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

(lecture version)

Tadeusz Litak

December 11, 2018

Informatik 8, FAU Erlangen-Nürnberg

- Hoare reasoning goes funny when playing with data structures

- Hoare reasoning goes funny when playing with data structures
- At least static arrays, say

- 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 ...

- Lists (single- or doubly-linked), queues, trees, stacks …
- …and with these, issues of pointers, heaps, allocation …

- 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*

- 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

- 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

- 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*

- 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* `AquireSpinLock`

- 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* `AquireSpinLock`
  - 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

- 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* `AquireSpinLock`
  - 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!

- 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* `AquireSpinLock`
  - 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

- 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* `AquireSpinLock`
  - 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?

- And why (shared) mutable data structures are so problematic?
- Let us quote from O'Hearn, Reynolds and Yang:

- 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 ...*

- 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 ...*

- 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
```

- 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

- 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)`

- 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) 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

- `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` 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 ...

- `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

- `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

- `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`

- And now try to reason about a bigger program using `appendlist`
- It can use many other data structures and areas of the heap

- 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

- 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 …

- 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

- 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

- 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

- 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 ...*

- 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*

- 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.*

- *The page with Q's number may be torn out.*
- *P may be blind.*

- *The page with Q's number may be torn out.*
- *P may be blind.*
- *Someone may have deliberately inked out Q's number.*

- *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.*

- *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 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.*

- *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

- ***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

- ***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

- 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: implict 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

- 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: implict 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

- 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 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*

- 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

- 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)

- 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)

- 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)}
```

- 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:

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

- 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 $-\!\!*$

- BI has also other connectives, like the the magic wand $-\!\!*$
- It plays the same role with $*$ as $\rightarrow$ with $\wedge$

- BI has also other connectives, like the the magic wand $-\!\!*$
- It plays the same role with $*$ 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

- BI has also other connectives, like the the magic wand $-\!\!*$
- It plays the same role with $*$ 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 that the separating conjunction though

- BI has also other connectives, like the the magic wand $-\!\!*$
- It plays the same role with $*$ 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 that 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

$$c_1, c_2 ::= \texttt{SKIP} \mid \texttt{X} \texttt{ := } a \mid c1 \texttt{ ; } c2 \mid \texttt{IF } b \texttt{ THEN } c_1 \texttt{ ELSE } c_2 \texttt{ ENDIF} \mid \texttt{WHILE } b \texttt{ DO } c_1 \texttt{ END} \mid$$

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

- We could give a denotational/operational semantics for DIMP in terms of *pairs of* old states (now stores) and heap(let)s ...

- 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 $\mathbb{Z}$

- 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*)

- 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 purse the program we originally tried with IMP too

- 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 purse 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.,

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

  where $i \in$ 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 extended arithmetical expressions EAExp is defined as before, i.e.,

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

  where $i \in$ 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

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

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

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

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

  where $i \in$ 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

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

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

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

  $\bot, \quad a_1 \texttt{ <= } a_2, \quad a_1 \texttt{ >= } a_2, \quad B_1 \vee B_2, \quad B_1 \to B_2,$

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

- Note: the assertion language already is in some ways richer than that of VeriFast and Implicit Dynamic Frames ...

  e.g., presence of infinitary connectives

- 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!

- 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?

- 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) $\sigma : \mathsf{ProgVar} \to \mathbb{Z}$ and heap(let)s $h : \mathbb{N} \rightharpoonup \mathbb{Z}$

- 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) $\sigma : \mathsf{ProgVar} \to \mathbb{Z}$ and heap(let)s $h : \mathbb{N} \rightharpoonup \mathbb{Z}$
- The old assertions, involving just arithmetical expressions and program variables, depend only on the store (no rôle played by the heaplet)

- 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) $\sigma : \mathsf{ProgVar} \to \mathbb{Z}$ and heap(let)s $h : \mathbb{N} \rightharpoonup \mathbb{Z}$
- 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 $\mathbb{N}$

- 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

- 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`.

- 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:

- 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

- 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`

- 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, $\mathtt{X}$ `:=[`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

- 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, $\mathtt{X}$ `:=[`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

- 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

- 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

- 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 lager parts of the heap using $*$

- 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 lager parts of the heap using $*$
- Actually, let us give formal semantics for assertions

- 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 lager parts of the heap using $*$
- Actually, let us give formal semantics for assertions
- Set $\mathsf{States}_{\mathsf{DIMP}} := \mathsf{Stores} \times \mathsf{Heaplets}$, where

- 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 lager parts of the heap using $*$
- Actually, let us give formal semantics for assertions
- Set $\mathsf{States}_{\mathsf{DIMP}} := \mathsf{Stores} \times \mathsf{Heaplets}$, where
  - $\mathsf{Stores} := \mathsf{States}_{\mathsf{IMP}}$, i.e., $\mathsf{Stores} = \mathsf{ProgVar} \to \mathbb{Z}$ and

- 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 lager parts of the heap using $*$
- Actually, let us give formal semantics for assertions
- Set $\mathsf{States}_{\mathsf{DIMP}} := \mathsf{Stores} \times \mathsf{Heaplets}$, where
  - $\mathsf{Stores} := \mathsf{States}_{\mathsf{IMP}}$, i.e., $\mathsf{Stores} = \mathsf{ProgVar} \to \mathbb{Z}$ and
  - $\mathsf{Heaplets} := \mathbb{N} \rightharpoonup \mathbb{Z}$.

- Given $\sigma \in \mathsf{States}_{\mathsf{DIMP}}$ and $\mathfrak{h} \in \mathsf{Heaplets}$, define the forcing relation as follows:
- $\sigma, \mathfrak{h} \vDash^I \texttt{true}$
- $\sigma, \mathfrak{h} \vDash^I a_1 \texttt{ == } a_2$ if $[\![a_1]\!]\sigma I = [\![a_2]\!]\sigma I$
- $\sigma, \mathfrak{h} \vDash^I a_1 \texttt{ < } a_2$ if $[\![a_1]\!]\sigma I < [\![a_2]\!]\sigma I$
- $\sigma, \mathfrak{h} \vDash^I \texttt{not } B$ if not $\sigma, \mathfrak{h} \vDash^I B$
- $\sigma, \mathfrak{h} \vDash^I \bigwedge_{z \in \mathbb{Z}} B_z$ if $\sigma, \mathfrak{h} \vDash^I B_z$ for all z. Now the new clauses:

- Given $\sigma \in \mathsf{States}_{\mathsf{DIMP}}$ and $\mathfrak{h} \in \mathsf{Heaplets}$, define the forcing relation as follows:
- $\sigma, \mathfrak{h} \vDash^I \texttt{true}$
- $\sigma, \mathfrak{h} \vDash^I a_1 \texttt{ == } a_2$ if $[\![a_1]\!]\sigma I = [\![a_2]\!]\sigma I$
- $\sigma, \mathfrak{h} \vDash^I a_1 \texttt{ < } a_2$ if $[\![a_1]\!]\sigma I < [\![a_2]\!]\sigma I$
- $\sigma, \mathfrak{h} \vDash^I \texttt{not } B$ if not $\sigma, \mathfrak{h} \vDash^I B$
- $\sigma, \mathfrak{h} \vDash^I \bigwedge_{z \in \mathbb{Z}} B_z$ if $\sigma, \mathfrak{h} \vDash^I B_z$ for all z. Now the new clauses:
- $\sigma, \mathfrak{h} \vDash^I \texttt{emp}$ if $\mathfrak{h} = \varnothing$

- Given $\sigma \in \mathsf{States}_{\mathsf{DIMP}}$ and $\mathfrak{h} \in \mathsf{Heaplets}$, define the forcing relation as follows:
- $\sigma, \mathfrak{h} \vDash^I \texttt{true}$
- $\sigma, \mathfrak{h} \vDash^I a_1 \texttt{ == } a_2$ if $[\![a_1]\!]\sigma I = [\![a_2]\!]\sigma I$
- $\sigma, \mathfrak{h} \vDash^I a_1 \texttt{ < } a_2$ if $[\![a_1]\!]\sigma I < [\![a_2]\!]\sigma I$
- $\sigma, \mathfrak{h} \vDash^I \texttt{not } B$ if not $\sigma, \mathfrak{h} \vDash^I B$
- $\sigma, \mathfrak{h} \vDash^I \bigwedge_{z \in \mathbb{Z}} B_z$ if $\sigma, \mathfrak{h} \vDash^I B_z$ for all z. Now the new clauses:
- $\sigma, \mathfrak{h} \vDash^I \texttt{emp}$ if $\mathfrak{h} = \varnothing$
- $\sigma, \mathfrak{h} \vDash^I a_1 \texttt{ :-> } a_2$ if $[\![a_1]\!]\sigma I \in \mathbb{N}$ and $\mathfrak{h} = \{\langle [\![a_1]\!]\sigma I, [\![a_2]\!]\sigma I \rangle\}$

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

Points to note

- `emp * true` is trivially valid

Points to note

- `emp * true` is trivially valid
- More generally, any B is equivalent to B `* emp` …

Points to note

- `emp * true` is trivially valid
- More generally, any B is equivalent to B `*` `emp` …
- …mathematically, `*` and `emp` form a commutative monoid

  We are talking about formulas quotiented by logical equivalence, that is

Points to note

- `emp * true` is trivially valid
- More generally, any B is equivalent to B `* 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

Points to note

- `emp * true` is trivially valid
- More generally, any B is equivalent to B `* 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?

Points to note

- `emp * true` is trivially valid
- More generally, any B is equivalent to B `* 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) `* true`

Points to note

- `emp * true` is trivially valid
- More generally, any B is equivalent to B `* 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) `* true`
- We can also take e `:->` a_1, …, a_n to mean

  (e `:->` a_1) `*` ((e `+ 1`) `:->` a_2) `*` … `*` ((e `+` $n-1$) `:->` a_n)

- We will thus try to characterize our operations using as small portions of heaplet as possible ...

- We will thus try to characterize our operations using as small portions of heaplet as possible ...
- ...thence small or local axioms

- 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*

- 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)$\wedge$emp} X := CONS $\overline{a}$ {X :-> $\overline{a}[i/\mathtt{X}]$}

- The small axiom for allocation:

  $\vdash$ {(X==i)$\wedge$emp} X := CONS $\overline{a}$ {X :-> $\overline{a}$[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 allocation:

  $\vdash$ {(X==i)$\wedge$emp} X := CONS $\overline{a}$ {X :-> $\overline{a}[i/\mathtt{X}]$}

- The small axiom for lookup:

  $\vdash$ {(X==i)$\wedge$(a :-> i')} X := [a] {(X==i')$\wedge$($a[i/\mathtt{X}]$:-> i')}

- The small axiom for mutation:

  $\vdash$ {$\exists i.a$:-> i} [a] := a' {a :-> a'}

- The small axiom for allocation:

  $\vdash$ {(X==i)$\wedge$emp} X := CONS $\overline{a}$ {X :-> $\overline{a}[i/\mathrm{X}]$}

- The small axiom for lookup:

  $\vdash$ {(X==i)$\wedge$(a :-> i')} X := [a] {(X==i')$\wedge$($a[i/\mathrm{X}]$:-> 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} DISPOSE a {emp}

- The small axiom for allocation:

  $\vdash$ {(X==i)$\wedge$emp} X := CONS $\overline{a}$ {X :-> $\overline{a}[i/\mathrm{X}]$}

- The small axiom for lookup:

  $\vdash$ {(X==i)$\wedge$(a :-> i')} X := [a] {(X==i')$\wedge$($a[i/\mathrm{X}]$:-> 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} DISPOSE a {emp}

- Small axioms indeed!

- The small axiom for allocation:

  $\vdash$ {(X==i)$\wedge$emp} X := CONS $\overline{a}$ {X :-> $\overline{a}[i/\mathsf{X}]$}

- The small axiom for lookup:

  $\vdash$ {(X==i)$\wedge$(a :-> i')} X := [a] {(X==i')$\wedge$($a[i/\mathsf{X}]$:-> 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} DISPOSE a {emp}

- Small axioms indeed!
  - Precondition for CONS and postcondition for DISPOSE: empty heaplet

- The small axiom for allocation:

  $\vdash$ {(X==i)$\wedge$emp} X := CONS $\overline{a}$ {X :-> $\overline{a}$[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} DISPOSE a {emp}

- Small axioms indeed!
  - Precondition for CONS and postcondition for DISPOSE: empty heaplet
  - Postcondition for CONS: heaplet the size of $\overline{a}$

- The small axiom for allocation:

  $\vdash$ {(X==i)$\wedge$emp} X := CONS $\overline{a}$ {X :-> $\overline{a}$[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} DISPOSE a {emp}

- Small axioms indeed!
  - Precondition for CONS and postcondition for DISPOSE: empty heaplet
  - Postcondition for CONS: heaplet the size of $\overline{a}$
  - Everywhere else: heaplet size 1

- For convenience, write "a`:->_`" for "a is allocated", i.e., "$\exists i.a$`:->`i".

- 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'`}`

- 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`:->_}` `DISPOSE` a `{emp}`

- 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`:->_}` `DISPOSE` a `{emp}`

- In order to derive global specifications from these local ones, let us recall the frame rule:

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

- 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`:->_}` `DISPOSE` a `{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

$$
\begin{aligned}
&\mathit{modifies}(\texttt{X := } a) = \mathit{modifies}(\texttt{X := [}a\texttt{]}) \\
&\qquad\qquad = \mathit{modifies}(\texttt{X := CONS } \overline{a}) = \{\texttt{X}\}, \\
&\mathit{modifies}(\texttt{SKIP}) = \mathit{modifies}(\texttt{DISPOSE } a) \\
&\qquad\qquad = \mathit{modifies}(\texttt{[}a\texttt{] := } a') = \varnothing, \\
&\mathit{modifes}(c_1\texttt{;}c_2) = \mathit{modifies}(\texttt{IF } b \texttt{ THEN } c_1 \texttt{ ELSE } c_2) \\
&\qquad\qquad = \mathit{modifies}(c_1) \cup \mathit{modifies}(c_2), \\
&\mathit{modifies}(\texttt{WHILE } b \texttt{ DO } c \texttt{ END}) = \mathit{modifies}(c).
\end{aligned}
$$

- Set

$$
\begin{aligned}
&\mathit{modifies}(\texttt{X := } a) = \mathit{modifies}(\texttt{X := [}a\texttt{]}) \\
&\qquad\qquad = \mathit{modifies}(\texttt{X := CONS } \overline{a}) = \{\texttt{X}\}, \\
&\mathit{modifies}(\texttt{SKIP}) = \mathit{modifies}(\texttt{DISPOSE } a) \\
&\qquad\qquad = \mathit{modifies}(\texttt{[}a\texttt{] := } a') = \varnothing, \\
&\mathit{modifes}(c_1\texttt{;}c_2) = \mathit{modifies}(\texttt{IF } b \texttt{ THEN } c_1 \texttt{ ELSE } c_2) \\
&\qquad\qquad = \mathit{modifies}(c_1) \cup \mathit{modifies}(c_2), \\
&\mathit{modifies}(\texttt{WHILE } b \texttt{ DO } c \texttt{ END}) = \mathit{modifies}(c).
\end{aligned}
$$

- Also, let $\mathit{appears}(B)$ be the set of program variables syntactically occurring in B

- Set

$$
\begin{aligned}
&\mathit{modifies}(\mathtt{X\ :=\ } a) = \mathit{modifies}(\mathtt{X\ :=\ [}a\mathtt{]}) \\
&\qquad\qquad = \mathit{modifies}(\mathtt{X\ :=\ CONS\ } \overline{a}) = \{\mathtt{X}\}, \\
&\mathit{modifies}(\mathtt{SKIP}) = \mathit{modifies}(\mathtt{DISPOSE\ } a) \\
&\qquad\qquad = \mathit{modifies}(\mathtt{[}a\mathtt{]\ :=\ } a') = \varnothing, \\
&\mathit{modifes}(c_1\,\mathtt{;}\,c_2) = \mathit{modifies}(\mathtt{IF\ } b\ \mathtt{THEN\ } c_1\ \mathtt{ELSE\ } c_2) \\
&\qquad\qquad = \mathit{modifies}(c_1) \cup \mathit{modifies}(c_2), \\
&\mathit{modifies}(\mathtt{WHILE\ } b\ \mathtt{DO\ } c\ \mathtt{END}) = \mathit{modifies}(c).
\end{aligned}
$$

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

$$\frac{\vdash \{A\}\,c\,\{B\} \qquad \mathit{modifies}(c) \cap \mathit{appears}(A') = \varnothing}{\vdash \{A * A'\}\,c\,\{B * A'\}}\ \textsc{Frame}$$

- Set

$$\begin{aligned}
&\mathit{modifies}(\texttt{X := } a) = \mathit{modifies}(\texttt{X := [}a\texttt{]}) \\
&\qquad\qquad = \mathit{modifies}(\texttt{X := CONS } \overline{a}) = \{\texttt{X}\}, \\
&\mathit{modifies}(\texttt{SKIP}) = \mathit{modifies}(\texttt{DISPOSE } a) \\
&\qquad\qquad = \mathit{modifies}(\texttt{[}a\texttt{] := } a') = \varnothing, \\
&\mathit{modifes}(c_1\texttt{;}c_2) = \mathit{modifies}(\texttt{IF } b \texttt{ THEN } c_1 \texttt{ ELSE } c_2) \\
&\qquad\qquad = \mathit{modifies}(c_1) \cup \mathit{modifies}(c_2), \\
&\mathit{modifies}(\texttt{WHILE } b \texttt{ DO } c \texttt{ END}) = \mathit{modifies}(c).
\end{aligned}$$

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

$$\frac{\vdash \{A\}\, c\, \{B\} \qquad \mathit{modifies}(c) \cap \mathit{appears}(A') = \varnothing}{\vdash \{A * A'\}\, c\, \{B * A'\}}\ \textsc{Frame}$$

- 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:

$$\textsc{Frame}\ \frac{\vdash \{a :\!\!\rightarrow \_\}\,\texttt{DISPOSE}\ a\,\{\texttt{emp}\}}{\vdash \{(a :\!\!\rightarrow \_) * B\}\,\texttt{DISPOSE}\ a\,\{\texttt{emp} * B\}}$$

Computing the global specification of deallocation is easy:

$$\textsc{Conseq}\ \frac{\textsc{Frame}\ \dfrac{\vdash \{a :\!\!\to \_\}\,\texttt{DISPOSE}\ a\,\{\texttt{emp}\}}{\vdash \{(a :\!\!\to \_) * B\}\,\texttt{DISPOSE}\ a\,\{\texttt{emp} * B\}} \qquad \vDash (\texttt{emp} * B) \to B}{\vdash \{(a :\!\!\to \_) * B\}\,\texttt{DISPOSE}\ a\,\{B\}.}$$

Computing the global specification of deallocation is easy:

$$\textsc{Conseq}\ \frac{\textsc{Frame}\ \dfrac{\vdash \{a :\!\!\to \_\}\,\texttt{DISPOSE}\ a\,\{\texttt{emp}\}}{\vdash \{(a :\!\!\to \_) * B\}\,\texttt{DISPOSE}\ a\,\{\texttt{emp} * B\}} \qquad \vDash (\texttt{emp} * B) \to B}{\vdash \{(a :\!\!\to \_) * B\}\,\texttt{DISPOSE}\ a\,\{B\}.}$$

Only slightly harder for lookup:

Computing the global specification of deallocation is easy:

$$\textsc{Conseq}\ \frac{\textsc{Frame}\ \dfrac{\vdash \{a :\!\!\to \_\}\,\texttt{DISPOSE}\ a\,\{\texttt{emp}\}}{\vdash \{(a :\!\!\to \_) * B\}\,\texttt{DISPOSE}\ a\,\{\texttt{emp} * B\}} \qquad \vDash (\texttt{emp} * B) \to B}{\vdash \{(a :\!\!\to \_) * B\}\,\texttt{DISPOSE}\ a\,\{B\}.}$$

Only slightly harder for lookup:

$$\frac{\vdash \{(X = i) \wedge (a :\!\!\to i')\}\,\mathsf{X}\!:\!=[a]\,\{(X = i') \wedge a[i/X] :\!\!\to i'\}}{\vdash \{((X = i) \wedge (a :\!\!\to i')) * A[i'/X]\}\,\mathsf{X}\!:\!=[a]\,\{((X = i') \wedge a[i/X] :\!\!\to i') * A[i'/X]\}}$$

Computing the global specification of deallocation is easy:

$$\textsc{Conseq}\ \frac{\textsc{Frame}\ \dfrac{\vdash \{a :\!\!\to \_\}\ \texttt{DISPOSE}\ a\ \{\texttt{emp}\}}{\vdash \{(a :\!\!\to \_) * B\}\ \texttt{DISPOSE}\ a\ \{\texttt{emp} * B\}} \qquad \vDash (\texttt{emp} * B) \to B}{\vdash \{(a :\!\!\to \_) * B\}\ \texttt{DISPOSE}\ a\ \{B\}.}$$

Only slightly harder for lookup:

$$\frac{\dfrac{\vdash \{(X = i) \wedge (a :\!\!\to i')\}\ \texttt{X:=[}a\texttt{]}\ \{(X = i') \wedge a[i/X] :\!\!\to i'\}}{\vdash \{((X = i) \wedge (a :\!\!\to i')) * A[i'/X]\}\ \texttt{X:=[}a\texttt{]}\ \{((X = i') \wedge a[i/X] :\!\!\to i') * A[i'/X]\}} \qquad \vDash (((X = i') \wedge \beta) * \alpha[i'/X]) \to \beta * \alpha}{\vdash \{((X = i) \wedge (a :\!\!\to i')) * A[i'/X]\}\ \texttt{X:=[}a\texttt{]}\ \{(a[i/X] :\!\!\to i') * A\}}$$

Computing the global specification of deallocation is easy:

$$\textsc{Conseq}\ \frac{\textsc{Frame}\ \dfrac{\vdash \{a :\!\!\to \_\}\,\texttt{DISPOSE}\ a\,\{\texttt{emp}\}}{\vdash \{(a :\!\!\to \_)*B\}\,\texttt{DISPOSE}\ a\,\{\texttt{emp}*B\}} \qquad \vDash (\texttt{emp}*B) \to B}{\vdash \{(a :\!\!\to \_)*B\}\,\texttt{DISPOSE}\ a\,\{B\}.}$$

Only slightly harder for lookup:

$$\frac{\dfrac{\vdash \{(X = i) \wedge (a :\!\!\to i')\}\,\texttt{X:=[}a\texttt{]}\,\{(X = i') \wedge a[i/X] :\!\!\to i'\}}{\vdash \{((X = i) \wedge (a :\!\!\to i'))*A[i'/X]\}\,\texttt{X:=[}a\texttt{]}\,\{((X = i') \wedge a[i/X] :\!\!\to i')*A[i'/X]\}} \qquad \vDash (((X = i') \wedge \beta)*\alpha[i'/X]) \to \beta*\alpha}{\vdash \{((X = i) \wedge (a :\!\!\to i'))*A[i'/X]\}\,\texttt{X:=[}a\texttt{]}\,\{(a[i/X] :\!\!\to i')*A\}}$$

(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?)

...and similarly for allocation:

...and similarly for allocation:

$$\frac{\vdash \{(X = i) \wedge \mathsf{emp}\}\ \mathtt{X{:=}CONS}\ \overline{a}\ \{X :\!\!\rightarrow \overline{a}[i/X]\}}{\vdash \{((X = i) \wedge \mathsf{emp}) * A[i/X]\}\ \mathtt{X{:=}CONS}\ \overline{a}\ \{(X :\!\!\rightarrow \overline{a}[i/X]) * A[i/X]\}}\ \textsc{Frame}$$

...and similarly for allocation:

$$\dfrac{\dfrac{\vdash \{(X = i) \wedge \mathsf{emp}\}\, \mathtt{X{:=}CONS}\, \overline{a}\, \{X :\!\!\rightarrow \overline{a}[i/X]\}}{\vdash \{((X = i) \wedge \mathsf{emp}) * A[i/X]\}\, \mathtt{X{:=}CONS}\, \overline{a}\, \{(X :\!\!\rightarrow \overline{a}[i/X]) * A[i/X]\}}\ \textsc{Frame} \qquad \models (X = i) \wedge A \rightarrow ((X = i) \wedge \mathsf{emp}) * A[i/X]}{\vdash \{(X = i) \wedge A\}\, \mathtt{X{:=}CONS}\, \overline{a}\, \{(X :\!\!\rightarrow \overline{a}[i/X]) * A[i/X])\}}\ \textsc{conseq}$$

...and similarly for allocation:

$$\dfrac{\dfrac{\vdash \{(X = i) \wedge \mathsf{emp}\}\, \mathtt{X{:=}CONS}\, \overline{a}\, \{X :\!\!\to \overline{a}[i/X]\}}{\vdash \{((X = i) \wedge \mathsf{emp}) * A[i/X]\}\, \mathtt{X{:=}CONS}\, \overline{a}\, \{(X :\!\!\to \overline{a}[i/X]) * A[i/X]\}}\ \textsc{Frame}}{\vdash \{(X = i) \wedge A\}\, \mathtt{X{:=}CONS}\, \overline{a}\, \{(X :\!\!\to \overline{a}[i/X]) * A[i/X])\}}\ \textsc{conseq}$$

Finally, it's breathtakingly trivial for mutate:

$$\dfrac{\vdash \{a :\!\!\to \_\}\, [a] := a'\, \{a :\!\!\to a'\}}{\vdash \{\{(a :\!\!\to \_) * B\}\, [a] := a'\, \{(a :\!\!\to a') * B\}}\ \textsc{Frame}$$