

# Separation Logic

## A Logic for Reasoning About Shared Mutable Data Structures

Dennis Walter

Chalmers TH  
Dept. of Comp. Sci.

Mar 30, 2006

# Motivating Example

## List Reversal

```
 $j := \text{nil}$   
while  $i \neq \text{nil}$  do  
   $k := [i + 1]$   
   $[i + 1] := j$   
   $j := i$   
   $i := k$   
end
```

- In-place list reversal: Simple to dance, hard to verify
- Standard Hoare logic inappropriate due to **frame problem** (among others)
- Desire for *local reasoning* where one can concentrate on *footprint* of algorithm

# Example Specification

## Loop Invariant: Problems

- Say that lists represented by  $i, j$  do not share anything
- Requires reachability predicates in FOL
- Other structures (e.g. some list  $k$ ) should not be affected

## Loop Invariant as Sep. Logic Formula

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

# Separation Logic in a Nutshell

- Used in the context of Hoare logics (so far)
- New logical operator \* separating heap portions
- Imperative language with storage (de)allocation
- $States = Heaps \times Stores$
- Address arithmetic + Memory faults
- Tight specifications

# Outline

- 1 **Syntax and Semantics**
  - Programming Language
  - Assertions
- 2 **Specifications**
  - Small Specifications
  - Specifying Datatypes
  - Example Proofs
- 3 **Information Hiding**
  - Module Specifications

# Additional Language Constructs

## Heap-altering commands

Allocation:

$x := \mathbf{cons}(E_1, \dots, E_n)$

Mutation:

$[E] := E'$

Deallocation:

$\mathbf{dispose } E$

Lookup:

$x := [E]$

- Allocation possible for all  $n > 0$
- All operations are *commands*
- Expressions cannot refer to heap
- **dispose** deallocates only one cell

# Values, Locations, States

$Values \stackrel{\text{def}}{=} \{\dots, -1, 0, 1, \dots\}$

$Loc \stackrel{\text{def}}{=} \{1, 2, \dots\}$

$Stores \stackrel{\text{def}}{=} Var \rightarrow Values$

$Heaps \stackrel{\text{def}}{=} Loc \rightarrow_{\text{fin}} Values$

$States \stackrel{\text{def}}{=} Stores \times Heaps$

- Locations  $Loc$  subset of  $Values$
- Heaps always finite
- Programs live on  $States$

# Structural Operational Semantics (1)

$$\langle \text{"x := cons}(E_1, \dots, E_n)\text{"}, (s, h) \rangle \longrightarrow \\ \langle (s[x : \xi], h[\xi : [E_1]s, \dots, \xi + (n - 1) : [E_n]s]) \rangle \\ (\xi \notin \text{dom } h)$$

$$\langle \text{"dispose } E'\text{"}, (s, h) \rangle \longrightarrow \langle s, h \upharpoonright \text{dom } h - [E]s \rangle \\ ([E]s \in \text{dom } h)$$

$$\langle \text{"dispose } E'\text{"}, (s, h) \rangle \longrightarrow \text{abort} \\ ([E]s \notin \text{dom } h)$$

## Structural Operational Semantics (2)

$$\langle \text{"x := [E]"}, (s, h) \rangle \longrightarrow \langle (s[x : h([E]s)], h) \rangle$$

$([E]s \in \text{dom } h)$

$$\langle \text{"x := [E]"}, (s, h) \rangle \longrightarrow \mathbf{abort}$$

$([E]s \notin \text{dom } h)$

$$\langle \text{"[E] := E'_1"}, (s, h) \rangle \longrightarrow \langle s, h[[E]s : [E'_1]s] \rangle$$

$([E]s \in \text{dom } h)$

$$\langle \text{"[E] := E'_1"}, (s, h) \rangle \longrightarrow \mathbf{abort}$$

$([E]s \notin \text{dom } h)$

# Separation Logic Primitives

Separation Logic: FOL + Heap assertions

$P ::= \dots$

<b>emp</b>	empty heap
$E \mapsto F$	single heap cell
$P * Q$	separating conjunction
$P \multimap Q$	“magic wand”

## Syntactic Sugar

- $x \mapsto - \stackrel{\text{def}}{=} \exists n. x \mapsto n$
- $x \mapsto a, b \stackrel{\text{def}}{=} x \mapsto a * x + 1 \mapsto b$

# Semantics of Assertions

- Conventional connectives/predicates interpreted through *Store* component
- Separating conjunction: “possible to split heap in two parts, each of which validates one subformula”

$$(s, h) \models P * Q \quad \text{iff} \quad \exists h_1 h_2. (s, h_1) \models P \wedge (s, h_2) \models Q \\ \wedge h = h_1 * h_2$$

- Empty heap

$$(s, h) \models \mathbf{emp} \quad \text{iff} \quad \text{dom } h = \{\}$$

- Heap pointer

$$(s, h) \models E \mapsto E_1 \quad \text{iff} \quad \text{dom } h = \{[E]s\} \wedge h([E]s) = [E_1]s$$

## Examples

Let  $s$  be given and  $h_1 = \{(s\ x) : 1\}$  and  $h_2 = \{(s\ y) : 2\}$  be domain disjoint singleton heaps. Then:

If $p$ is	$(s, h) \models p$ means
$x \mapsto 1 * y \mapsto 2$	$h = h_1 * h_2$
$x \mapsto 1 * x \mapsto 1$	<i>false</i>
$x \mapsto 1 * (x \mapsto 1 \vee y \mapsto 2)$	$h = h_1 * h_2$
$x \mapsto 1 \vee y \mapsto 2$	$h = h_1$ or $h = h_2$
$x \mapsto 1 * \neg x \mapsto 1$	$h_1 \subseteq h$

For  $(s, h) \models x \mapsto 1, y \wedge y \mapsto 1, x$  to hold we need

- $s\ x = s\ y$
- $h = \{s\ x : 1, s\ (x + 1) : s\ x\}$

# Hoare-style Specifications for the Imperative Language

## The Root of All Evil<sup>TM</sup>

- In address manipulating programs, the *Rule of Constancy* fails:

$$\frac{\{P\} C \{Q\}}{\{P \wedge R\} C \{Q \wedge R\}} \quad (\text{given } \text{mod}(C) \cap \text{fv}(R) = \{\})$$

- Aliasing or address arithmetic breaks down modularity

## Solution: *Frame Rule*

$$\frac{\{P\} C \{Q\}}{\{P * R\} C \{Q * R\}} \quad (\text{given } \text{mod}(C) \cap \text{fv}(R) = \{\})$$

# Interpretation of Hoare Triples

- In this talk: only partial correctness considered
- Specification  $\{P\} C \{Q\}$  interpreted *tightly*.  
For all states  $(s, h)$  with  $(s, h) \models P$ 
  - not  $\langle "C", (s, h) \rangle \longrightarrow^* \mathbf{abort}$
  - if  $\langle "C", (s, h) \rangle \longrightarrow^* \langle (s_1, h_1) \rangle$ , then  $(s_1, h_1) \models Q$
  - *well-specified programs don't go wrong*
- Implicit quantification over all executions
- Rules for conventional commands unmodified  
In particular: Existential Introduction and Substitution
- Recall: Expressions heap-independent

# Small Specifications

- Axiomatically, only *footprint* versions of specifications given
- Rules for larger context derivable with Frame Rule

$$\{\mathbf{emp} \wedge x = x'\} x := \mathbf{cons}(E) \{x \mapsto E[x'/x]\}$$

$$\{E \mapsto -\} \mathbf{dispose} E \{\mathbf{emp}\}$$

$$\{x = x' \wedge E \mapsto v_e\} x := [E] \{x = v_e \wedge E[x'/x] \mapsto v_e\}$$

$$\{E \mapsto -\} [E] := E_1 \{E \mapsto E_1\}$$

# Specifying Datatypes

- Pointer structures usually represent elements of some useful data type
- Predicates needed to correlate pointers and abstract value they represent
- Standard examples: lists, doubly-linked lists, XOR-lists, trees, DAGs
- Predicates definable by structural induction on data type

# Examples (1)

- Singly-linked lists:

$$\mathit{list} \ \epsilon \ l \quad \stackrel{\text{def}}{=} \quad \mathbf{emp} \wedge l = \mathbf{nil}$$

$$\mathit{list} \ a \ \alpha \ l \quad \stackrel{\text{def}}{=} \quad \exists k. l \mapsto a, k * \mathit{list} \ \alpha \ k$$

- Binary trees:

$$\mathit{tree} \ a \ t \quad \stackrel{\text{def}}{=} \quad \mathbf{emp} \wedge t = a$$

$$\mathit{tree} \ (\tau_1 \tau_2) \ t \quad \stackrel{\text{def}}{=} \quad \exists t_1 \ t_2. t \mapsto t_1, t_2 * \mathit{tree} \ \tau_1 \ t_1 * \mathit{tree} \ \tau_2 \ t_2$$

## Examples (2)

Note the slightly different definition for Directed Acyclic Graphs (DAGs):

$$\text{dag } a \ d \stackrel{\text{def}}{=} \quad d = a$$

$$\text{dag } (\tau_1 \tau_2) \ d \stackrel{\text{def}}{=} \quad \exists d_1 \ d_2. \ d \mapsto d_1, d_2 * (\text{dag } \tau_1 \ d_1 \wedge \text{dag } \tau_2 \ d_2)$$

The use of  $\wedge$  instead of  $*$  allows for sharing between  $\tau_1$  and  $\tau_2$

# Adding an Element in Front of a List

 $\{list \ \alpha \ i\}$ (def. of *list*) $\{\exists i'. i = i' \wedge list \ \alpha \ i'\}$ 

(ex. intro.)

 $\{i = i' \wedge \mathbf{emp} * list \ \alpha \ i'\}$ 

(frame)

 $\{i = i' \wedge \mathbf{emp}\}$ 

(allocation axiom)

 $i := \mathbf{cons}(0, i)$  $\{i \mapsto 0, i'\}$  $\{i \mapsto 0, i' * list \ \alpha \ i'\}$  $\{\exists i'. i \mapsto 0, i' * list \ \alpha \ i'\}$  $\{list \ 0 \ \alpha \ i\}$

# Separation and Information Hiding (POPL'04)

One can do more:

- Add procedures (for simplicity non-recursive here)
- Realize *modules* by hiding internal data representation from client programs
- Elegant ownership transfer of heap portions from module to client and back
- “Ownership is in the eye of the asserter”  
(not necessarily in the PL)

# Modular Procedure Declaration Rule (0)

Add local **procedure declarations**

$$\begin{array}{c} \Gamma \vdash \{P_1 * R\} C_1 \{Q_1 * R\} \\ \vdots \\ \Gamma \vdash \{P_n * R\} C_n \{Q_n * R\} \\ \Gamma, \{P_i\} k_i \{Q_i\} [X_i] \text{ (for } i \leq n) \vdash \{P\} C \{Q\} \\ \hline \Gamma \vdash \{P * R\} \text{let } k_i = C_i \text{ in } C \{Q * R\} \end{array}$$

# Modular Procedure Declaration Rule (1)

Introducing **assumptions**  $\Gamma$  which is just a list of procedure specifications

$$\begin{array}{c} \Gamma \vdash \{P_1 * R\} C_1 \{Q_1 * R\} \\ \vdots \\ \Gamma \vdash \{P_n * R\} C_n \{Q_n * R\} \\ \Gamma, \{P_i\} k_i \{Q_i\} [X_i] \text{ (for } i \leq n) \vdash \{P\} C \{Q\} \\ \hline \Gamma \vdash \{P * R\} \text{ let } k_i = C_i \text{ in } C \{Q * R\} \end{array}$$

# Modular Procedure Declaration Rule (2)

Further: **Resource Invariant**  $R$  shared by procedures  $k_i$ ,  
but invisible to 'client'  $C$

$$\begin{array}{c} \Gamma \vdash \{P_1 * R\} C_1 \{Q_1 * R\} \\ \vdots \\ \Gamma \vdash \{P_n * R\} C_n \{Q_n * R\} \\ \Gamma, \{P_i\} k_i \{Q_i\} [X_i] \text{ (for } i \leq n) \vdash \{P\} C \{Q\} \\ \hline \Gamma \vdash \{P * R\} \text{ let } k_i = C_i \text{ in } C \{Q * R\} \end{array}$$

# Modular Procedure Declaration Rule (3)

**Modifies clauses** for each module procedure  
indicate what portion of the *store* may change on invocation

$$\begin{array}{c} \Gamma \vdash \{P_1 * R\} C_1 \{Q_1 * R\} \\ \vdots \\ \Gamma \vdash \{P_n * R\} C_n \{Q_n * R\} \\ \Gamma, \{P_i\} k_i \{Q_i\} [X_i] \text{ (for } i \leq n) \vdash \{P\} C \{Q\} \end{array}$$

---

$$\Gamma \vdash \{P * R\} \text{ let } k_i = C_i \text{ in } C \{Q * R\}$$

# Modules?

## A simple memory management module

- Interfaces:

$$\{\mathbf{emp}\} \text{ alloc } \{x \mapsto -, -\}[x]$$
$$\{x \mapsto -, -\} \text{ free } \{\mathbf{emp}\}$$

- Resource invariant:  $R = \text{list } f$  (ignoring contents of list)
- Let `alloc` be  
*if*  $f = \text{nil}$  *then*  $x := \mathbf{cons}(-, -)$   
*else*  $x := f; f := [x + 1]$
- `free`:  $[x + 1] := f; f := x;$

# How Does That Work? Module Perspective

In `free` resource invariant 'devours' current `x` cons cell:

$$\{list\ f * (x \mapsto -, -)\}$$
$$[x + 1] := f;$$
$$\{list\ f * (x \mapsto -, f)\}$$
$$\{list\ x\}$$
$$f := x$$
$$\{list\ f\}$$
$$\{list\ f * \mathbf{emp}\}$$

# Client's Perspective on a Module

Deletion of an element in a list (wrong implementation)

$$\{(y \mapsto 1, x) * (x \mapsto 2, z) * (z \mapsto 3, w)\}$$

*free*

$$\{(y \mapsto 1, x) * (z \mapsto 3, w)\}$$

*// Client has given up ownership of cell x*

$$y := [x + 1]$$

*{no postcondition can be verified}*

Of course, it is possible to verify the correct implementation

$$y := [x + 1]; \textit{free } x$$

# The Eye of The Observer

- For the memory manager, cons cells are handed in and out of the module
- In general, module specifications can be generic in the amount of rights they convey to clients
- The implementation does not know about information hiding in any way
- Only the specification defines rights towards / ownership of data

# Current and Future Directions

- Separation Logic in Abstract Interpretation (e.g. Byron Cook et al.)
- Shape analysis
- Symbolic execution and automated theorem proving
- Concurrency
- Separation Logic and Types
- ...

**THANK YOU**