Separation Logic: A Logic for Shared Mutable Data Structures

Paper presentation of John C. Reynolds

Ramūnas Gutkovas[1]

[1]Uppsala University
Department of Information Technology

May 11, 2012

Motivation

Example

Reversing a linked-list in place

$$
\begin{array}{l}
j := \textbf{nil}; \\
\textbf{while } i \neq \textbf{nil do } (\\
\qquad k := [i+1]; \\
\qquad [i+1] := j; \\
\qquad j := i; \\
\qquad i := k \ )
\end{array}
$$

Input/Output

i is a pointer to a linked-list being reversed. After execution, j points to the reversed list.

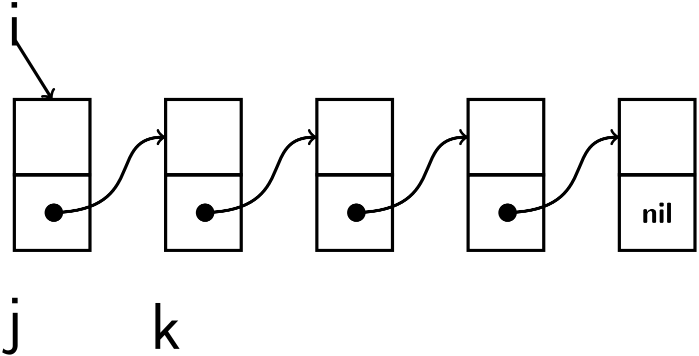
Invariant

What is invariant of the loop?

Motivation

Example

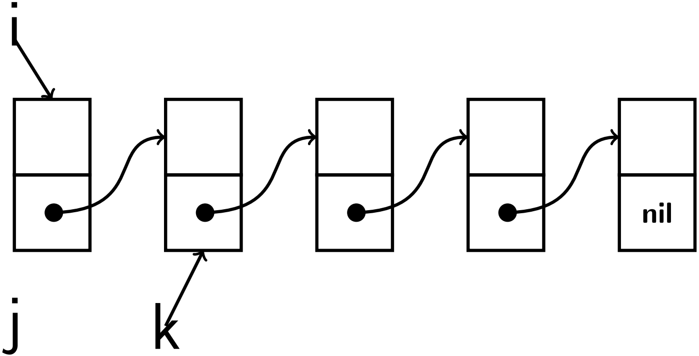
$$k := [i+1]$$
$$[i+1] := j$$
$$j := i$$
$$i := k$$



Motivation

Example

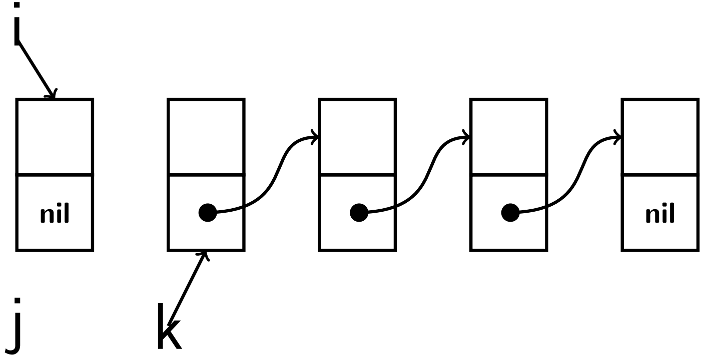
$\rightarrow$ $k := [i + 1]$
$[i + 1] := j$
$j := i$
$i := k$



Motivation

Example

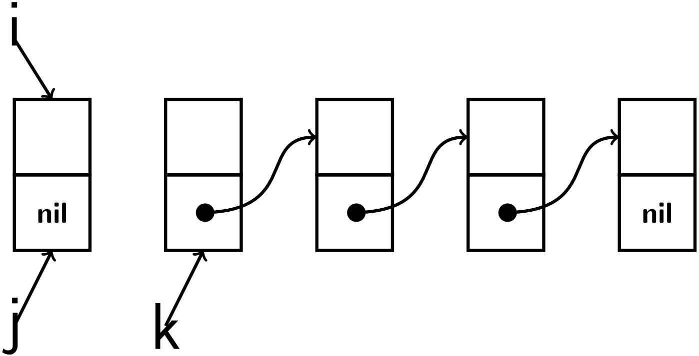
$$k := [i + 1]$$
$$\rightarrow [i + 1] := j$$
$$j := i$$
$$i := k$$



Motivation

Example

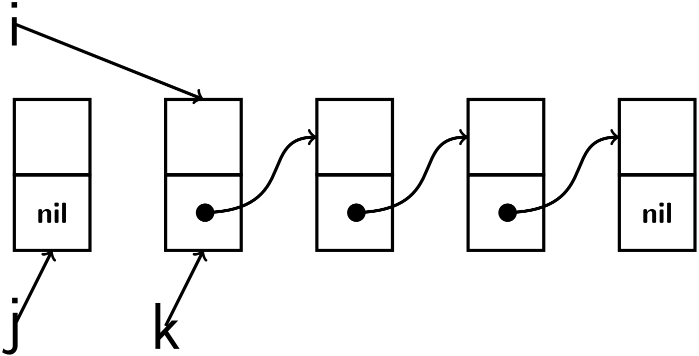
$k := [i + 1]$
$[i + 1] := j$
$j := i$
$i := k$



Motivation

Example

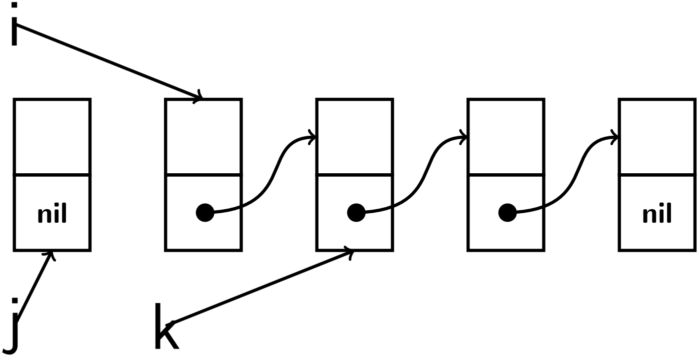
$k := [i+1]$
$[i+1] := j$
$j := i$
$i := k$



Motivation

Example

- $k := [i + 1]$
  $[i + 1] := j$
  $j := i$
  $i := k$



Motivation

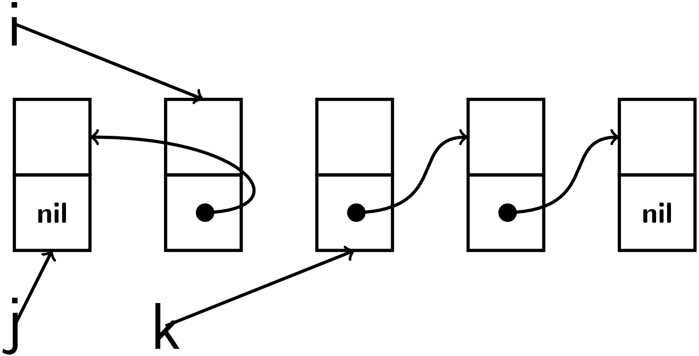
Example

$k := [i+1]$

$\rightarrow$ $[i+1] := j$

$j := i$

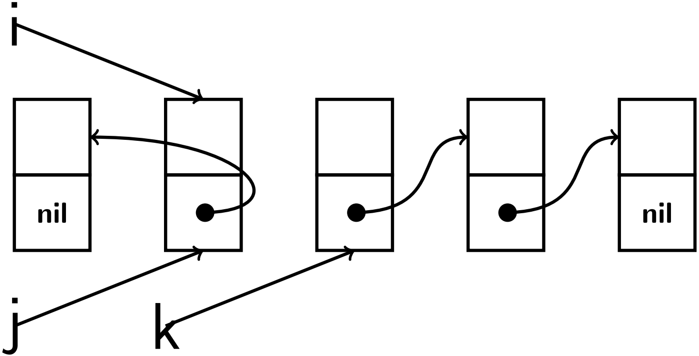
$i := k$



Motivation

Example

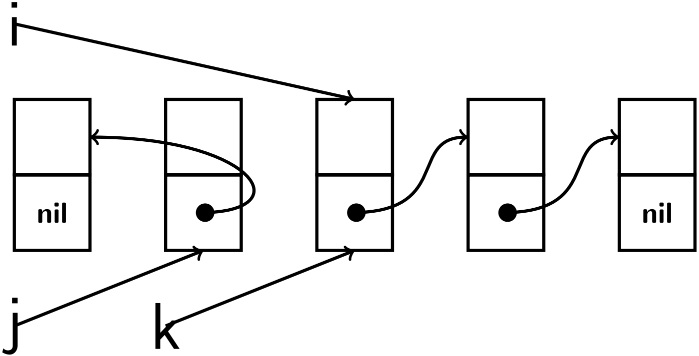
$k := [i + 1]$
$[i + 1] := j$
→ $j := i$
$i := k$



Motivation

Example

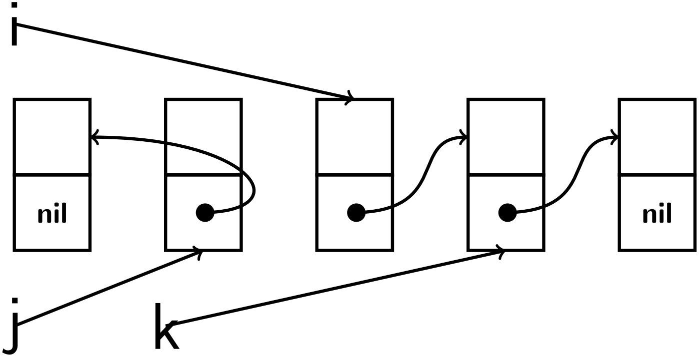
$k := [i + 1]$
$[i + 1] := j$
$j := i$
→ $i := k$



Motivation

Example

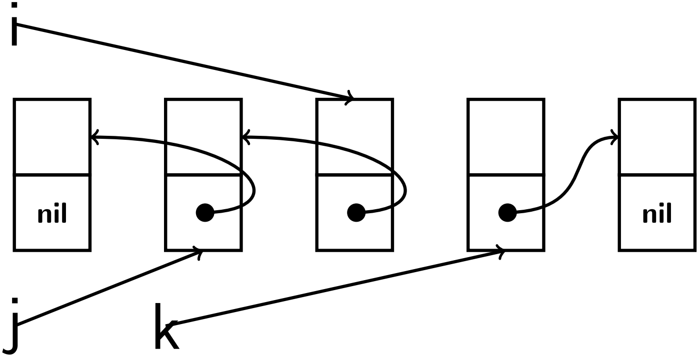
→ $k := [i+1]$
$[i+1] := j$
$j := i$
$i := k$



Motivation

Example

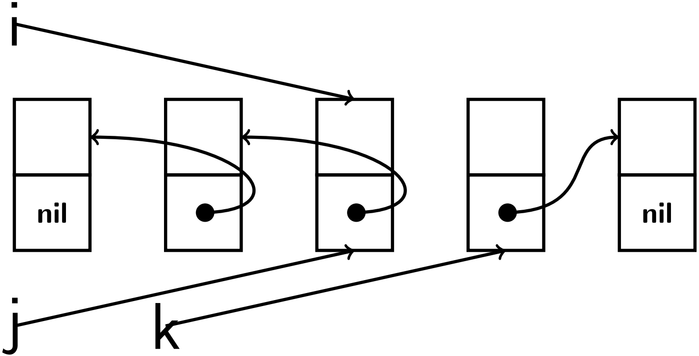
$k := [i + 1]$
$\rightarrow$ $[i + 1] := j$
$j := i$
$i := k$



Motivation

Example

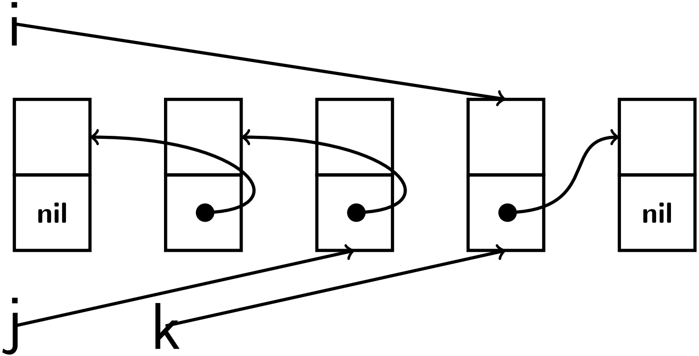
$k := [i + 1]$
$[i + 1] := j$
$\rightarrow$ $j := i$
$i := k$



Motivation

Example

$$k := [i+1]$$
$$[i+1] := j$$
$$j := i$$
$$\rightarrow \; i := k$$



Motivation

Example

- Suppose we have a predicate which relates sequences and their representation in the program.

  $$\mathsf{list}\,\alpha\,i$$

  Asserts that α is a sequence represented by the linked list pointed by the variable i.
- Then the invariant of the loop may look like

  $$\exists\alpha,\beta.\ \mathsf{list}\,\alpha\,i\ \wedge\ \mathsf{list}\,\beta\,j\ \wedge\ \alpha_0^\dagger = \alpha^\dagger\cdot\beta$$

  where α_0 is the given sequence (initially pointed by i), $\alpha^\dagger$ is reflection of α, and $\alpha\cdot\beta$ is concatenation of α,β.

Motivation

Example

Reversing a linked-list in place and ?

$$
\begin{array}{l}
\textbf{while } i \neq \textbf{nil do } (\\
\qquad k := [i+1]; \\
\qquad [i+1] := j; \\
\qquad j := i; \\
\qquad i := k \ )
\end{array}
$$

Input/Output

i and j are pointers to linked lists.

Invariant

Do we need to change the invariant?

Motivation

Example

Reversing a linked-list in place and appending[1]

$$\begin{array}{l}\textbf{while } i \neq \textbf{nil do } (\\ \qquad k := [i+1]; \\ \qquad [i+1] := j; \\ \qquad j := i; \\ \qquad i := k \ )\end{array}$$

Input/Output

i and j are pointers to linked lists. After execution, j points to a list where the initial segment is the reveres i list and the tail is j list before execution.

Invariant

Do we need to change the invariant?

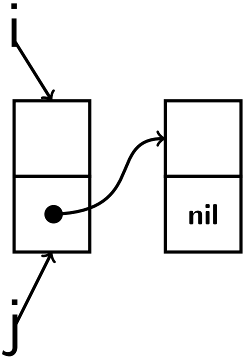
[1] e.g., `revappend` in CL

Motivation

Example

If the lists i and j are shared.

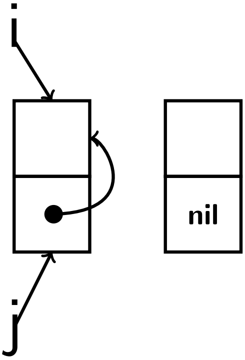
$k := [i+1]$
$[i+1] := j$
$j := i$
$i := k$



Motivation

Example

If the lists i and j are shared.



$k := [i + 1]$
$[i + 1] := j$
$j := i$
$i := k$

Motivation
Example

- We need to extend the invariant which asserts that the lists cannot be shared.

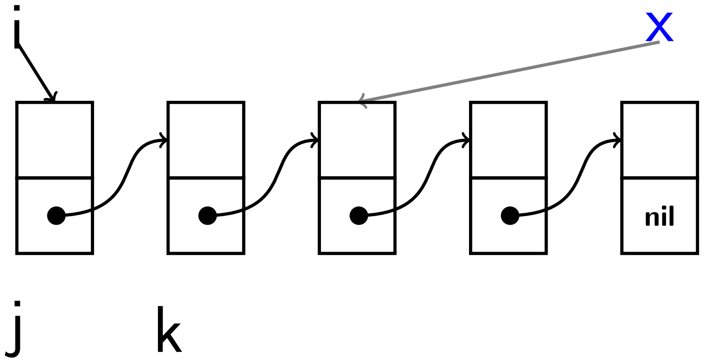
$$\begin{aligned}\exists \alpha, \beta.\ \mathsf{list}\,\alpha\, i \ \wedge\ \mathsf{list}\,\beta\, j \ \wedge\ \alpha_0^\dagger = \alpha^\dagger \cdot \beta \\ \wedge\ (\forall k.\ \mathsf{reach}\, i\, k \wedge \mathsf{reach}\, j\, k \implies k = \mathbf{nil})\end{aligned}$$

where the predicate reach $i\, j$ tells that there is a path from the pointer i to j by following the links in the linked list linked by i.

Motivation

Example

The list m shares the structure of the list i



Unwanted result

The execution of the algorithm would affect m.

Motivation

Example

- Now we need to talk about, in the invariant, that we did not accidentally clobber an unrelated list!

$$
\begin{aligned}
(\exists \alpha, \beta.\ \mathsf{list}\,\alpha\, i \ \wedge\ \mathsf{list}\,\beta\, j \ \wedge\ \alpha_0^\dagger = \alpha^\dagger \cdot \beta) \wedge \mathsf{list}\,\gamma\, x \\
\wedge\ (\forall k.\ \mathsf{reach}\, i\, k \wedge \mathsf{reach}\, j\, k \implies k = \mathbf{nil}) \\
\wedge\ (\forall k.\ \mathsf{reach}\, x\, k \wedge (\mathsf{reach}\, i\, k \vee \mathsf{reach}\, j\, k) \implies k = \mathbf{nil})
\end{aligned}
$$

- This is just 5 lines of code!

Enter Separation Logic

- The achievement of Separation Logic is to invert the reasoning: instead of specifying what is not shared, one specifies what is.
- This is done by introducing a logical operation called *separating conjunction* denoted $P * Q$.
- This logical operation allows us to express the loop invariant intuitively

$$\exists \alpha, \beta.\ (\mathsf{list}\, \alpha\, i \ * \ \mathsf{list}\, \beta\, j) \ \wedge \ \alpha_0^\dagger = \alpha^\dagger \cdot \beta$$

Separation Logic

How does it work: The Language

The programming language is an extension of Hoare's imperative language with primitives for the manipulation of mutable shared data structures. Semantically, computation states contain a *store* and a *stack*.

- $x := \mathbf{cons}(e_1, \ldots, e_n)$
  Allocates new active cells in the heap and assigns the address of the first cell to the variable x in the store. Never aborts.

Separation Logic

How does it work: The Language

The programming language is an extension of Hoare's imperative language with primitives for the manipulation of mutable shared data structures. Semantically, computation states contain a *store* and a *stack*.

- $x := \mathbf{cons}(e_1, \ldots, e_n)$
  Allocates new active cells in the heap and assigns the address of the first cell to the variable x in the store. Never aborts.
- $x := [e]$
  Dereferences the address computed from the expression e and assigns the value to x in the store. Aborts if there is no active cell at this address.

Separation Logic

How does it work: The Language

The programming language is an extension of Hoare's imperative language with primitives for the manipulation of mutable shared data structures. Semantically, computation states contain a *store* and a *stack*.

- $x := \mathbf{cons}(e_1, \ldots, e_n)$
  Allocates new active cells in the heap and assigns the address of the first cell to the variable x in the store. Never aborts.
- $x := [e]$
  Dereferences the address computed from the expression e and assigns the value to x in the store. Aborts if there is no active cell at this address.
- $[e_l] := e_r$
  Mutates the cell at the address computed from e_l. Aborts if the cell is not active.

Separation Logic

How does it work: The Language

The programming language is an extension of Hoare's imperative language with primitives for the manipulation of mutable shared data structures. Semantically, computation states contain a *store* and a *stack*.

- $x := \textbf{cons}(e_1, \ldots, e_n)$
  Allocates new active cells in the heap and assigns the address of the first cell to the variable x in the store. Never aborts.
- $x := [e]$
  Dereferences the address computed from the expression e and assigns the value to x in the store. Aborts if there is no active cell at this address.
- $[e_l] := e_r$
  Mutates the cell at the address computed from e_l. Aborts if the cell is not active.
- $\textbf{dispose}(e)$ Frees the cell at the address e. Aborts if the cell is not active.

Separation Logic

How does it work: Assertions

Assertions are an extension of the usual predicate calculus.

- **emp** empty heap
  Asserts that the heap is empty.

Assertions are an extension of the usual predicate calculus.

- **emp** empty heap
  Asserts that the heap is empty.
- $e_a \mapsto e_v$ singleton heap
  Asserts that the heap contains exactly one cell with address e_a and the value stored e_v.

Assertions are an extension of the usual predicate calculus.

- **emp** empty heap
  Asserts that the heap is empty.
- $e_a \mapsto e_v$ singleton heap
  Asserts that the heap contains exactly one cell with address e_a and the value stored e_v.
- $P * Q$ separating conjunction
  Asserts that the heap can be split into two disjoint heaps (address wise) and P is true in one, and Q is true in the other.

Separation Logic

How does it work: Assertions

Assertions are an extension of the usual predicate calculus.

- **emp** empty heap
  Asserts that the heap is empty.
- $e_a \mapsto e_v$ singleton heap
  Asserts that the heap contains exactly one cell with address e_a and the value stored e_v.
- $P * Q$ separating conjunction
  Asserts that the heap can be split into two disjoint heaps (address wise) and P is true in one, and Q is true in the other.
- $P \mathrel{-\!\!*} Q$ separating implication
  Asserts that if the heap can be extended with a disjoint heap in which P holds, then Q holds in the extended heap.

Separation Logic

- $x \mapsto 1$ asserts that x points to a cell in the heap which stores 1.

Separation Logic

- $x \mapsto 1$ asserts that x points to a cell in the heap which stores 1.
- $x \mapsto 2 * y \mapsto 2$ asserts that there are either two cells with different addresses or one cell.

Separation Logic

- $x \mapsto 1$ asserts that x points to a cell in the heap which stores 1.
- $x \mapsto 2 * y \mapsto 2$ asserts that there are either two cells with different addresses or one cell.
- $x \mapsto 2 \wedge y \mapsto 2$ asserts that x and y must be aliases.

Separation Logic

- $x \mapsto 1$ asserts that x points to a cell in the heap which stores 1.
- $x \mapsto 2 * y \mapsto 2$ asserts that there are either two cells with different addresses or one cell.
- $x \mapsto 2 \wedge y \mapsto 2$ asserts that x and y must be aliases.
- $x \mapsto 1 * \textbf{true}$ asserts that the heap contains a cell which x points to and stores 1.

Separation Logic

- $x \mapsto 1$ asserts that x points to a cell in the heap which stores 1.
- $x \mapsto 2 * y \mapsto 2$ asserts that there are either two cells with different addresses or one cell.
- $x \mapsto 2 \wedge y \mapsto 2$ asserts that x and y must be aliases.
- $x \mapsto 1 * \textbf{true}$ asserts that the heap contains a cell which x points to and stores 1.
- $(x \mapsto 1) -\!\!* \; p$ asserts that in every possible one cell extension of current heap, that cell is present in the heap where p holds.

Separation Logic

Properties

- Separation Logic is a substructural logic.

Separation Logic

Properties

- Separation Logic is a substructural logic.
- Every inference rule of first order logic remains sound.

Separation Logic

Properties

- Separation Logic is a substructural logic.
- Every inference rule of first order logic remains sound.
- But contraction and weakening are not sound for the separating conjunction, i.e.

$$p \implies p * p \qquad p * q \implies p$$

are *not* sound in general.

Separation Logic

Properties

- Separation Logic is a substructural logic.
- Every inference rule of first order logic remains sound.
- But contraction and weakening are not sound for the separating conjunction, i.e.

  $$p \implies p * p \qquad p * q \implies p$$

  are *not* sound in general.
- Separating conjunction, $*$, is commutative, associative, has a unit (namely **emp**), $*$ distributes over conjunction and disjunction.

Separation Logic

Specification

- Specification is Hoare triple:

Separation Logic

Specification

- Specification is Hoare triple:
  - Partial: $\{p\}c\{q\}$.

Separation Logic

Specification

- Specification is Hoare triple:
  - Partial: $\{p\}c\{q\}$.
  - Total: $[p]c[q]$.

Separation Logic

Specification

- Specification is Hoare triple:
  - Partial: $\{p\}c\{q\}$.
  - Total: $[p]c[q]$.
- The usual Hoare inference rules for triples hold: consequence, auxiliary variable elimination, substitution.

Separation Logic

Specification

- Specification is Hoare triple:
  - Partial: $\{p\}c\{q\}$.
  - Total: $[p]c[q]$.
- The usual Hoare inference rules for triples hold: consequence, auxiliary variable elimination, substitution.
- Except for the rule of constancy

$$\frac{\{p\}c\{q\}}{\{p \wedge r\}c\{q \wedge r\}}$$

Separation Logic

Specification

- Specification is Hoare triple:
  - Partial: $\{p\}c\{q\}$.
  - Total: $[p]c[q]$.
- The usual Hoare inference rules for triples hold: consequence, auxiliary variable elimination, substitution.
- Except for the rule of constancy

$$\frac{\{p\}c\{q\}}{\{p \wedge r\}c\{q \wedge r\}}$$

- Example of failure

$$\frac{\{\exists z.x \mapsto z\}[x] := 4\{x \mapsto 4\}}{\{(\exists z.x \mapsto z) \wedge y \mapsto 3\}c\{x \mapsto 4 \wedge y \mapsto 3\}}$$

The postcondition in the conclusion does not hold, since x and y are not aliases.

Separation Logic

Frame Rule

- A similar sound rule is introduced for separating conjunction, called frame rule

$$\frac{\{p\}c\{q\}}{\{p * r\}c\{q * r\}}$$

- The frame rule allows for local and global reasoning: allows to talk about only the part of the heap which is used.

Separation Logic

(Local) Inference Rules

- Mutation

$$\frac{}{\{\exists z.e \mapsto z\}[e] := e'\{e \mapsto e'\}}$$

Separation Logic

(Local) Inference Rules

- Mutation

$$\frac{}{\{\exists z. e \mapsto z\}[e] := e'\{e \mapsto e'\}}$$

- Deallocation

$$\frac{}{\{\exists z. e \mapsto z\}\ \mathbf{dispose}\ e\ \{\mathbf{emp}\}}$$

Separation Logic

(Local) Inference Rules

- Mutation

$$\frac{}{\{\exists z. e \mapsto z\}[e] := e'\{e \mapsto e'\}}$$

- Deallocation

$$\frac{}{\{\exists z. e \mapsto z\}\ \textbf{dispose}\ e\ \{\textbf{emp}\}}$$

- Allocation

$$\frac{}{\{\textbf{emp}\}\ v := \textbf{cons}\ e_0, \ldots, e_{n-1}\ \{v \mapsto e_0, \ldots, v + n \mapsto e_{n-1}\}}$$

Separation Logic

(Local) Inference Rules

- Mutation

$$\frac{}{\{\exists z.e \mapsto z\}[e] := e'\{e \mapsto e'\}}$$

- Deallocation

$$\frac{}{\{\exists z.e \mapsto z\}\, \mathbf{dispose}\, e\, \{\mathbf{emp}\}}$$

- Allocation

$$\frac{}{\{\mathbf{emp}\}\ v := \mathbf{cons}\, e_0, \ldots, e_{n-1}\, \{v \mapsto e_0, \ldots, v+n \mapsto e_{n-1}\}}$$

- Lookup

$$\frac{}{\{v = v' \wedge (e \mapsto v'')\}\ v := [e]\, \{v = v'' \wedge (e[v'/v] \mapsto v'')\}}$$

Separation Logic

Inference Rules

- Global and backward rules can be obtained by using the frame rule.

- Global and backward rules can be obtained by using the frame rule.
- The obtained backward reasoning rules give the complete weakest precondition.

- Global and backward rules can be obtained by using the frame rule.
- The obtained backward reasoning rules give the complete weakest precondition.
- Backward rules use separating implication, e.g.

$$\frac{}{\{(\exists z.e \mapsto z) * ((e \mapsto e') \mathrel{-\!\!*} p)\}\ [e] := e'\ \{p\}}$$

- Separation Logic home: `http://www.cs.ucl.ac.uk/staff/p.ohearn/SeparationLogic/Separation_Logic/SL_Home.html`
- Jesper Bengtson on Wednesday, May 16th, 2012 at 10:30 in room 1112 will talk about "Efficient verification of Java-programs using *higher-order separation logic* in Coq".

Questions

?