Advances in Programming Languages

Lecture 14: Separation Logic

Ian Stark

School of Informatics
The University of Edinburgh

Monday 5 November 2018
Semester 1 Week 8

THE UNIVERSITY *of* EDINBURGH

https://wp.inf.ed.ac.uk/apl18
https://course.inf.ed.ac.uk/apl

Outline

Outline

Topic: Augmented Languages for Correctness and Certification

The next block of lectures cover some language techniques and tools for improving program correctness:

- Specification and Verification
- Practical Java tools for Correctness
- Separation Logic
- Augmented Programming and Certifying Correctness

The focus here is not necessarily on changing what a program does, or making it do that thing faster, or using less memory, or less power. Instead we want to make sure that what it does is the right thing to do.

This lecture introduces an enrichment of Hoare Logic that supports local reasoning about code that works with linked-pointer data structures in memory.

Homework from Monday

JML is just one of many frameworks for specifying and verifying code in various programming languages.

For example: Java annotations, FindBugs™, QuickCheck, the C specification language ACSL and Frama-C platform, Spec#, ...

Your homework is to use a small part of one such system.

1. Read This

Find out about *Type Annotations and Pluggable Type Systems* by reading the Oracle Java Tutorial lesson with that name.

2. Do this

Try out the @NonNull and @Nullable annotations on the *Checker Framework* live demo:

- **http://eisop.uwaterloo.ca/live**

Outline

Plan of Work

- The store/heap model for linked datastructures
- Decomposing a linked heap into heaplets
- Assertions about the store and heap
- New operations: star $*$ and magic wand $-\!\!*$
- Representing datastructures with this logic

Store

A Hoare triple $\{P\}$ C $\{Q\}$ involves assertions P and Q which talk about the *state* of an imperative program.

$$\{a > 3\}\ \ b := a+a\ \ \{b > 6\}$$

$$\{(d > z)\ \wedge\ (d' > z)\ \wedge\ (z \geqslant 0)\}\ \ c := d*d'\ \ \{c > z^2\}$$

$$\{\mathbf{true}\}\ \text{while i>0 do i := i-1}\ \ \{i \leqslant 0\}$$

In these examples the state is the values of the variables mentioned — a, b, c, d, d' and i. This is usually described as an *environment* or *store*, formalised as a mapping s from variable names to stored values.

$$s \in \mathrm{Store} \stackrel{\mathrm{def}}{=} \mathrm{VarName} \to \mathrm{Int}$$

Heap

Most imperative languages also provide larger and more complex datastructures that an a simple store of variables. Things like objects, linked lists, trees, or graphs are typically implemented using a *heap*: an area of memory over which either the programmer or the language runtime has very flexible control. The following lists some distinctive characteristics of the heap.

- The heap contains cells that can hold values.
- Each cell has an index or address that uniquely identifies its location in the heap.
- Cells in consecutive locations may be grouped into structured records.
- Cells may contain index values (*pointers*) that identify other cells in the heap.
- Variables in the store can also contain pointers.
- Records and pointers can be used to build complex linked datastructures.
- Portions of the heap can be arbitrarily allocated and deallocated as a program executes.

Modelling the Heap

To model the behaviour of a heap, we can combine the store $\mathtt{s}$ with another partial map $\mathtt{h}$ recording which heap cells are allocated and what values they hold. For simplicity, we'll assume cells have integer addresses and contain integer values.

$$\begin{aligned} \mathtt{s} \in \mathtt{Store} &\stackrel{\mathtt{def}}{=} \mathtt{VarName} \to \mathtt{Int} \\ \mathtt{h} \in \mathtt{Heap} &\stackrel{\mathtt{def}}{=} \mathtt{Int} \rightharpoonup \mathtt{Int} \\ (\mathtt{s}, \mathtt{h}) \in \mathtt{State} &\stackrel{\mathtt{def}}{=} \mathtt{Store} \times \mathtt{Heap} \end{aligned}$$

The *domain* of the map $\mathtt{h}$, written $\mathrm{dom}(\mathtt{h})$, indicates which heap cells have been allocated. It's possible to have *dangling pointers* in both the heap and the store, where an address points to a heap cell that's not currently in use.

Operations on Heaps and Heaplets

A heap is partial, as only some cells will be allocated, and may include dangling pointers.

$$\mathtt{h} \in \mathtt{Heap} \stackrel{\mathtt{def}}{=} \mathtt{Int} \rightharpoonup \mathtt{Int}$$

Two heaps $\mathtt{h}_1$ and $\mathtt{h}_2$ are *disjoint* $\mathtt{h}_1 \perp \mathtt{h}_2$ if they allocate non-overlapping sets of cells. In this case we can combine them to get their *union* $\mathtt{h}_1 \cdot \mathtt{h}_2$.

$$\mathtt{h}_1 \perp \mathtt{h}_2 \quad \stackrel{\mathtt{def}}{\iff} \quad \mathsf{dom}(\mathtt{h}_1) \cap \mathsf{dom}(\mathtt{h}_2) = \emptyset$$

$$\mathtt{h}_1 \cdot \mathtt{h}_2 \quad \stackrel{\mathtt{def}}{=} \quad \mathtt{h}_1 \cup \mathtt{h}_2 \text{ provided that } \mathtt{h}_1 \perp \mathtt{h}_2$$

Going in the other direction, it's possible to separate a larger heap $\mathtt{h}$ into smaller *heaplets* $\mathtt{h} = \mathtt{h}_1 \cdot \mathtt{h}_2$, usually in many different ways. Note that the heaplets $\mathtt{h}_1$ and $\mathtt{h}_2$ may have dangling pointers crossing between them.

Assertions about Heaps

Empty	$\mathbf{emp}$	The heap is empty, no cells are allocated.
Singleton	$e \mapsto e'$	The heap has a single allocated cell, at address e, containing value e'.
Separating Conjunction "Star"	$\mathrm{P} * \mathrm{P}'$	The heap can be split into two disjoint heaplets, with one satisfying P and the other satisfying P'.
Separating Implication "Magic Wand"	$\mathrm{P} \mathrel{-\!\!*} \mathrm{Q}$	If the heap is combined with any disjoint heap h' that satisfies P, then the combined heap will satisfy Q.

Derived Assertions about Heaps

$$e \hookrightarrow e' \quad \stackrel{\text{def}}{\iff} \quad e \mapsto e' * \mathbf{true}$$

$$e \mapsto - \quad \stackrel{\text{def}}{\iff} \quad \exists x \,.\, e \mapsto x$$

$$e \hookrightarrow - \quad \stackrel{\text{def}}{\iff} \quad (e \mapsto -) * \mathbf{true}$$

$$e \mapsto e_1, \ldots, e_n \quad \stackrel{\text{def}}{\iff} \quad (e \mapsto e_1) * ((e+1) \mapsto e_2) \cdots * ((e+(n-1)) \mapsto e_n)$$

$$e \hookrightarrow e_1, \ldots, e_n \quad \stackrel{\text{def}}{\iff} \quad (e \mapsto e_1 \ldots, e_n) * \mathbf{true}$$

Datastructures in Separation Logic

The language of separation logic makes it possible to describe properties of datastructures built using pointers in the heap: tuples, records, lists, trees, graphs, objects, ...

Lists

Assertion $\mathtt{list}\ xs\ e$ describes a heap made of a linked list that starts at the cell pointed to by e and contains exactly the values in xs.

$$\begin{array}{rcl} \mathtt{list}\ []\ \mathsf{i} & \stackrel{\mathsf{def}}{=} & \mathbf{emp} \wedge (\mathsf{i} = 0) \\ \mathtt{list}\ (\mathsf{x}::xs)\ \mathsf{i} & \stackrel{\mathsf{def}}{=} & \exists \mathsf{j}\ .\ (\mathsf{i} \mapsto \mathsf{x}, \mathsf{j}) \ * \ (\mathtt{list}\ xs\ \mathsf{j}) \end{array}$$

The use of separating conjunction "$*$" here means that every node in the list uses a distinct part of the heap. The end of the list is signalled by a null pointer (value 0).

Datastructures in Separation Logic

The language of separation logic makes it possible to describe properties of datastructures built using pointers in the heap: tuples, records, lists, trees, graphs, objects, ...

Binary Trees

Assertion $\mathrm{tree}\ \mathrm{t}\ e$ describes a heap containing a binary tree with structure matching t and root at the cell pointed to by e.

$$\begin{array}{rcl} \mathrm{tree}\ (\mathrm{Leaf}\ \mathrm{x})\ \mathrm{i} & \stackrel{\mathrm{def}}{=} & \mathrm{i} \mapsto 0, \mathrm{x} \\ \mathrm{tree}\ (\mathrm{Branch}\ (\mathrm{s}, \mathrm{t}))\ \mathrm{i} & \stackrel{\mathrm{def}}{=} & \exists \mathrm{j}, \mathrm{k}\ .\ (\mathrm{i} \mapsto 1, \mathrm{j}, \mathrm{k})\ *\ (\mathrm{tree}\ \mathrm{s}\ \mathrm{j})\ *\ (\mathrm{tree}\ \mathrm{t}\ \mathrm{k}) \end{array}$$

Tags 0 and 1 distinguish between leaves and internal nodes. Again, the separating conjunction "$*$" means that the tree really is a tree: none of the subtrees overlap or share nodes.

Outline

Triples

In separation logic, a triple $\{P\}$ C $\{Q\}$ now has assertions P and Q that make statements about the whole state (s, h) not just the store s.

Assertions that don't depend on the heap are *pure* assertions; those that do depend on the contents of the heap are *spatial* assertions.

The same rules of inference apply as before, together with additional rules for the new heap operations. For example, the star and magic wand operators satisfy the following:

$$\frac{s, h \vdash P * (P \mathrel{-\!\!*} Q)}{s, h \vdash Q}$$

Heap Inference Rules

Any language operation relating to the heap gets an appropriate inference rule.

$$\frac{}{\{\mathbf{emp}\}\ \mathsf{v} := \mathsf{alloc}(\mathsf{e}_1,\ldots,\mathsf{e}_n)\ \{\mathsf{v} \mapsto \mathsf{e}_1, \ldots, \mathsf{e}_n\}}$$

$$\frac{}{\{(\mathsf{e} \mapsto -)\}\ [\mathsf{e}] := \mathsf{e}'\ \{\mathsf{e} \mapsto \mathsf{e}'\}}$$

$$\frac{}{\{\mathsf{e} \mapsto -\}\ \mathsf{dispose}\ \mathsf{e}\ \{\mathbf{emp}\}}$$

These can also be elaborated to account for the remaining heap.

$$\frac{}{\{(\mathsf{e} \mapsto -) * \mathsf{r}\}\ [\mathsf{e}] := \mathsf{e}'\ \{\mathsf{e} \mapsto \mathsf{e}' * \mathsf{r}\}}$$

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

Frame Rule

Crucially, separation logic makes it possible to reason *compositionally*, where *local reasoning* about use of the heap can be extended to reasoning about a complete program. Central to this is the *frame rule*:

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}$$ Where C does not modify any of $\mathrm{dom}(R)$.

This *modularity* is one thing that makes it feasible to scale and automate separation logic to checking and verifying large programs.

Outline

Applications

Program Analysis

Separation logic has been extensively applied in academic research and industrial practice. As an enrichment of Hoare logic, it's suitable for annotating programs or libraries, specifying desired behaviour, statically checking properties, or proving program correctness.

It's been used to prove correctness of numerous pointer-manipulation algorithms for efficient processing of complex data structures.

The *Space Invader* analyser checks pointer manipulation in C code; Microsoft's *SLAyer* verifies correctness of device drivers; Facebook's *Infer* tool statically checks C, C++, Java and Objective C.

Other Directions

Separation logic has applications to reasoning about concurrent code, where $\mathrm{P} * \mathrm{Q}$ describes properties P and Q of distinct threads. There are higher-order versions of separation logic; *Hoare Type Theory* incorporating these ideas into a type system; and an mathematical *Logic of Bunched Implications* that provided the groundwork for separation logic.

Homework

1. Watch this

Peter O'Hearn: "Continuous Reasoning: Scaling the impact of formal methods"
Plenary talk at the Federated Logic Conference 2018.
Video: https://is.gd/continuousreasoning
More information: https://www.floc2018.org/speaker/peter-ohearn

2. Do this

Complete and submit your written assignment. If you have questions then send me mail or post on Piazza.