Program Verification Using Separation Logic

Cristiano Calcagno

Adapted from material by Dino Distefano

Lecture 1

Goal of the course

Study Separation Logic having automatic verification in mind

Learn how some notions of mathematical logic can be very helpful in reasoning about real world programs

```c
void t1394Diag_CancelIrp(PDEVICE_OBJECT DeviceObject, PIRP Irp)
{
  KIRQL               Irql, CancelIrql;
  BUS_RESET_IRP       *BusResetIrp, *temp;
  PDEVICE_EXTENSION   deviceExtension;

  deviceExtension = DeviceObject->DeviceExtension;

  KeAcquireSpinLock(&deviceExtension->ResetSpinLock, &Irql);

  temp = (PBUS_RESET_IRP)deviceExtension;
  BusResetIrp = (PBUS_RESET_IRP)deviceExtension->Flink2;

  while (BusResetIrp) {

    if (BusResetIrp->Irp == Irp) {
      temp->Flink2 = BusResetIrp->Flink2;
      free(BusResetIrp);
      break;
    }
    else if (BusResetIrp->Flink2 == (PBUS_RESET_IRP)deviceExtension) {
      break;
    }
    else {
      temp = BusResetIrp;
      BusResetIrp = (PBUS_RESET_IRP)BusResetIrp->Flink2;
    }
  }

  KeReleaseSpinLock(&deviceExtension->ResetSpinLock, Irql);

  IoReleaseCancelSpinLock(Irp->CancelIrql);
  Irp->IoStatus.Status = STATUS_CANCELLED;
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
} // t1394Diag_CancelIrp
```

A piece of a windows device driver.

Is this correct?

Or at least: does it have basic properties like it won't crash or leak memory?

Today's plan

- Motivation for Separation Logic
- Assertion language
- Mathematical model
- Data structures

Motivations...

Simple Imperative Language

- Safe commands:
  - S::= skip | x:=E | x:=new(E1,...,En)
- Heap accessing commands:
  - A(E) ::= dispose(E) | x:=[E] | [E]:=F

where E is and expression x, y, nil, etc.

- Command:
  - C::= S | A | C1;C2 | if B { C1 } else {C2} | while B do { C }

where B boolean guard E=E, E!=E, etc.

Example Program: List Reversal

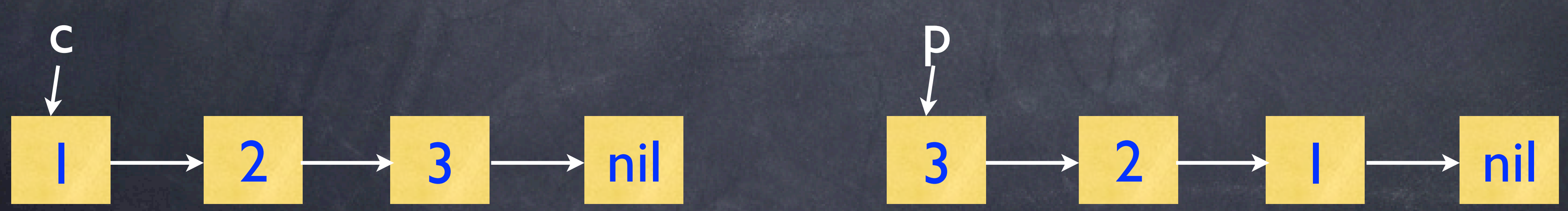
```
p:=nil;
while (c !=nil) do {
  t:=p;
  p:=c;
  c:=[c];
  [p]:=t;
}
```

Some properties we would like to prove:

Does the program preserve acyclicity/cyclicity?

Does it core-dump?

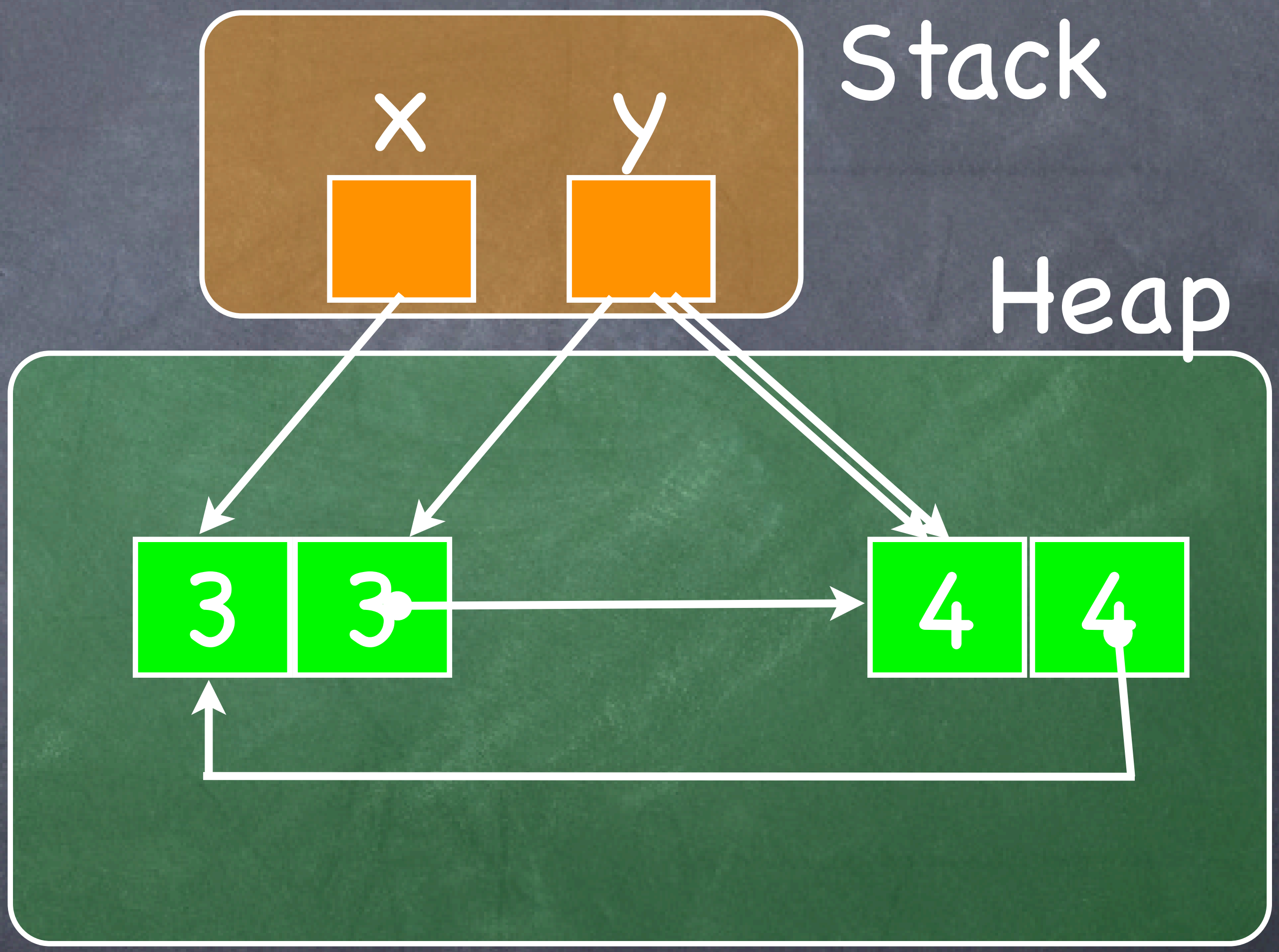
Does it create garbage?



Example Program

We are interested in pointer manipulating programs

```
x = new(3,3);
y = new(4,4);
[x+1] = y;
[y+1] = x;
y = x+1;
dispose x;
y = [y];
```



Why Separation Logic?

Consider this code:

```
Assume([x] = 3)&& x!=y && x!=z)      Add assertion?
Assume(y != z)                       Add assertion?
   [y] = 4;
   [z] = 5;
Guarantee([y] != [z])
Guarantee([x] = 3)
```

We need to know that things are different. How?

We need to know that things stay the same. How?

Framing

We want a general concept of things not being affected.

$$\frac{\{P\}\ C\ \{Q\}}{\{R\ \&\&\ P\}\ C\ \{Q\ \&\&\ R\}}$$

What are the conditions on C and R?

Hard to define if reasoning about a heap and aliasing

This is where separation logic comes in

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

Introduces new connective * used to separate state.

The Logic

Storage Model

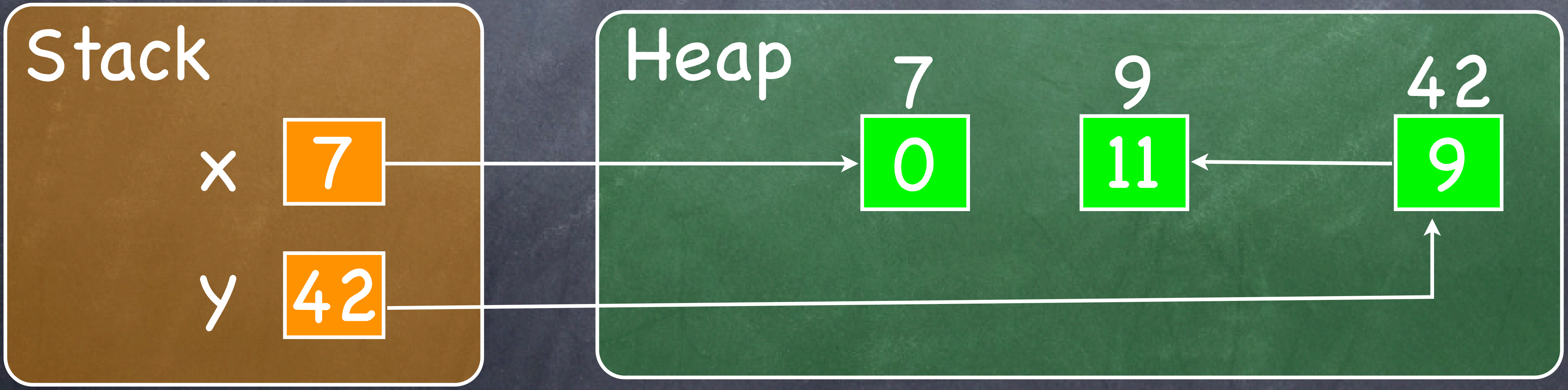
$$\text{Vars} \stackrel{def}{=} \{x, y, z, \ldots\}$$

$$\text{Locs} \stackrel{def}{=} \{1, 2, 3, 4, \ldots\} \qquad \text{Vals} \supseteq \text{Locs}$$

$$\text{Heaps} \stackrel{def}{=} \text{Locs} \rightharpoonup_{\text{fin}} \text{Vals}$$

$$\text{Stacks} \stackrel{def}{=} \text{Vars} \to \text{Vals}$$

$$\text{States} \stackrel{def}{=} \text{Stacks} \times \text{Heaps}$$



Mathematical Structure of Heap

$$\mathsf{Heaps} \stackrel{def}{=} \mathsf{Locs} \rightharpoonup_{\mathsf{fin}} \mathsf{Vals}$$

$$h_1 \# h_2 \stackrel{def}{\Longleftrightarrow} \mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \emptyset$$

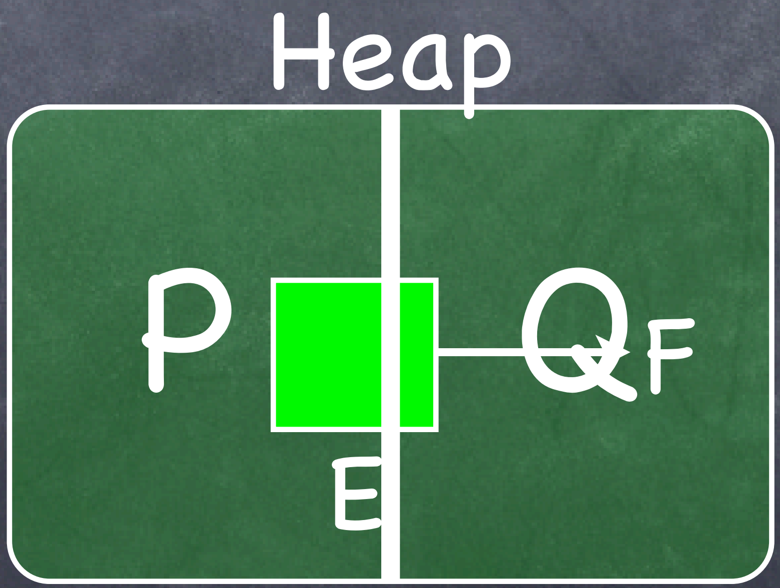
$$h_1 * h_2 \stackrel{def}{=} \begin{cases} h_1 \cup h_2 & \text{if } h_1 \# h_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

1) * has a unit
2) * is associative and commutative
3) (Heap,*,{}) is a partial commutative monoid

Assertions

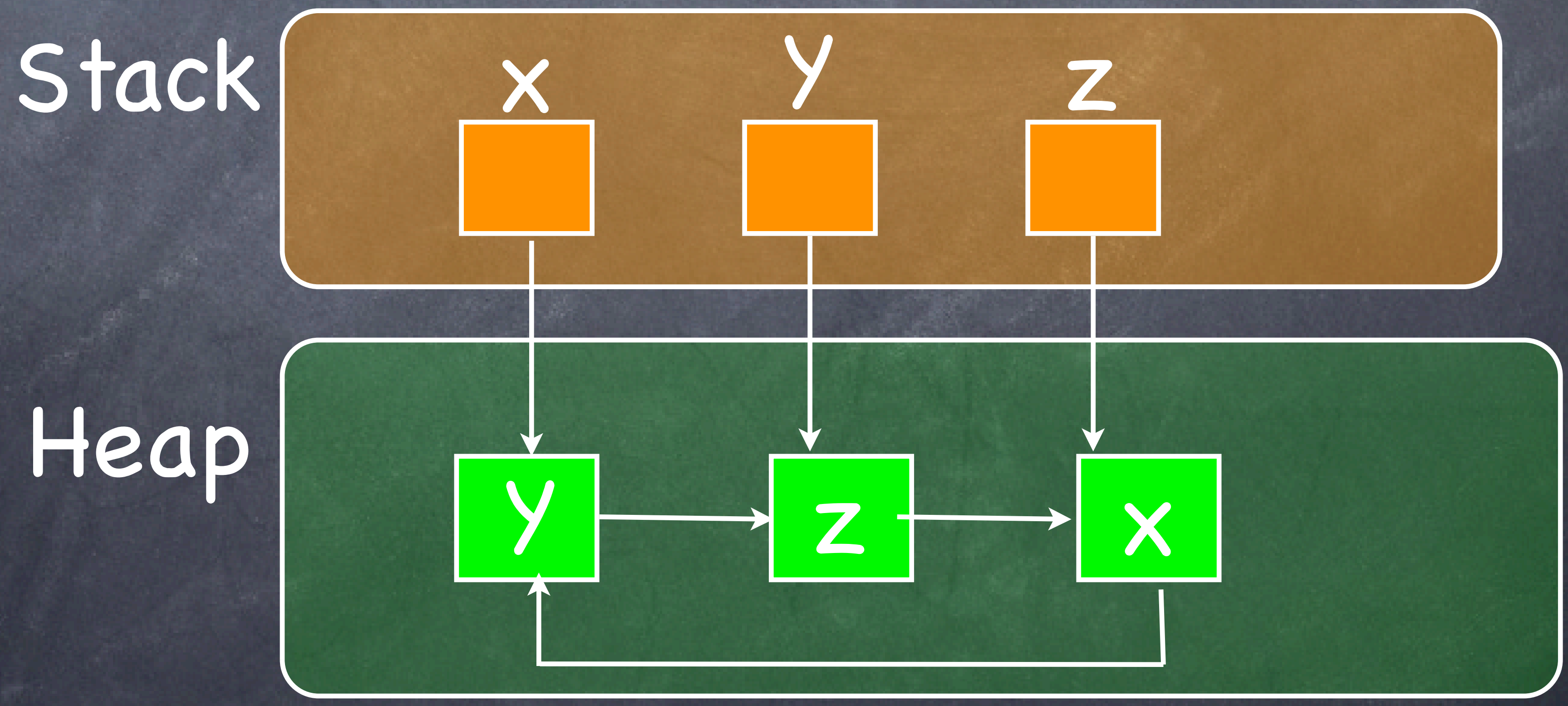
E, F	$::=$	$x \mid n \mid E{+}F \mid -E \mid \ldots$	Heap-independent Exprs
P, Q	$::=$	$E = F \mid E \geq F \mid E \mapsto F$	Atomic Predicates
	$\mid$	$\mathsf{emp} \mid P * Q$	Separating Connectives
	$\mid$	$\mathsf{true} \mid P \wedge Q \mid \neg P \mid \forall x.\, P$	Classical Logic

Informal Meaning



Examples

Formula: emp*x|->y * y|->z* z|->x



Semantics of Assertions

- Expressions mean maps from stacks to integers.

$$[\![E]\!] \quad : \quad \mathsf{Stacks} \longrightarrow \mathsf{Vals}$$

- Semantics of assertions given by satisfaction relation between states and assertions.

$$(s, h) \quad \models \quad P$$

Stack

Heap

Semantics of Assertions

$$
\begin{array}{lll}
(s,h) \models E \geq F & \text{iff} & [\![E]\!]s, [\![F]\!]s \in \mathsf{Integers} \text{ and } [\![E]\!]s \geq [\![F]\!]s \\
(s,h) \models E \mapsto F & \text{iff} & \mathsf{dom}(h) = \{[\![E]\!]s\} \ \text{ and } \ h([\![E]\!]s) = [\![F]\!]s \\
 & & \\
(s,h) \models \mathsf{emp} & \text{iff} & h = [] \ \ (\text{i.e., } \mathsf{dom}(h) = \emptyset) \\
(s,h) \models P * Q & \text{iff} & \exists h_0 h_1.\ \ h_0 * h_1 = h,\ \ (s,h_0) \models P \ \text{ and } \ (s,h_1) \models Q \\
 & & \\
(s,h) \models \mathsf{true} & & \text{always} \\
(s,h) \models P \wedge Q & \text{iff} & (s,h) \models P \ \text{ and } \ (s,h) \models Q \\
(s,h) \models \neg P & \text{iff} & \text{not } \ ((s,h) \models P) \\
(s,h) \models \forall x.\, P & \text{iff} & \forall v \in \mathsf{Vals}.\ (s[x{\mapsto}v], h) \models P)
\end{array}
$$

Abbreviations

The address E is active:

$$E \mapsto - \triangleq \exists x'. E \mapsto x'$$

where x' not free in E

E points to F somewhere in the heap:

$$E \hookrightarrow F \triangleq E \mapsto F * \mathsf{true}$$

E points to a record of several fields:

$$E \mapsto E_1, \ldots, E_n \triangleq E \mapsto E_1 * \cdots * E + n - 1 \mapsto E_n$$

Example

$$x \mapsto 3, y$$

$$y \mapsto 3, x$$

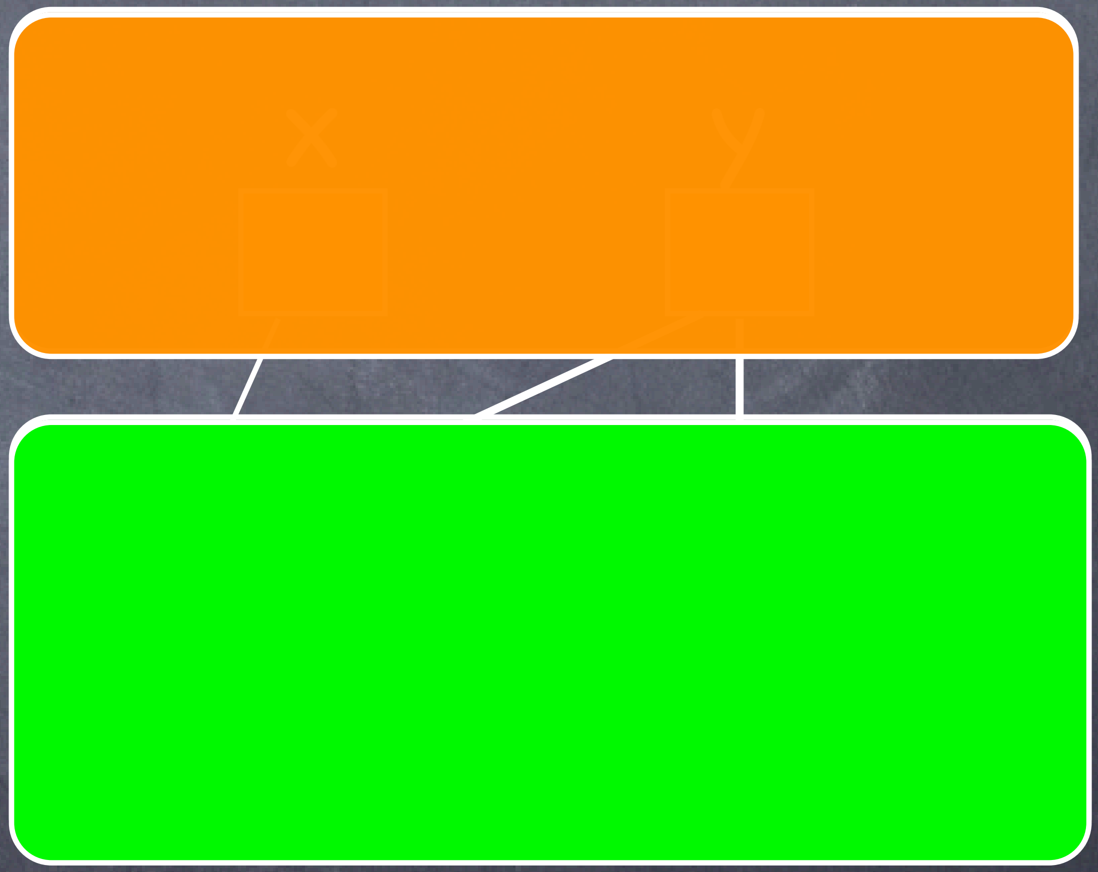
$$x \mapsto 3, y * y \mapsto 3, x$$

$$x \mapsto 3, y \wedge y \mapsto 3, x$$

$$x \hookrightarrow 3, y \wedge y \hookrightarrow 3, x$$

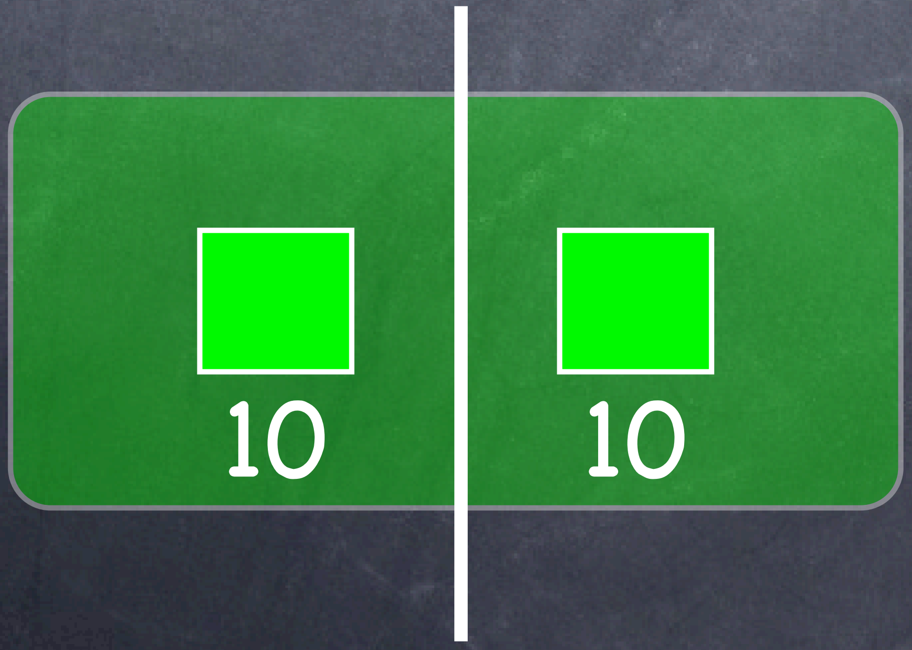
Stack

Heap



An inconsistency

- What's wrong with the following formula?
  - 10|->3 * 10|->3



Try to be in two places at the same time

Small details

- E=F is completely heap independent.

(E=F) *P   where is it true?

In a state where E=F hold in the store and P holds for the same store and a heap contained in the current one

Example: x=y * z|->0   holds in (s,h1) (s,h2)

s(x)=s(y) s(z)=10

dom(h1)={10, 15}   h1(10)=0   h1(15)=37

dom(h2)={10, 42, 73} h2(10)=0 h2(42)=11 h2(73)=0

...but

- E=F is completely heap independent.

(E=F) /\ P   where is it true?

In a state where E=F hold in the store and P holds for the same store and exactly the current heap.

In other words: P determines the heap

Example: x=y /\ z|->0

holds in any state (s,h) such that s(x)=s(y)

dom(h)={s(z)}   h(s(z))=0

so many stores but the shape of the heap is fixed

Exercise

what is h such that s,h|= p

h1={(s(x),1)}

h2={(s(y),2)}

with s(x)!=s(y)

x|->1 h=h1

y|->2 h=h2

x|->1 * y|->2 h=h1 * h2

x|->1 * true h1 contained in h

x|->1 * y|->2 * (x|->1 \/ y|->2) Homework!

Validity

- P is valid if, for all s,h, s,h|=P
- Examples:
  - E|->3 => E>0   Valid!
  - E|-> _ * E|-> _   Invalid!
  - E|-> _ * F |-> _ => E != F   Valid!
  - E |-> 3 /\ F |-> 3 => E=F   Valid!
  - E|->3 * F |->3 => E|->3 /\ F |->3   Invalid!

Some Laws and inference rules

$$p_1 * p_2 \iff p_2 * p_1$$

$$(p_1 * p_2) * p3 \iff p_1 * (p_2 * p_3)$$

$$p * \mathsf{emp} \iff p$$

$$(p_1 \vee p_2) * q \iff (p_1 * q) \vee (p_2 * q)$$

$$(\exists x.p_1) * p_2 \iff \exists x.(p_1 * p_2) \quad \text{when } x \text{ not in } p_2$$

$$(\forall x.p_1) * p_2 \iff \forall x.(p_1 * p_2) \quad \text{when } x \text{ not in } p_2$$

$$\frac{p_1 \implies p_2 \qquad q_1 \implies q_2}{p_1 * q_1 \implies p_2 * q_2}$$ Monotonicity

Substructural logic

- Separation logic is a substructural logic:

No Contraction $A \nvdash A * A$

No Weakening $A * B \nvdash A$

Examples:

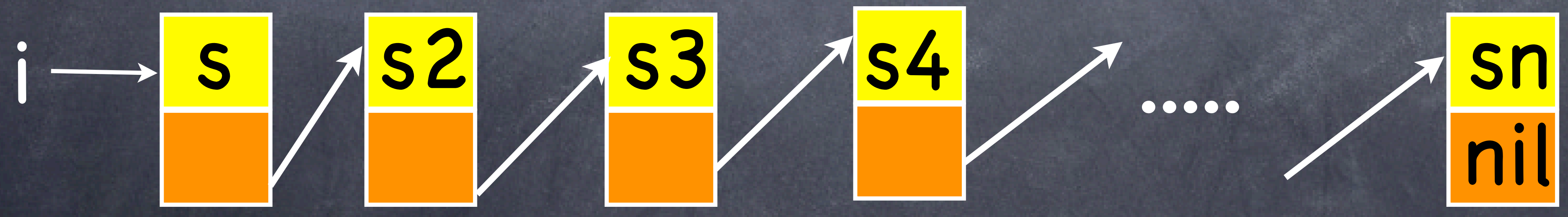
$$10 \mapsto 3 \nvdash 10 \mapsto 3 * 10 \mapsto 3$$

$$10 \mapsto 3 * 42 \mapsto 7 \nvdash 42 \mapsto 7$$

Lists

A non circular list can be defined with the following inductive predicate:

list [] i = emp /\ i=nil
list (s::S) i = exists j. i|->s,j * list S j



List segment

Possibly empty list segment

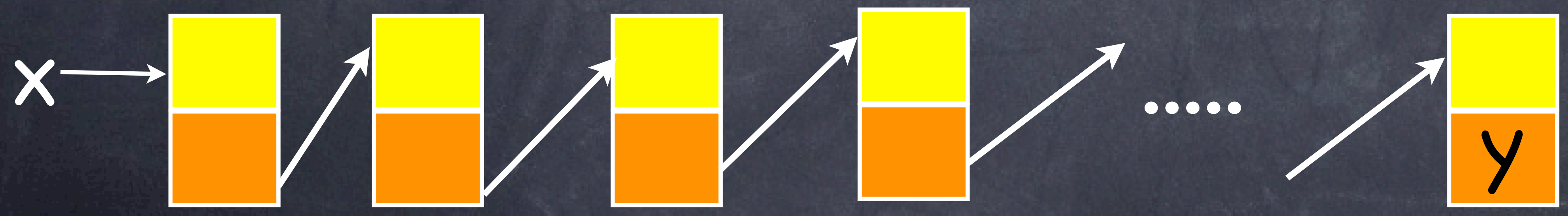
lseg(x,y) = (emp /\ x=y) OR

exists j. x|->j * lseg(j,y)

Non-empty non-circular list segment

lseg(x,y) = x!=y /\

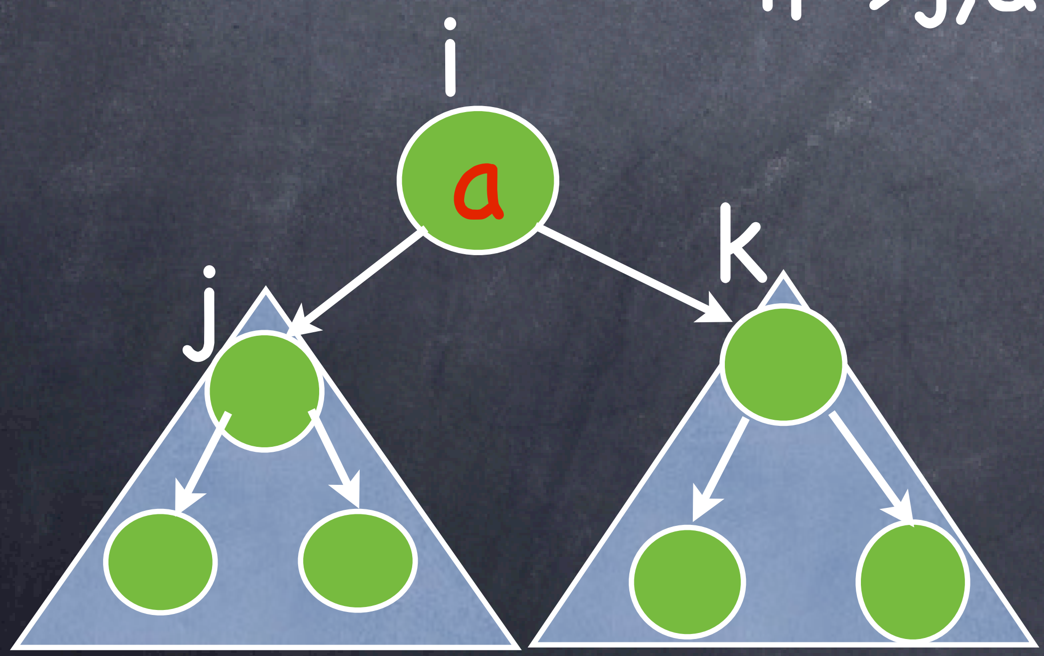
((x|->y) OR exists j. x|->j * lseg(j,y))



Trees

A tree can be defined with this inductive definition:

```
      tree [] i = emp /\ i=nil
tree (t1,a,t2) i = exists j,k.
             i|->j,a,k * (tree t1 j) * (tree t2 k)
```



References

- J.C. Reynolds. Separation Logic: A logic for shared mutable data structures. LICS 2002
- S. Ishtiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. POPL 2001.