Software Verification

Matthew Parkinson
Lent 2010 (16 Lectures)

Part II: Heap

Overview

Part I: Basics

Part 2: Heap

- Encoding the heap
- Spec#/JML
- Separation logic basics
- Separation logic lists/trees

Part 3: Concurrency

Heap data structures

Related material can be found in
Richard Bornat has a paper that explains the difficulties nicely
http://www.eis.mdx.ac.uk/staffpages/r_bornat/papers/MPC2000.pdf

More advanced material can be found in
Peter Müller's thesis
http://people.inf.ethz.ch/lehnerh/pm/publications/getpdf.php?bibname=Own&id=Mueller01.pdf

Rustan Leino's thesis
http://research.microsoft.com/en-us/um/people/leino/papers/Caltech-CS-TR-95-03.pdf

Cristiano Calcagno's thesis
http://www.doc.ic.ac.uk/~ccris/ftp/thesis.pdf

Encoding heap and fields

In languages like C a common source of problems is pointers in the heap.

C ::= … | [E] := E | x := [E] | x := new | …

5

We will extend our representation to allow allocation of memory. We will just allow allocation of a two-word block, that is,

x := new

means that

[x]

and

[x+1]

are allocated after calling it.

We can extend this to blocks of varying sizes in the style of C, but for illustrative purposes we will simply present the two location version, as this allows the encoding of lists and trees.

Rules

{ P [@heap:= @heap{E ← E'}] ∧ alloc(E) }
  [E] := E'
{ P }

{ P [x := @heap[E]] ∧ alloc(E) }
  x := [E]
{ P }

6

The system's initial pre-condition will be

∀x. @heap[x] = @unalloc

We change the elements of @heap to contain the values from the integers and a special value @unalloc. We then define

alloc(E) = @heap[E] ≠ @unalloc

Rules

$$\{\ \forall y.\ @heap[y] = @unalloc \wedge @heap[y+1] = @unalloc \wedge y > 0 \Rightarrow P\ [x := y, @heap := @heap\{y \leftarrow 0, y+1 \leftarrow 0\}]\ \}$$

x := new

{ P }

Example

{ @heap[3] = 4 }
x := new;
y := [x]
{ x ≠ 3 ∧ y = 0 }

What does this say if there the heap does not have space to allocate a new block?

Can we make assumption about where it will allocate a new block?

{ @heap[3] = 4 }
{ @heap[3] ≠ @unalloc }
{ ∀y. @heap[y] = @unalloc ∧ y = 3 ⇒ false }
{ ∀y. @heap[y] = @unalloc ∧ @heap[y+1] = @unalloc ⇒ y ≠ 3 }
{ ∀y. @heap[y] = @unalloc ∧ @heap[y+1] = @unalloc ∧ y>0
⇒ (y ≠ 3 ∧ @heap{y ← 0}{y+1 ← 0} [y] = 0) }
x := new;
{ x ≠ 3 ∧ @heap[x] = 0 }
y := [x]
{ x ≠ 3 ∧ y = 0 }

Exercises

Verify

{ ∀x. @heap[x] ≠ @unalloc }
y := new
{ false }

and

{ @heap[2] = 2 }
y := new
[y] := 3
{ @heap[2] = 2 }

Representing a list

First attempt:

list(x) ⇔ x = 0 ∨ ∃y. @heap[x] = y ∧ list(y)

Admits cycles

Consider the heap given by
@heap[4] = 4

This satisfies the axiom/definition: assuming we take a greatest fixed point of the definition.

Representing a list

$$\text{list}(x,vs) \Leftrightarrow$$
$$x = 0 \wedge vs = \{\}$$
$$\vee\ \exists y.\ @heap[x] = y \wedge \text{list}(y, vs \setminus \{x\}) \wedge x \in vs$$

We can see that
@heap[4] = 4
does not satisfy this definition.

Assume
@heap[4] = 4 ⇒ list(4,vs)

By definition these means
@heap[4] = 4 ⇒ list(4, vs \ {4})

Unrolling the definition again, we get
@heap[4] = 4 ⇒ 4 ∈ vs \ {4}

But 4 ∈ vs \ {4} is false, so it cannot be a list.

List creation

```
let create_list () =
    local x,y;
    x := new;
    if nondet() then
       y := create_list ();
       [x] := y;
    return x
in …
```


This function creates a list of a random but non-empty length. nondet() randomly returns either true or false.

List creation specification

First attempt

{ true } create_list () { ∃vs. list(return, vs) ∧ vs≠{} }

Specification does not say what is unmodified, that is we cannot prove

{ @heap[4] = 3 }
x := create_list ()
{ @heap[4] = 3 ∧ ∃vs. list(x,vs) }

with this specification.

List creation specification

Second attempt

{ oldheap = @heap }
create_list ()
{ ∃vs. list(return, vs) ∧ vs≠{}
∧ ∀i ∉ vs. @heap[i] = oldheap[i] }

Is this disjoint from previously allocated memory? Could vs be already allocated memory locations?

We still cannot prove

{ @heap[4] = 3 }
x := create_list ()
{ @heap[4] = 3 ∧ ∃vs. list(x,vs) }

as we don't know that vs and 4 are disjoint.

List creation specification

{ oldheap = @heap }
create_list ()
{ ∃vs. list(return, vs) ∧ vs≠{}
∧ ∀i ∉ vs. @heap[i] = oldheap[i]
∧ ∀i ∈ vs. oldheap[i] = @unalloc }

Exercise

Prove
{ initheap }
x := create_list ();
y := create_list ();
{ x ≠ y }

Now, we can verify the example

{ @heap[4] = 3 }
{ ∃oldheap. oldheap = @heap ∧ oldheap[4] = 3 }
{ oldheap = @heap ∧ oldheap[4] = 3 }
{ oldheap = @heap }
x := create_list ()
{ ∃vs. list(x,vs) ∧ ∀i ∉ vs. @heap[i] = oldheap[i] ∧ ∀i ∈ vs. oldheap[i] = @unalloc }
{ ∃vs. list(x,vs) ∧ ∀i ∉ vs. @heap[i] = oldheap[i] ∧ ∀i ∈ vs. oldheap[i] = @unalloc ∧ oldheap[4] = 3}
{∃oldheap. ∃vs. list(x,vs) ∧ ∀i ∉ vs. @heap[i] = oldheap[i] ∧ ∀i ∈ vs. oldheap[i] = @unalloc ∧ oldheap[4] = 3}
{ @heap[4] = 3 ∧ ∃vs. list(x,vs) }

The interesting step is the final one. At this point we can see that
∀i ∈ vs. oldheap[i] = @unalloc ∧ oldheap[4] = 3
⇒ 4 ∉ vs

And hence by ∀i ∉ vs. @heap[i] = oldheap[i] and oldheap[4] = 3
we can see that @heap[4] = 3

If we removed the "∧ vs≠{} " from the specification, demonstrate where the proof would fail.

Exercise: verify the body

{ oldheap = @heap }
local x,y;
x := new;
if nondet() then
  y := create_list ();
  [x] := y;
return x
{ ∃vs. list(return, vs)
  ∧ ∀i ∉ vs. @heap[i] = oldheap[i]
  ∧ ∀i ∈ vs. oldheap[i] = @unalloc }

Framing

What stays the same is becoming the majority of the proof!

Solutions

The difficulty here is we have to state the obvious.

Solutions, we will now explore

- Enforce a programming discipline
  - JML/Spec#/ESCJava
- Make logic more expressive
  - Separation logic

I have a personal bias towards the second, because that is my research area. Please take this into account when following the presentation, and draw your own conclusions.

Spec#/JML

I will provide separate notes for this lecture as it will be given from pre-prepared slides.

I would recommend reading the following paper:
http://research.microsoft.com/en-us/um/people/leino/papers/krml189.pdf

Separation logic

I would recommend reading John Reynolds's notes on separation logic to accompany this part of the course.

http://www.cs.cmu.edu/~jcr/copenhagen08.pdf

A new approach

Recall our original specification for creating a list:

{ true } create_list () { ∃vs. list(return, vs) }

Idea pre-condition describes what the code requires to execute, and post-condition says what it will provide on return. Importantly it won't describe the whole memory of the system, just the bit of interest. Everything not described will be unchanged. The pre-condition must state all the memory we will use.

So, we might like to write

{ empty } create_list() { list(return) }

that is, we require no memory to execute, and we return the memory for a list starting at return.

The next two lectures will develop all of the theory to make the intuitive specification formally correct.

The key feature is we are dealing with a logic of partial states. Assertion will not describe the whole state, just part of it.

Programming language

C ::= … | [E] := E | x := [E] | x := new | dispose E | …

Now, we will even add the ability to deallocate memory.

Exercise, extend the earlier encoding to deal with deallocation.

A new logic

P ::= B | P ∧ P | P ∨ P | ¬P | P ⇒ P | ∃x. P | ∀x. P

| E ↦ E' | empty | P * P | P -* P

We take classical logic and add two primitive assertions and two connectives.

We have two basic assertions for describing the memory:

E ↦ E‘ “E points to E' ”

This means the heap contains a single element

empty

This means the heap is empty

The two new connectives mean

P * Q Separating conjunction (or star)

This means the heap can be split into two disjoint pieces, such that one satisfies P and the other Q.

P -* Q Separating implication (or wand)

This one means for all extension of the heap that satisfy P, the extended heap satisfies Q.

The separating implication is hard to understand.

Semantics of assertions

Assertions are interpreted over partial heaps

$h : Loc \rightharpoonup Val + unalloc$

$s : Var \rightarrow Val$

where Loc is the positive non-zero integers
and Val is all the integers

Semantics of assertions

We say a stack, s, and heap, h, satisfy an assertion iff

$s,h \models empty \quad \Leftrightarrow \quad h = \{\}$

$s,h \models E \mapsto E' \quad \Leftrightarrow \quad h = \{ [\![E]\!]_s \mapsto [\![E']\!]_s \}$

$s,h \models P_1 * P_2$
$\Leftrightarrow \exists h_1,h_2.\ h = h_1 \uplus h_2 \wedge s,h_1 \models P_1 \wedge s,h_2 \models P_2$

We define a composition operation on heaps

$h = h1 \uplus h2 \quad \Leftrightarrow$

$\forall$ i. h1(i) defined $\Rightarrow$ h(i) = h1(i) and h2(i) undefined

$\wedge$ $\forall$ i. h2(i) defined $\Rightarrow$ h(i) = h2(i) and h1(i) undefined

$\wedge$ $\forall$ i. h1(i) undefined $\wedge$ h2(i) undefined $\Rightarrow$ h(i) undefined

This combines heaps that are disjoint. Do not have the same location allocated.

Intuitively, we can relate these to our earlier assertion language as

empty = $\forall x.$ @heap[x] = @unalloc

$E \mapsto E'$ = @heap[E] = E' $\wedge$ $\forall x.$ $x \neq E \Rightarrow$ @heap[x] = @unalloc

and could view the separating conjunction as

P * Q = $\exists$heap1,heap2. P[@heap := heap1] $\wedge$ Q [@heap := heap2] $\wedge$ union(heap1,heap2) = @heap $\wedge$ disjoint (heap1,heap2)

where

disjoint(heap1,heap2) = $\nexists x.$ heap1[x] $\neq$ @unalloc $\wedge$ heap2[x] $\neq$ @unalloc

heap2[i] = unalloc $\Rightarrow$ union(heap1,heap2)[i] = heap1[i]

heap2[i] $\neq$ unalloc $\Rightarrow$ union(heap1,heap2)[i] = heap2[i]

[Thanks to Tony Hoare for suggesting this intuitive presentation]

Magic wand

$s,h \models P_1 \text{ -* } P_2$

$\Leftrightarrow \forall h_1,h_2.\ h_2 = h_1 \uplus h \wedge s,h_1 \models P_1 \Rightarrow s,h_2 \models P_2$

This is useful for giving weakest pre-condition rules and axioms for separation logic.

Can be useful for loop invariants.

Example implications

The following implications all hold:

$P * \text{empty} \Leftrightarrow P$

$P_1 * (P_2 * P_3) \Leftrightarrow (P_1 * P_2) * P_3$

$P_1 * P_2 \Leftrightarrow P_2 * P_1$

$P_1 * (P_2 \vee P_3) \Leftrightarrow (P_1 * P_2) \vee (P_1 * P_3)$

$P_1 * (P_2 \wedge P_3) \Rightarrow (P_1 * P_2) \wedge (P_1 * P_3)$

Exercise prove these implications.

For example,

The first requires we prove
$\forall s,h.\ \ s,h \models (P * \text{empty}) \Leftrightarrow P$

Pick arbitrary s and h. We must prove
$s,h \models P * \text{empty}$ iff $s,h \models P$

Left to right direction:
Assume $s,h \models P * \text{empty}$
Therefore, there exists h_1 and h_2, such that $s,h_1 \models P$ and $s,h_2 \models \text{empty}$ and $h_1 * h_2 = h$
By $(s,h_2 \models \text{empty})$ we know h_2 must be the empty heap, {}.
By $(h_1 * \{\} = h)$, we know $h_1 = h$.
Hence $s,h \models P$ as required.

Right to left direction:
Assume $s,h \models P$
Prove $s,h \models P * \text{empty}$.
This require we find h_1 and h_2 such that $s,h_1 \models P$ and $s,h_2 \models \text{empty}$ and $h_1 * h_2 = h$.
Pick h_1 as h, and h_2 as {}. Rest follows trivially.

Example implications

In classical logic we have

$P \wedge \neg P \Rightarrow$ false $\qquad P \Rightarrow P \wedge P \qquad P \wedge Q \Rightarrow P$

Do we have the same with *?

- $P * \neg P \Rightarrow$ false
- $P \Rightarrow P * P$
- $P * Q \Rightarrow P$

Example implications

Do the following implications hold?

$x \mapsto 5 * y \mapsto 6 \Rightarrow x \neq y$

$x \mapsto 5 * y \mapsto 5 \Rightarrow x \neq y$

$x \mapsto 5 \wedge y \mapsto 5 \Rightarrow x = y$

$x \mapsto 5 \wedge y \mapsto 6 \Rightarrow$ false

$x \mapsto 5 * x \mapsto 4 \Rightarrow$ false

Give concrete interpretations for P and Q for each implication and exhibit a heap such that this does not hold.

The Invariance rule is dead!

$$\frac{\Gamma \vdash \{ P \}\ C\ \{ Q \}}{\Gamma \vdash \{ P \wedge R \}\ C\ \{ Q \wedge R \}}$$

Provided free variables of R are not in mod(C).

Tight interpretation

Semantics of judgements

$\{ P \}\ C\ \{ Q \}$

means

If $s,h \models P$ then

- C,s,h safe
- $C,s,h \rightarrow^* skip,s',h' \Rightarrow s',h' \models Q$

Doesn't access memory outside h.

Recall earlier we had an invariance rule. Now given this interpretation of the logic it is no longer valid.

Consider

$\{ x \mapsto 4 \wedge x \mapsto 4 \}$
$\{ x \mapsto 4 \}$
x := 5;
$\{ x \mapsto 4 \}$
$\{ x \mapsto 5 \wedge x \mapsto 4 \}$

We do not modify any variables in this so using the invariance rule it is perfectly valid to preserve $x \mapsto 4$. The problem comes from the implicit heap variable. In some sense, we have modified the @heap variable. If we treat this as modified then we could not preserve any facts about the heap that weren't specified.

To deal with this we rewrite our operational semantics to deal with the heap access being unalloc.

x := [E], s, h → fault provided h(⟦ E ⟧s) = @unalloc
[E] := E', s, h → fault provided h(⟦ E ⟧s) = @unalloc
dispose E, s, h → fault provided h(⟦ E ⟧s) = @unalloc or h(⟦ E+1 ⟧s) = @unalloc

We also give reductions for successful computations

x := [E], s, h → skip, s[x:=v], h provided h(⟦ E ⟧s) = v
[E] := E', s, h → skip, s, h[v := ⟦E'⟧s] provided h(⟦ E ⟧s) = v
dispose E, s, h → skip, s, h[⟦E⟧ := @unalloc, ⟦E+1⟧ := @unalloc]
provided h(⟦ E ⟧s) ≠ @unalloc and h(⟦ E+1 ⟧s) ≠ @unalloc
x := new, s, h → skip, s[x:=v], h[v:=0, v+1:=0]
provided {h(v), h(v+1)} = {@unalloc}

Given this semantics we can define safe as simply not faulting

C,s,h safe ⇔ ¬(C,s,h →* fault)

Long live the Frame rule!

$$\frac{\Gamma \vdash \{ P \}\ C\ \{ Q \}}{\Gamma \vdash \{ P * R \}\ C\ \{ Q * R \}}$$

Provided free variables of R are not in mod(C).

Due to the semantics of assertions, we know anything not described in the pre-condition will be unchanged by the command: pre-conditions describe all the state that the program will access. This leads to the most important rule of separation logic, the frame rule.

We still use standard Hoare logic reasoning for variables, but for the heap we use the new connective * to decide what will remain unchanged in the heap.

[Note there is work on using * to reason about variables: see Bornat, Calcagno, Yang: Variables as Resource in Separation Logic; and Parkinson, Bornat, Calcagno: Variable as Resource in Hoare Logic. These works remove the side-condition on the variables at the expense of complicating the logic to have assertions about “variable ownership”.]

Heap read

We can present the heap read rule it two ways, Backwards:

$\{ \exists Y.\ P[x:=Y] \wedge (E \mapsto Y * \text{true}) \}\ \ x := [E]\ \ \{ P \}$

Small axiom:

$\{ E \mapsto Y \wedge E = Z \}\ x := [E]\ \{ x = Y \wedge Z \mapsto Y \}$

The first formulation takes an arbitrary post-condition and gives you what the pre-condition must have been. Most of the rules give so far have been in this form.

However, in separation logic it is possible to give rules in a different form, small axioms. A small axiom says just what the command uses. In the first, the P can describe any amount of heap, where as with the second, we just describe the “footprint” of the command.

The small axiom for heap read is one of the least elegant rules as it has to deal with expression and variable manipulation. The rest are much nicer.

Aside: small axiom for assignment

In standard Hoare logic we can give a "small" axiom for assignment. Consider the standard axiom

(1) $\{ P [x := E] \}\ x := E \{ P \}$

If we tried to write this in the small axiom style it would be

(2) $\{ Y = E \}\ x := E \{ x = Y \}$

where Y is a logical variable.

We can see this is trivially derived from (1), the converse is also possible

$\{ P [x := E] \}$
$\{ \exists\ Y.\ \ Y=E \wedge P [x := Y] \}$
  $\{ Y=E \wedge P [x := Y] \}$
    $\{ Y=E \}$
      x := E
    $\{ Y=x \}$
  $\{ Y=x \wedge P [x := Y] \}$
$\{ \exists Y.\ Y=x \wedge P [x := Y] \}$
$\{ P \}$

Note that, P [x := Y] cannot mention x hence it is valid to preserve it with the invariance rule.

We can also derive the small axiom from the standard. Exercise, perform this derivation.

Rules for commands

$\{ E \mapsto \_ \}\ [E] := E' \{ E \mapsto E' \}$

$\{ \text{empty} \}\ x := \text{new}\ \ \{ x \mapsto 0 * x+1 \mapsto 0 \}$

$\{ x \mapsto \_\ *\ \ x+1 \mapsto \_\ \}$ dispose x $\{ \text{empty} \}$

Example verification

{empty}
x := new;
[x] := x+1;
[x+1] := x;
while x != 0 do
    x := [x]
{false}

Here we have a similar verification to we saw earlier. Let us proceed to do it using separation logic
The initialisation phase can be verified as
{ empty }
  x := new
{ x ↦ 0 * x+1 ↦ 0 }
   { x ↦ 0 }
  [x] := x+1
   { x ↦ x+1 }
{ x ↦ x+1 * x+1 ↦ 0 }
   { x+1 ↦ 0 }
   [x+1] := x
   { x+1 ↦ x }
{ x ↦ x+1 * x+1 ↦ x}
{ ∃y. x ↦ y * y ↦ x }

Example verification

{empty}
x := new;
[x] := x+1;
[x+1] := x;
while x != 0 do
    x := [x]
{false}

Then we get to the loop, we will assume the loop invariant is
    ∃y. x ↦ y * y ↦ x

{ ∃y. x ↦ y * y ↦ x ∧ x!=0}
{ ∃z. ∃y. x ↦ y * y ↦ z ∧ z=x }
  { x ↦ y * y ↦ z ∧ z=x }
    { x ↦ y ∧ z=x }
    x := [x]
    { z ↦ y ∧ y=x }
  { z ↦ y ∧ y=x * y ↦ z }
{ ∃z. ∃y. z ↦ y * y ↦ z ∧ y=x }
{ ∃z. z ↦ x * x ↦ z }

Exercises

Verify

```
{ empty }
x := new;
y := new;
while y ≠ x do
   dispose y;
   y := new
{ false }
```


Verify, or say why it is not possible to prove

```
{ empty }
x := new;
x := new;
dispose x
{ empty }
```

and

```
{ empty }
x := new;
dispose x;
dispose x
{ empty }
```

Lists and trees

In this lecture, we will represent recursive data structures with recursively defined predicates in separation logic.

List

We use the following axiom

$$list(x) \Leftrightarrow \exists y,v.\ x \mapsto y,v * (y=0 \vee list(y))$$

Does this allow cycles?

This predicate does not describe what the data contained in the list it, just that it is the shape of a list.

We use
$x \mapsto y0,\ldots,yn$ as a shorthand for $x \mapsto y0 * \ldots * x \mapsto yn$

In the early examples for encoding the heap we had difficulties representing a list. Firstly, we had to know it was acyclic. Can this list be cyclic?

First, we can prove
$list(y) \Rightarrow y \neq 0$

Prove
$list(y) \wedge y=0 \Rightarrow false$

By definition LHS is equivalent to
$(\exists x,v.\ y \mapsto x,v * (\ldots)\ ) \wedge y=0$

As $y \mapsto \_$ implies $y \neq 0$, this is unsatisfiable, therefore our initial implication holds.

Now, consider a one element cyclic list
$\exists v.\ x \mapsto x,v$

Does this satisfy the list(x) predicate? That is,
$\exists v.\ x \mapsto x,v \ \Rightarrow\ list(x)$

Let us assume it does and derive a contradiction. By the definition of list we know
$(\exists v.\ x \mapsto x,v) \ \Rightarrow\ \exists y,v.\ x \mapsto y,v * (y=0 \vee list(y))$

If this holds, then it must also be the case that
$empty \wedge x \neq 0 \ \Rightarrow\ x=0 \vee list(x)$

but this cannot hold as empty does not imply list(x). So the original assumption that a one element cyclic list satisfies list(x) must be wrong.

If we look at the models that satisfy list(x) we can see they are all acyclic.

Create List

$$\{\ empty\ \}\ create\_list()\ \{\ list(return)\ \}$$

This is basically the specification that we gave early. However, in separation logic this has the right meaning. Calling it twice leads to two distinct lists.

Prove

```
{ empty }
    x := create_list()
    y := create_list()
{ x ≠ y * true }
```

Note that we are using non-empty lists, earlier we specified the set of locations was non-empty.

Reverse list

```
reverse(x) {
  n := [x];
  [x] := 0;
  while n ≠ 0 do
     p := x;
     x := n;
     n := [x];
     [x] := p;
}
```


{ list(x) }
  n := [x];
{ x ↦ n,_ * (n=0 ∨ list(n)) }
  [x] := 0;
{ x ↦ 0,_ * (n=0 ∨ list(n)) }
{ list(x) * (n=0 ∨ list(n)) }
  while n ≠ 0 do
    { list(x) * list(n) }
      p := x;
      x := n;
    { list(p) * list(x) }
      n := [x];
    { list(p) * x ↦ n,_ * (n=0 ∨ list(n)) }
      [x] := p;
    { list(p) * x ↦ p,_ * (n=0 ∨ list(n)) }
    { list(x) * (n=0 ∨ list(n)) }
{ list(x) * (n=0 ∨ list(n)) ∧ n=0 }
{ list(x) }

Append List

```
append(x,y) {
  local n in
  n := x;
  while n ≠ 0 do
     p := n;
     n := [p];
  [p] := y;
}
```


This procedure takes two lists and appends the second onto the end of the first.

{ list(x) * list(y) }
append(x,y)
{ list(x) }

The problem with trying to verify this is, what is the loop invariant?

We always have list(x) and list(y).

We could define the loop invariant as
  (list(x) ∧ (true * list(n))) * list(y)
but it is difficult in separation logic to work with the standard conjunction ∧ on datastructures. [This is a topic of current research]

It would be really useful to be able to describe just a segment of a list.

List segments

$$\text{listseg}(x,z) \Leftrightarrow x=z \vee (x \neq z * \exists y,v.\ x \mapsto y,v * \text{listseg}(y,z))$$

We can express the loop invariant to the previous program as

The loop invariant is
listseg(x,n) * list(n)
∨ listseg(x,p) * p↦0,_ * n=0

Exercise: Prove or find a counter-example to the following

list(x) ⇔ listseg(x,0)

listseg(x,y) * list(y) ⇔ list(x)

listseg(x,y) * listseg(y,z) ⇔ listseg(x,z)

Exercise: Perform the verification on the previous slide.

Tree

We can define a tree predicate as

$$\text{tree}(x) \Leftrightarrow x=0 \vee \exists i,j.\ x \mapsto i,j * \text{tree}(i) * \text{tree}(j)$$

Exercise: Can this admit graphs? Either show how, or give details why not.

Dispose tree

{ tree(x) }disposetree(x) { emp } ⊢
  { tree(x) }
    i := [x];
    j := [x+1];
    disposetree(i);
    disposetree(j);
    dispose x
  { empty }

We can verify this recursive procedure for disposing of a tree. Is it possible to double dispose a locations?

Could we implement this concurrently, where each branch is disposed in parallel with the other?

Parallel Dispose tree

{ tree(x) }disposetree(x) { emp } ⊢
  { tree(x) }
    i := [x+1];
    j := [x];
    disposetree(i) || disposetree(j) || dispose x
  { empty }

Here $C_1 \,||\, C_2$ means perform C_1 and C_2 at the same time.