed if $b^{\rceil} \leq \lceil b + \square$ (no dangling pointers)

Lemma For proper and closed a_i we have $\lceil a_1 \cdot \lceil a_2 = 0 \Rightarrow a_1 \mathbin{\#} a_2$

it can be shown by induction that all access relations characterised by the analogue of the *list* predicate are closed

this is why for a large part of SL the weak separation suffices

7 An Algebra of Linked Structures

call an access element a

- *acyclic* iff for all atomic tests $p \neq \Box$ we have $p \cdot a^+ \cdot p = 0$, where $a^+ = a \cdot a^*$
- *deterministic* iff $\forall p : \langle a|\,|a\rangle\, p \leq p$, where the dual diamond is defined by $|a\rangle\, p = \ulcorner(a \cdot p)$
- *injective* if $\forall p : |a'\rangle\, \langle a'|\, p \leq p$ where $a' = a \cdot \neg\Box$

assume now a finite set L of *selector names*, e.g., left or right in binary trees, and a modal Kleene algebra S.

- A *linked structure* is a family $a = (a_l)_{l \in L}$ of proper and deterministic access elements $a_l \in S$ (access along each particular selector is deterministic)
- associated overall access element: $\Sigma_{l \in L}\, a_l$, again denoted by a
- a is a *forest* if a is acyclic and injective
- a forest a is a *tree* if $\overline{a} = \langle a^* | r$ for some atomic test r, called the *root* of a

we now define programming constructs and assertions

- a *store* is a mapping from program identifiers to atomic tests
- a *state* is a pair (s, a) with a store s and an access element a
- for identifier i and selector name l we define the semantics

$$[\![i.l]\!]_{(s,a)} =_{df} \langle a_l | (s(i))$$

- program *commands* are relations between states
- the semantics of plain assignment $i := e$ and Hoare triples is defined as usual

assignments of the form $i.l := e$ will be discussed below

assume atomic tests with $p \cdot q = 0 \wedge p \cdot \Box = 0$

- a *twig* is a tree of the form $p \mapsto q =_{df} p \cdot \top \cdot q$
- the corresponding *update* is $(p \mapsto q) \mid a =_{df} (p \mapsto q) + \neg p \cdot a$
- intuitively, the single node of p is connected to the single node in q, while a is restricted to links that start from $\neg p$ only
- For identifier i, selector name l and expression e we set $i.l := e =_{df} \{((s,a),(s,(s(i) \mapsto [\![e]\!]_{(s,a)}) \mid a_l) : s(i) \neq \Box, s(i) \leq \lceil a \}$

in general such an assignment does not preserve treeness

8 Directed Separation

assume a tree a

- the set of *terminal nodes* is $terms(a) =_{df} \lceil a \rceil - \lceil a$
- we define *directed combinability* (assuming a_2 to be a tree) by

$$a_1 \triangleright a_2 \quad \Leftrightarrow_{df} \quad \lceil a_1 \cdot \overline{\lceil a_2} = 0 \wedge \lceil a_1 \rceil \cdot \lceil a_2 \rceil \leq \Box \wedge$$
$$\lceil a_1 \rceil \cdot \lceil a_2 = root(a_2)$$

- guarantees domain disjointness and excludes cycles, since

$$\lceil a_1 \cdot \overline{\lceil a_2} = 0 \Leftrightarrow \lceil a_1 \cdot \lceil a_2 = 0 \wedge \lceil a_1 \cdot terms(a_2) = 0$$

- excludes links from non-terminal a_1-nodes to non-root a_2-nodes
- ensures that a_1 and a_2 can be combined by identifying some non-nil terminal node of a_1 with the root of a_2

we set tree $=_{df} \{a : a \text{ is a tree}\}$

for $P_1, P_2 \subseteq$ tree we define *directed combinability* $\circledcirc$ by

$$P_1 \circledcirc P_2 =_{df} \{a_1 + a_2 : a_i \in P_i, a_1 \triangleright a_2\}$$

this allows, conversely, talking about decomposability: if $a \in P_1 \circledcirc P_2$ then a can be split into two disjoint parts a_1, a_2 such that $a_1 \triangleright a_2$ holds

for selector name l, an *l-context* is a linked structure a such that a_l is a *linkable cell*, i.e., has an atomic domain

hence a has a "hole" as its l-branch

the corresponding predicate is $l\_context =_{df} \{a : a \text{ is an } l\text{-context}\}$

Lemma (Structure preservation) For predicates Q, R and selector name $l \in L$ we have

$$\{(l\_context(i) \circledcirc Q) \circledast R(j)\} \quad i.l := j \quad \{(l\_context(i) \circledcirc R(j)) \circledast Q\}$$

$$\{(Q \circledcirc l\_context(i)) \circledast R(j)\} \quad i.l := j \quad \{Q \circledcirc (l\_context(i) \circledcirc R(j))\}$$

$$\{l\_context(i)) \circledcirc Q\} \quad j := i.l \quad \{l\_context(i) \circledcirc Q(j)\}$$

9 Example: In-Situ List Reversal

- list is the set of all trees with the only selector next
- abstraction function li_a for $a \in \mathsf{list}$

$$li_a(p) =_{df} \begin{cases} \langle\rangle & \text{if } p \cdot \ulcorner a = 0 \text{ ,} \\ \langle p \rangle \bullet li_a(\langle a | p) & \text{otherwise ,} \end{cases}$$

  where $\bullet$ is concatenation and $\langle\rangle$ is the empty word

- semantics of the expression $i^{\rightarrow}$ for a program identifier i:

$$[\![i^{\rightarrow}]\!]_{(s,a)} =_{df} li_a(s(i))$$

algorithm:

$$\mathsf{j} := \square \;;\; \text{while } (\mathsf{i} \neq \square) \text{ do } \big(\, \mathsf{k} := \mathsf{i.next} \;;\; \mathsf{i.next} := \mathsf{j} \;;\; \mathsf{j} := \mathsf{i} \;;\; \mathsf{i} := \mathsf{k} \,\big)$$

invariant ($\_^\dagger$ is word reversal):

$$\mathrm{I} \;\Leftrightarrow_{df}\; (\mathsf{j}^{\rightarrow})^\dagger \bullet \mathsf{i}^{\rightarrow} = \alpha$$

$$\mathrm{I} \;\Leftrightarrow_{df}\; (\mathsf{j}^{\rightarrow})^{\dagger} \bullet \mathsf{i}^{\rightarrow} = \alpha$$

$$\{\ \mathsf{list}\,(\mathsf{i}) \wedge \mathsf{i}^{\rightarrow} = \alpha\ \}$$

$\mathsf{j} := \square\,;$ $\quad \{\ (\mathsf{list}\,(\mathsf{i}) \circledast \mathsf{list}\,(\mathsf{j})) \wedge \mathrm{I}\ \}$

$\mathsf{while}\ (\mathsf{i} \neq \square)\ \mathsf{do}\ ($ $\quad \{\ ((\mathsf{l\_cell}\,(\mathsf{i}) \circledcirc \mathsf{list}\,(\mathsf{i.next})) \circledast \mathsf{list}\,(\mathsf{j})) \wedge \mathrm{I}\ \}$

$\mathsf{k} := \mathsf{i.next}\,;$ $\quad \{\ ((\mathsf{l\_cell}\,(\mathsf{i}) \circledcirc \mathsf{list}\,(\mathsf{k})) \circledast \mathsf{list}\,(\mathsf{j})) \wedge (\mathsf{j}^{\rightarrow})^{\dagger} \bullet \mathsf{i} \bullet \mathsf{k}^{\rightarrow} = \alpha\ \}$

$\{\ ((\mathsf{l\_cell}\,(\mathsf{i}) \circledcirc \mathsf{list}\,(\mathsf{k})) \circledast \mathsf{list}\,(\mathsf{j})) \wedge (\mathsf{i} \bullet \mathsf{j}^{\rightarrow})^{\dagger} \bullet \mathsf{k}^{\rightarrow} = \alpha\ \}$

$\mathsf{i.next} := \mathsf{j}\,;$ $\quad \{\ ((\mathsf{list}\,(\mathsf{i}) \circledast \mathsf{list}\,(\mathsf{k})) \wedge (\mathsf{i}^{\rightarrow})^{\dagger} \bullet \mathsf{k}^{\rightarrow} = \alpha\ \}$

$\mathsf{j} := \mathsf{i}\,;\ \mathsf{i} := \mathsf{k}\,;$ $\quad \{\ (\mathsf{list}\,(\mathsf{j}) \circledast \mathsf{list}\,(\mathsf{i})) \wedge \mathrm{I}\ \}$

$)$ $\quad \{\ \mathsf{list}\,(\mathsf{j}) \wedge (\mathsf{j}^{\rightarrow})^{\dagger} = \alpha\ \}$

$\{\ \mathsf{list}\,(\mathsf{j}) \wedge \mathsf{j}^{\rightarrow} = \alpha^{\dagger}\ \}$

- each assertion consists of a structural part and a part connecting the concrete and abstract levels of reasoning
- unlike standard SL we hide the existential quantifiers that were necessary there to describe the sharing relationships
- structural correctness properties of the occurring data structures and their interrelationship captured by the the new separation predicates
- quantifiers to state functional correctness not needed due to use of the abstraction function
- hence the formulas become easier to read and more concise

10 Conclusion and Outlook

achievements

- relating the approach of pointer Kleene algebra with SL
- algebra useful for stating and proving reachability conditions
- extended operations, similar to separating conjunction, that additionally assert structural properties of linked object structures
- concrete predicates and operations on linked lists and trees that enabled correctness proofs of an in-situ list-reversal algorithm and tree rotation
- a novel use of assertions with an abstraction function to further reduce the amount of quantifiers

future work

- explore more complex or other linked object structures such as doubly-linked lists or threaded trees
- tackle more complex algorithms like Schorr-Waite Graph Marking or concurrent garbage collection