

FMSoft

**Lecture 9 — Frame Your Reasoning: Intro
to Shared Mutable Data Structures**

(pre-lecture version)

Tadeusz Litak

December 11, 2018

Informatik 8, FAU Erlangen-Nürnberg

- Hoare reasoning goes funny when playing with **data structures**
- At least static arrays, say
- And there is still much more to life than such arrays ...

- Lists (single- or doubly-linked), queues, trees, stacks ...
- ...and with these, issues of pointers, heaps, allocation ...
- Recall the definition of **shared mutable data structures** we took from John Reynolds: *structures where an updatable field can be referenced from more than one point*
- Clearly, a subject of immense significance
- Also quite clearly, a major source of errors, issues and bugs: dangling pointers, memory leaks, segmentation faults ...

Instructive exercise: revisit one of lists of (in-)famous software bugs I mentioned in October and try to find out how many of them were caused this way

**Shared mutable data structures:
whence the pain?**

- Perhaps unsurprisingly: such programs notoriously difficult to reason about
- Examples taken from Peter O'Hearn presentations: by 2000's, *impressive practical advances in automatic program verification*
 - Microsoft's **SLAM** Protocol (mentioned in Bill Gates' 2002 keynote address): properties of procedure calls in device drivers, e.g. *any call to `ReleaseSpinLock` is preceded by a call to `AcquireSpinLock`*
 - In Nov. 2003, the **Astrée static analyzer** proved *completely automatically the absence of any RTE in the primary flight control software of the Airbus A340 fly-by-wire system*
see the project webpage
- ...but even these projects were steering clear of automatic heap verification!
 - The first assumed **memory safety**
 - The second assumed **no dynamic pointer allocation**

- And why (shared) mutable data structures are so problematic?
- Let us quote from O'Hearn, Reynolds and Yang:
- *The main difficulty is **not** one of finding an in-principle adequate axiomatization of pointer operations ...*
- *rather there is a **mismatch** between simple intuitions about the way that pointer operations work and the **complexity of their axiomatic treatments** ...*
- *For example, pointer assignment is operationally simple, but ...**aliasing** ...may affect the values of many syntactically unrelated expressions ...*

- Consider the following code:

```
PROC appendlist(x,y)
  LOCAL t, u;
  IF (x == nil) THEN x := y ELSE
    t := x; u := t ->n;
    WHILE not (u == nil) DO t := u; u := t ->n END;
    t ->n := y
  ENDIF
ENDPROC
```

- Assume the assertion language contains a predicate $ls(x, t)$ meaning *a linked list segment*:
there is a path from x to t with no points repeating
- A complete linked list: $ls(x, nil)$
- Is this a valid triple?
 $\{ls(x, nil) \text{ and } ls(y, nil)\}$
appendlist(x,y)
 $\{ls(x, nil)\}$

- x cannot be a sublist of y : we have to be able to state that
- but also, y cannot be a sublist of x . Is this enough?
- x and y should not have a common final segment ...
- ...in short, they should live in disjoint areas of memory
- And now think of the loop invariants
- And remember you should not only be able to state all the information, but have some way to reason about, decide and infer such assertions

- And now try to reason about a bigger program using `appendlist`
- It can use many other data structures and areas of the heap
- When `appendlist` is invoked, the Floyd-Hoare reasoning can only use information in the contract
- Pre- and postconditions should allow us to infer that nothing outside the area occupied now by `x` changed ...
- In short, we are staring in the face of ...

The frame problem

- The term **the frame problem** appeared in early days of (the philosophy of) Artificial Intelligence
- McCarthy and Hayes, *Some philosophical problems from the standpoint of Artificial Intelligence*, 1969
- Try to formalize, say, the obvious fact that P can get into conversation with Q by looking up Q's number in the phone book and then dialling it up
- Whatever formalization you are going to write, it is almost certain in the end you will be missing some hypotheses like *if a person has a telephone he still has it after looking up a number in the telephone book ...*
- ...or that *if P looks up Q's phone-number in the book, he will know it*
- And then of course there are all sorts of special-case scenarios you need to exclude. Quoting McCarthy and Hayes still further ...

- *The page with Q's number may be torn out.*
- *P may be blind.*
- *Someone may have deliberately inked out Q's number.*
- *The telephone company may have made the entry incorrectly.*
- *Q may have got the telephone only recently.*
- *The phone system may be out of order.*
- *Q may be incapacitated suddenly ...*

- *When formally describing a change in a system, how do we specify what parts of the state of the system are not affected by that change?*

as paraphrased later by Kassios

- The fact that an analogous problem arises with formal specifications using Floyd-Hoare logics discussed (in the context of OO code: inheritance issues etc.) by Borgida, Mylopoulos and Reiter. **On the frame problem in procedure specifications**. IEEE Transactions of Software Engineering, 1995
- Hoare-style formalisms invented in the noughties for shared mutable data structures use the term *frame* very prominently ...

- In **separation logic** (Reynolds, Ishtiaq, O’Hearn ..., 1999–2002 and developed since), a central Hoare rule (proposed by O’Hearn) is **the frame rule**

It has to be either assumed as a primitive rule or derived.

Separation logic also found to be useful in the context of concurrency and other forms of resource sharing. Extended with *abstract predicates* by Parkinson and Bierman 2005

- An alternative approach proposed by Kassios in 2006: **dynamic frames**. A later relative: **implicit dynamic frames** (Smans, Jacobs, Piessens 2009)

Dynamic frames intended to use also in the OO context. Implicit dynamic frames are closer in spirit to separation logic

- IDF underly the **VeriFast** tool (that you will see in tutorials)

- A central idea of separation logic: suspend thinking of the global heap when writing/reading specifications
- Quoting Berdine, Calcagno, O’Hearn: think of **heaplets**, portions of heap.
- a spec $\{P\} C \{Q\}$ says that *if C is given a heaplet h satisfying P then it will never try to access heap outside of h (other than cells allocated during execution) and it will deliver a heaplet satisfying Q if it terminates*
- (Of course, this has implications for how C acts on the global heap.)

In the dynamic frames approach, one does think in terms of global heap, using instead: “reads/modifies” clauses in assertions, “swinging pivot postconditions” and permission masks

- Assertions about **disjoint heaplets** are combined using the **spatial conjunction** $A_1 * A_2$
a.k.a. the **separating conjunction** or the **independent conjunction**
- Logicians have studied properties of such connectives under the umbrella of **substructural logics**
linear logics, relevant logics, fuzzy logics...
(one can call them **resource-aware logics** and the connective itself often called **fusion**)
Reynolds was rather inspired by an early work of Burstall:
Some techniques for proving correctness of programs which alter data structures, 1972
- The assertional core of separation logic is **(B)BI**: the (boolean) logic of **bunched implications** (O'Hearn, Pym)
The name comes actually from proof theory of relevant logics

- Recall the problem with
 {ls(x, nil) and ls(y, nil)}
 appendlist(x,y)
 {ls(x, nil)}
- This is looking much better:
 {ls(x, nil) * ls(y, nil)}
 appendlist(x,y)
 {ls(x, nil)}

- We can also state the **frame rule**:

$$\frac{\{A\}C\{B\} \quad \text{no variable free in } A' \text{ modified by } C}{\{A * A'\}C\{B * A'\}}$$

- The second premise, of course, needs to be and will be suitably formalized

- BI has also other connectives, like the **the magic wand** \multimap
- It plays the same role with \ast as \rightarrow with \wedge
- It is useful, e.g., for deriving weakest preconditions
recall the role played by the boolean implication in calculating IMP preconditions
- Less broadly accepted than the separating conjunction though
- Time to move on to a formal development: a dynamic variant of IMP (call it DIMP)

The set of dynamic commands **DCom** is defined as

$c_1, c_2 ::= \text{SKIP} \mid X := a \mid c_1 ; c_2 \mid \text{IF } b \text{ THEN } c_1$
 $\text{ELSE } c_2 \text{ ENDIF} \mid \text{WHILE } b \text{ DO } c_1 \text{ END} \mid$

$X := \text{CONS}(a_1, \dots, a_n) \mid X := [a] \mid [a] := a' \mid$
 $\text{DISPOSE } a.$

- We could give a denotational/operational semantics for DIMP in terms of *pairs of* old states (now **stores**) and **heap(let)s** ...
- ...the latter being finite partial maps from \mathbb{N} to \mathbb{Z}
- Immediate complications, e.g., semantic of allocation will be probably non-deterministic (*pick an address not yet on the heaplet*)
- However, before we even start going there, we can try to pursue the program we originally tried with IMP too
- That is, characterize our commands in terms of **axiomatic semantics**

For the time being at least take my word that these commands are sound wrt a suitable semantics—for those interested, there is even a Coq formalization that we may present at SemProg

- The class of extended arithmetical expressions EAExp is defined as before, i.e.,

$$a_1, a_2 ::= i \mid \mathbf{z} \mid \mathbf{X} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \cdot a_2$$

where $i \in \text{LogVar}$ and the rest as in AExp

Note btw that we need to replace the multiplication symbol in assertions, as it would clash with separating conjunction

- The class of **assertions** of (**wandless infinitary**) **separation logic** (wandless dynamic Hoare logic) SepAss^- is defined as

$$B_1, B_2 ::= \top \mid a_1 == a_2 \mid a_1 < a_2 \mid \neg B_1 \mid \bigwedge_{z \in \mathbb{Z}} B_z \mid$$

$$\text{emp} \mid a_1 :-> a_2 \mid B_1 * B_2.$$

- We can repeat previous comments about interchangeability of notation and definability of

$$\perp, \quad a_1 <= a_2, \quad a_1 >= a_2, \quad B_1 \vee B_2, \quad B_1 \rightarrow B_2,$$

$$\forall i. B(i), \quad \exists i. B(i), \quad \bigwedge_{n \in \mathbb{N}} B_n, \quad \bigvee_{n \in \mathbb{N}} B_n \dots$$

- Note: the assertion language already is in some ways richer than that of VeriFast and Implicit Dynamic Frames ...
e.g., presence of infinitary connectives
- ...and we will still enrich it further!
- But what are $\{A\}c\{B\}$ supposed to mean now?
- Recall that our new notion of state combines **stores** (old states) $\sigma : \text{ProgVar} \rightarrow \mathbb{Z}$ and **heap(lets)** $h : \mathbb{N} \rightarrow \mathbb{Z}$
- The old assertions, involving just arithmetical expressions and program variables, depend only on the store (no rôle played by the heaplet)
- We need to explain how to interpret these involving **emp**, **$:->$** and **$*$** .

- First, arithmetical expressions (used for heap addressing) take values in \mathbb{Z} , but addresses themselves can be only in \mathbb{N}
- Not a problem: we can (and we will) use negative “addresses” as null pointers
- More importantly though, $X := [a]$, $[a] := a'$ and **DISPOSE** a are not supposed to access cells which have not been previously allocated by **CONS**.
- We build this into the interpretation of $\{A\}c\{B\}$, i.e. for it to be valid, A has to guarantee two things:
 - that B holds after each *terminating* execution of c
 - that c does not try to access cells not allocated by **CONS**
- The second condition is called **fault-avoiding correctness**
- Orthogonal to the partial/total distinction: we're still fine with non-terminating executions

- But the key thing to understand is: assertions are to be interpreted **locally**, relative to the heaplet
- Recall again: a spec $\{A\}c\{B\}$ says that *if c is given a heaplet \mathfrak{h} satisfying A then it will never try to access heap outside of \mathfrak{h} (other than cells allocated during execution) and it will deliver a heaplet satisfying B if it terminates*

Quoting Berdine, Calcagno, O'Hearn, some typos corrected

- Division of heap into heaplets is purely conceptual, not done by the programming language, OS, or hardware
- We combine “small” or “local” assertions into assertions about larger parts of the heap using \star
- Actually, let us give formal semantics for assertions
- Set $\text{States}_{\text{DIMP}} := \text{Stores} \times \text{Heaplets}$, where
 - $\text{Stores} := \text{States}_{\text{IMP}}$, i.e., $\text{Stores} = \text{ProgVar} \rightarrow \mathbb{Z}$ and
 - $\text{Heaplets} := \mathbb{N} \rightarrow \mathbb{Z}$.

- Given $\sigma \in \text{States}_{\text{DIMP}}$ and $\mathfrak{h} \in \text{Heaplets}$, define the forcing relation as follows:
 - $\sigma, \mathfrak{h} \models^I \text{true}$
 - $\sigma, \mathfrak{h} \models^I a_1 == a_2$ if $\llbracket a_1 \rrbracket \sigma I = \llbracket a_2 \rrbracket \sigma I$
 - $\sigma, \mathfrak{h} \models^I a_1 < a_2$ if $\llbracket a_1 \rrbracket \sigma I < \llbracket a_2 \rrbracket \sigma I$
 - $\sigma, \mathfrak{h} \models^I \text{not } B$ if not $\sigma, \mathfrak{h} \models^I B$
 - $\sigma, \mathfrak{h} \models^I \bigwedge_{z \in \mathbb{Z}} B_z$ if $\sigma, \mathfrak{h} \models^I B_z$ for all z . Now the new clauses:
 - $\sigma, \mathfrak{h} \models^I \text{emp}$ if $\mathfrak{h} = \emptyset$
 - $\sigma, \mathfrak{h} \models^I a_1 :-> a_2$ if $\llbracket a_1 \rrbracket \sigma I \in \mathbb{N}$ and $\mathfrak{h} = \{ \{ \llbracket a_1 \rrbracket \sigma I, \llbracket a_2 \rrbracket \sigma I \} \}$
 - $\sigma, \mathfrak{h} \models^I B_1 * B_2$ if there exist $\mathfrak{h}_1, \mathfrak{h}_2$ s.t.
 - $\text{dom} \mathfrak{h}_1 \cap \text{dom} \mathfrak{h}_2 = \emptyset$
 - $\mathfrak{h}_1 \cup \mathfrak{h}_2 = \mathfrak{h}$
 - $\sigma, \mathfrak{h}_1 \models^I B_1$ and $\sigma, \mathfrak{h}_2 \models^I B_2$

Points to note

- $\text{emp} * \text{true}$ is trivially valid
- More generally, any B is equivalent to $B * \text{emp}$...
- ...mathematically, $*$ and emp form a **commutative monoid**
We are talking about formulas quotiented by logical equivalence, that is
- Also, note $a_1 :-> a_2$ can only be true in a heaplet with a singleton domain
- How do you say that in the current heaplet the address denoted by a_1 is allocated and stores the value of a_2 ?
- ...that's right, it's $(a_1 :-> a_2) * \text{true}$
- We can also take $e :-> a_1, \dots, a_n$ to mean
 $(e :-> a_1) * ((e + 1) :-> a_2) * \dots * ((e + n - 1) :-> a_n)$

- We will thus try to characterize our operations using as small portions of heaplet as possible ...
- ...thence **small** or **local** axioms
- **Global** variants thereof will be derived using the *frame rule*
- In our version, we will use the convenience of having quantified metavariables **LogVar** (recall how we simulate quantification) to store the values of elements of **ProgVar**

- The small axiom for allocation:

$$\vdash \{(X==i) \wedge \text{emp}\} X := \text{CONS } \bar{a} \{X :-> \bar{a}[i/X]\}$$
- The small axiom for lookup:

$$\vdash \{(X==i) \wedge (a :-> i')\} X := [a] \{(X==i') \wedge (a[i/X] :-> i')\}$$
- The small axiom for mutation:

$$\vdash \{\exists i.a :-> i\} [a] := a' \{a :-> a'\}$$
- The small axiom for deallocation:

$$\vdash \{\exists i.a :-> i\} \text{DISPOSE } a \{\text{emp}\}$$
- Small axioms indeed!
 - Precondition for **CONS** and postcondition for **DISPOSE**: empty heaplet
 - Postcondition for **CONS**: heaplet the size of \bar{a}
 - Everywhere else: heaplet size 1

- For convenience, write “ $a :-> _$ ” for “ a is allocated”, i.e., “ $\exists i. a :-> i$ ”.
- We can write now the small axiom for mutation as

$$\vdash \{a :-> _ \} [a] := a' \{a :-> a' \}$$
- ...and the small axiom for deallocation as

$$\vdash \{a :-> _ \} \text{DISPOSE } a \{ \text{emp} \}$$
- In order to derive global specifications from these local ones, let us recall the frame rule:

$$\frac{\{A\}c\{B\} \quad \text{no variable occurring in } A' \text{ modified by } c}{\{A * A'\}c\{B * A'\}}$$

- We should make the second premise more specific ...

- Set

$$\begin{aligned} \text{modifies}(X := a) &= \text{modifies}(X := [a]) \\ &= \text{modifies}(X := \text{CONS } \bar{a}) = \{X\}, \end{aligned}$$

$$\begin{aligned} \text{modifies}(\text{SKIP}) &= \text{modifies}(\text{DISPOSE } a) \\ &= \text{modifies}([a] := a') = \emptyset, \end{aligned}$$

$$\begin{aligned} \text{modifies}(c_1; c_2) &= \text{modifies}(\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2) \\ &= \text{modifies}(c_1) \cup \text{modifies}(c_2), \end{aligned}$$

$$\text{modifies}(\text{WHILE } b \text{ DO } c \text{ END}) = \text{modifies}(c).$$

- Also, let $\text{appears}(B)$ be the set of program variables syntactically occurring in B
- Now we can state our rule as

$$\frac{\vdash \{A\} c \{B\} \quad \text{modifies}(c) \cap \text{appears}(A') = \emptyset}{\vdash \{A * A'\} c \{B * A'\}} \text{FRAME}$$

- **Exercise:** What goes wrong without that side condition?

Especially in concurrency, semantics of Hoare triples/syntax of commands is sometimes tortured to avoid side conditions in the frame rule ...

Computing the global specification of deallocation is easy:

$$\text{FRAME} \frac{\vdash \{a:\rightarrow _ \} \text{DISPOSE } a \{ \text{emp} \}}{\vdash \{ (a:\rightarrow _) * B \} \text{DISPOSE } a \{ \text{emp} * B \} \quad \models (\text{emp} * B) \rightarrow B} \text{CONSEQ}$$

$$\vdash \{ (a:\rightarrow _) * B \} \text{DISPOSE } a \{ B \}.$$

Only slightly harder for lookup:

$$\frac{\vdash \{ (X = i) \wedge (a:\rightarrow i') \} X := [a] \{ (X = i') \wedge a[i/X] :\rightarrow i' \}}{\vdash \{ ((X = i) \wedge (a:\rightarrow i')) * A[i'/X] \} X := [a] \{ ((X = i') \wedge a[i/X] :\rightarrow i') * A[i'/X] \} \\ \models (((X = i') \wedge \beta) * \alpha[i'/X]) \rightarrow \beta * \alpha}$$

$$\vdash \{ ((X = i) \wedge (a:\rightarrow i')) * A[i'/X] \} X := [a] \{ (a[i/X] :\rightarrow i') * A \}$$

(The last inference: arithmetical expressions and (in-)equalities between them are heap(let)-independent, hence they propagate through $*$. I'm using here Greek letters for schematic variables: which assertions are substituted for α and β , respectively?)

