Logics for Dynamic Data

Uday S. Reddy

University of Birmingham
Logics of Programs

- Very active research area in the 70's.
- Meant for proving "correctness" of programs.
- Logical principles.
- Techniques for reasoning.
- Automated verification.

Counterpoint in the 80's & 90's.

- Can one formalize "correctness"?
- Are proofs fool-proof?
- Can programmers do reasoning?
- Automated verification is hard.
Consensus

- Good programmers should know the logical principles of programming.
- Logical techniques can also be used for checking "safety."
- Automated verification

Outstanding problems

- Dynamic data structures
- Object-oriented features
  (higher-order, inheritance)
- Concurrency
Separation Logic

A logic for reasoning about "dynamic resources"

Resources:
Entities with a "physical" existence cannot be duplicated like "mathematical" entities (numbers, sets, functions...)

Examples:
* Memory cells
  * OS resources
  * Connection pools

"store" "heap"
Dynamic Memory

- The structure of the storage used by the program dynamically varies.

- When you change a heap cell, you are also changing the data structure that it is part of.

- There may be cells shared between different data structures.

(Sharing is really an exception, but the potential for sharing messes up formal reasoning.)
• When you dispose a cell, there may still be pointers to it.

"Dangling pointers"

• One may forget to dispose a cell before removing all pointers to it.

"Space leaks"

• Data encapsulation may be violated by pointers into a module's storage.

"Information leakage"

• One needs to be able to transfer cells into a module's storage and out of it.

"Ownership transfer"
Formal reasoning

It is hard to get pointer programs right. However, formal reasoning for heap has been traditionally cumbersome.

Hoare logic for local variables:

\[
\begin{array}{c}
\text{\{p(e)\}} \\
\text{x := e} \\
\text{\{p(x)\}}
\end{array}
\]

only x involved in reasoning

"separate" variables

Hoare logic with heap:

- Treats heap as a single data structure (e.g., as an array)
- Updates to any heap variable becomes an update of the global heap.

Modularity? Separation?
Historical development of Separation Logic

1978 Reynolds: Syntactic control of Interference

1994 Reddy: Passivity and Independence

1995 O'Hearn et al.: Syntactic control of Interference Revisited

1998 O'Hearn: Bunched Type System

1999 Pym & O'Hearn: The Logic of Bunched Implications

1972 Bunstall:
"Some techniques for proving correctness of programs that alter data structures"

2000 Reynolds:
"Intuitionistic reasoning about shared mutable data structure"

2001 Ishbiaq & O'Hearn: BI as an assertion logic
Traditional Hoare Logic

\[ \text{Store} = \text{Van} \rightarrow \text{Val} \]
\[ \text{comm} \subseteq \text{store} \times \text{store} \]
\[ \text{Assertions} = \text{first-order logic formulas} \]
\[ s \models P \]
\[ \text{store} \quad \text{formula} \]
\[ \{ P \} C \{ Q \} \]

Separation Logic

\[ \text{Store} = \text{Van} \rightarrow \text{Val} \]
\[ \text{Heap} = \text{Addr} \rightarrow \text{Val} \times \text{Val} \]
\[ \text{State} = \text{store} \times \text{Heap} \]
\[ \text{comm} \subseteq \text{store} \times (\text{State} + \text{State}) \]
\[ \text{Assertions} = \text{Bunched Implication formulas} \]
\[ s, h \models P \]
\[ \text{store} \quad \text{heap} \quad \text{formula} \]
The Assertion Language

\[ P ::= \ldots \mid e, \rightarrow e_2, e_2' \mid P_1 \land P_2 \mid \text{emp} \]

\[ e, \rightarrow e_2, e_2' \text{ describes the heap} \]

\[ e_1 \quad \begin{array}{c}
\vrule \\
\downarrow \\
\vrule \quad e_2 \\
\vrule \end{array} \quad e_2' \]

\[ P_1 \land P_2 \text{ describes the heap} \]

\[ \text{emp} \text{ describes the empty heap} \]

\[ (s, h) \models e, \rightarrow e_2, e_2' \iff \]

\[ \text{dom}(h) = \{[e_1]s\} \land \]

\[ h([e_2]s) = ([e_2]s, [e_2']s) \]

\[ (s, h) \models P_1 \land P_2 \iff \]

\[ \exists h_1, h_2. \quad h = h_1 \land h_2 \land \]

\[ (s, h_1) \models P_1 \]

\[ (s, h_2) \models P_2 \]

\[ (s, h) \models \text{emp} \iff \text{dom}(h) = \emptyset \]
classical connectives are as expected:

\[(s,h) \models P_1 \land P_2 \iff (s,h) \models P_1 \text{ and } (s,h) \models P_2\]

\[(s,h) \models \text{true} \text{ always}\]

\[(s,h) \models \text{false} \text{ never}\]

Spatial implication connective:

\[(s,h) \models P \rightarrow Q \iff \forall h'. \text{ dom}(h) \cap \text{ dom}(h') = \emptyset \land (s,h') \models P \Rightarrow (s, h \cup h') \models Q\]

\[\{P \rightarrow Q\} \star P \rightarrow Q.\]
Examples:

\( x \mapsto 3, y \)  
state: \( x: a, y: b \)  
heap: \( a: (3, b) \)

\( x \mapsto 3, y \)  
\( y \mapsto z \)  
\( z \leftarrow 3 \rightarrow 7 \rightarrow \)  
\( x \mapsto 3, y \)  
\( y \mapsto 7, x \)  
\( x \mapsto 3, y \)  
\( x \mapsto 7, z \)  
not satisfiable

\( x \mapsto 3, y \)  
\( x \mapsto 3, y \)  
not satisfiable

\( x \mapsto 3, y \)  
\( x \mapsto 3, z \)  
\( z \leftarrow 3 \rightarrow \)  
\( x \mapsto 3, y \)  
\( x \mapsto 3, y \)  
emp  
not satisfiable
\[ x \rightarrow 3, y \rightarrow \text{true} \quad \text{means} \quad x \rightarrow 3, y \]

\[ x \rightarrow 3, y \rightarrow \text{true} \]

**"Intuitionistic" formulas**

Formulas which, if true for a heap, are also true for all extensions of that heap.

\[ P \times \text{true is intuitionistic for any } P. \]

**Some facts:**

\[ P \times \text{emp} = P \]

\[ P \times (Q \land R) \supseteq (P \times Q) \land (P \times R) \]

\[ (\exists x. P) \times Q = \exists x. (P \times Q) \]

\[ P \times (Q \lor R) = (P \times Q) \lor (P \times R) \]

\[ P \times \text{false} = \text{false} \]
Recursive data structures

\[ \text{list} \in l \]

\[ l \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \]

\[ \text{list} \in l \iff l = \text{nil} \lor \text{emp} \]

\[ \text{list} (a \eta') l \iff \exists l'. l \rightarrow a, l' \ast \]

\[ \text{listseg} \in a \ast l \ m \]

\[ l \rightarrow a_1 \rightarrow \ldots \rightarrow a_n \rightarrow \]

Notation: \[ l \xrightarrow{a} m \]

\[ l \xrightarrow{5} m \iff l = m \lor \text{emp} \]

\[ l \xrightarrow{\eta'} m \iff \exists l'. l \xrightarrow{l'} \ast \rightarrow l' \xrightarrow{\eta'} m \]

\[ l \xrightarrow{a} l' \ast \xrightarrow{\eta'} m \]

Some facts:

\[ l \xrightarrow{a} m \ast m \xrightarrow{\beta} n \Rightarrow l \xrightarrow{a} n \]

\[ l \xrightarrow{a} m \land (l \xrightarrow{a} k \ast k \xrightarrow{5} m) \Rightarrow a = \beta \]
Local reasoning / tight interpretation

\[ \{ P \} c \{ Q \} \iff \forall (s, h). (s, h) \vdash P \Rightarrow \\
\neg (s, h) \models [c] \text{ abort} \land \forall (s', h'). (s, h) \models [c] (s', h') \Rightarrow (s', h') \not\models Q. \]

If \( P \) is true initially, then
1. The command runs without errors,
2. If it terminates, gives a state satisfying \( Q \).

**Consequence**

The command can only access the heap cells described by \( P \).

All other parts of the heap will be untouched !!!
Hoare logic specifications describe what a command does, not what it doesn't do.

Separation logic specifications describe precisely what a command does (as far as heap cells go).

**Hoare Logic:**

\[ \{ P \} \vdash \{ \text{true} \} \]
always valid.

**Separation Logic:**

\[ \{ P \} \vdash \{ \text{true} \} \]
means \( C \) is safe.
(touches only the cells described by \( P \)).

**Frame Rule**

\[
\frac{\{ P \} \vdash \{ Q \}}{\{ P \times R \} \vdash \{ Q \times R \}}
\]
if \( C \) does not modify the free variables of \( R \).

Note: The rule would be unsound if we replace \( \times \) by \( \land \).
The specification only needs to describe the cells touched by C. The "entire heap" is not involved in the specification.
Rules for commands

Expressions

\[ E ::= \ldots \quad (\text{traditional}) \]

Commands

\[ C ::= x := E \mid \ldots \]

\[ x := \text{cons} (E_1, E_2) \quad \text{allocate} \]

\[ \text{dispose} (E) \quad \text{deallocat} \]

\[ x := E.1 \quad \text{dereference} \]

\[ E.1 := e' \quad \text{mutate} \]

\{ emp \} \quad \{ e \rightarrow \ldots \}

\[ x := \text{cons} (e_1, e_2) \quad \text{dispose} (e) \quad \{ emp \} \]

\[ \{ e \rightarrow e_1, e_2 \} \quad \{ e \rightarrow - , e_1 \} \]

\[ x := e.1 \quad e.1 := e' \]

\[ \{ e \rightarrow e_1, e_2 \} \quad \{ e \rightarrow e', e_2 \} \]

"-" stands for an existentially quantified variable ("don't care" variable)