Compositional Shape Analysis by means of Bi-Abduction

Dino Distefano
Queen Mary University of London
and
Monoidics Ltd

MOVEP 2010, Aachen 29/06/2010

A lot of real code out there uses pointer manipulation...

```
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
```

Is this correct?

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

We want to build tool that automatically answer such questions

Crash course on Separation Logic

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 }

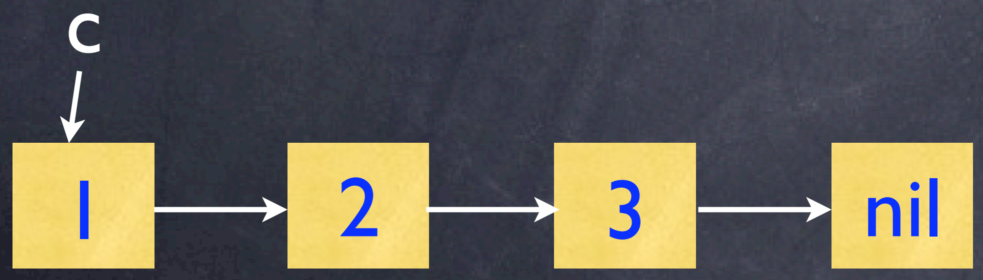
where B boolean guard E=E, E!=E, etc.

Example Program: List Reversal

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

Example Program: List Reversal

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



Example Program: List Reversal

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

Example Program: List Reversal

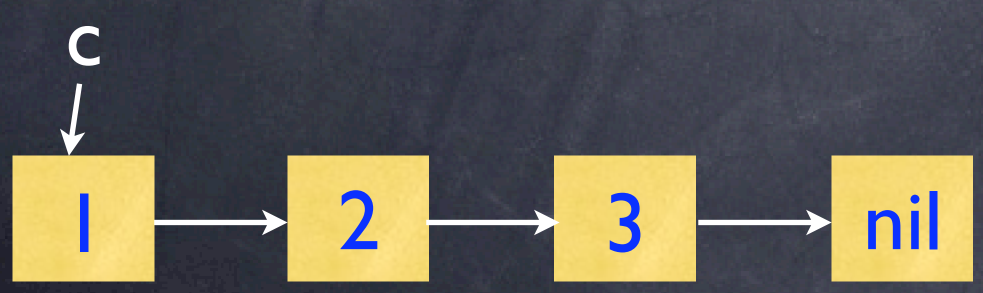
```
p:=nil;
while (c !=nil) do {
  t:=p;
  p:=c;
  c:=[c];
  [p]:=t;
}
```

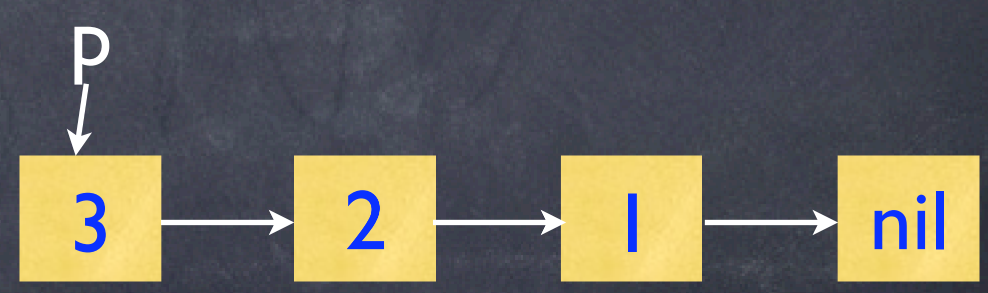
Some properties we would like to prove:

Does the program preserve acyclicity/cyclicity?

Does it core-dump?

Does it create garbage?

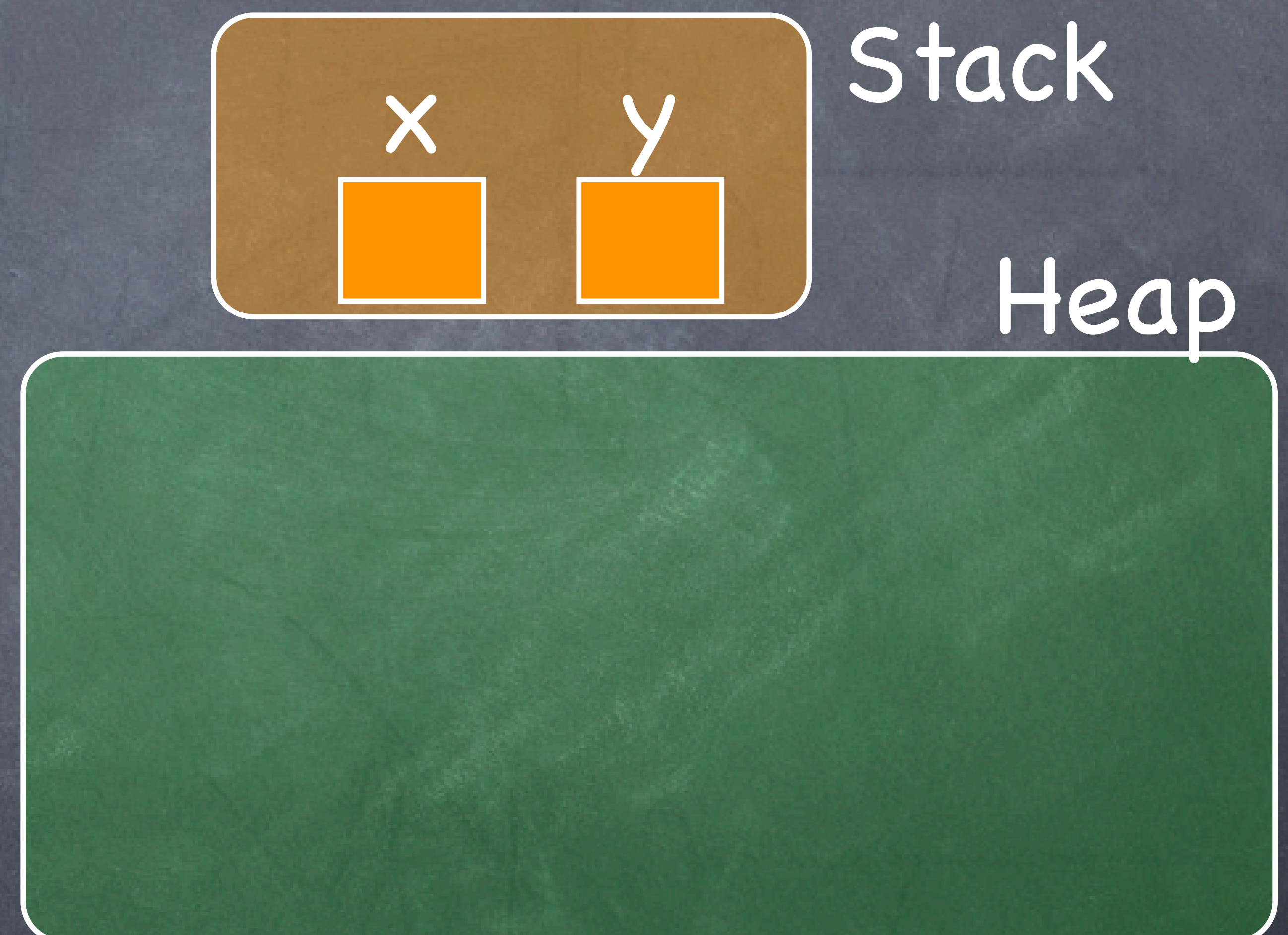




Example Program

We are interested in pointer manipulating programs

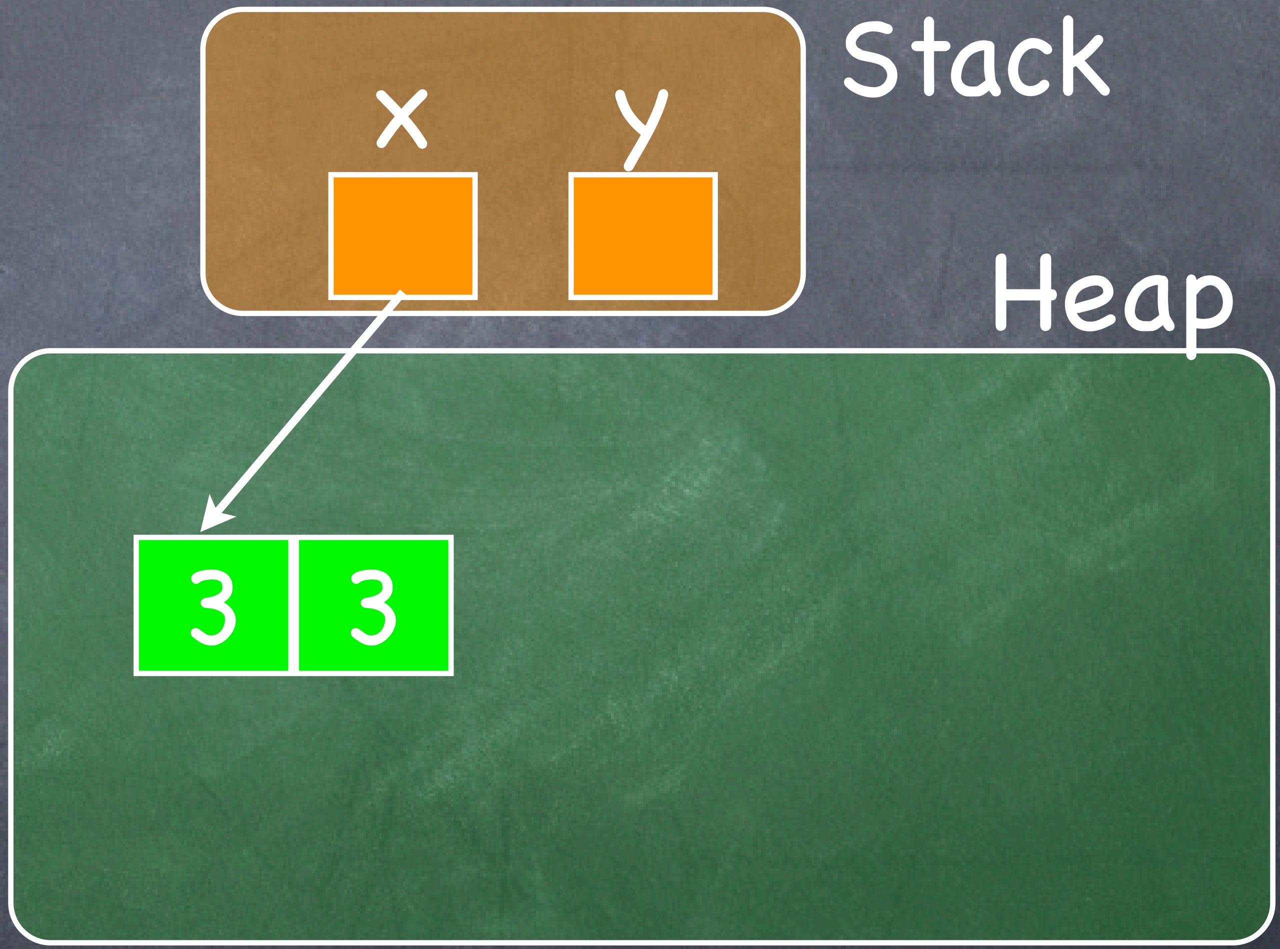
```
x := new(3,3);
y := new(4,4);
[x+1] := y;
[y+1] := x;
y := x+1;
dispose x;
y := [y];
```



Example Program

We are interested in pointer manipulating programs

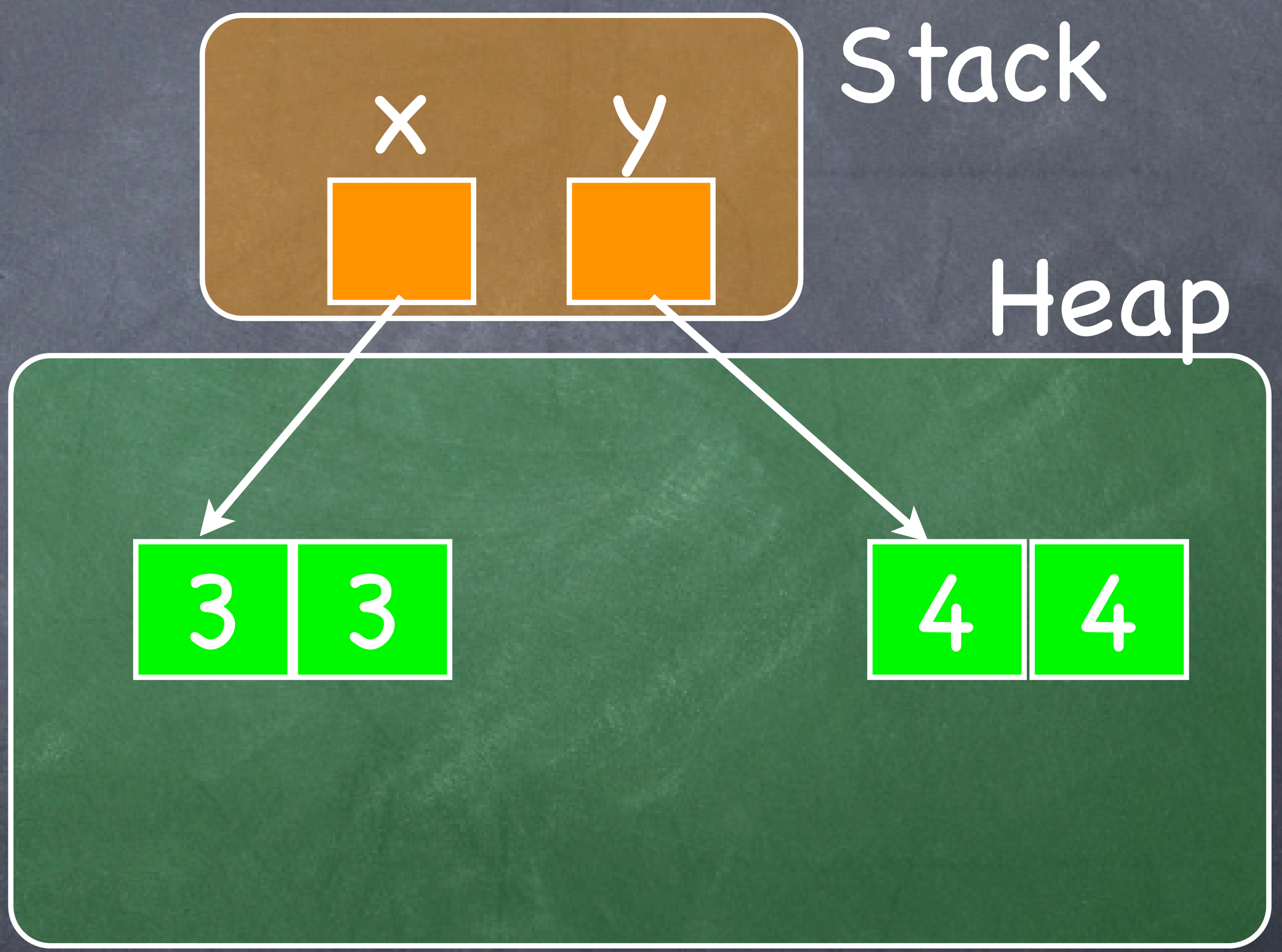
```
x := new(3,3);
y := new(4,4);
[x+1] := y;
[y+1] := x;
y := x+1;
dispose x;
y := [y];
```



Example Program

We are interested in pointer manipulating programs

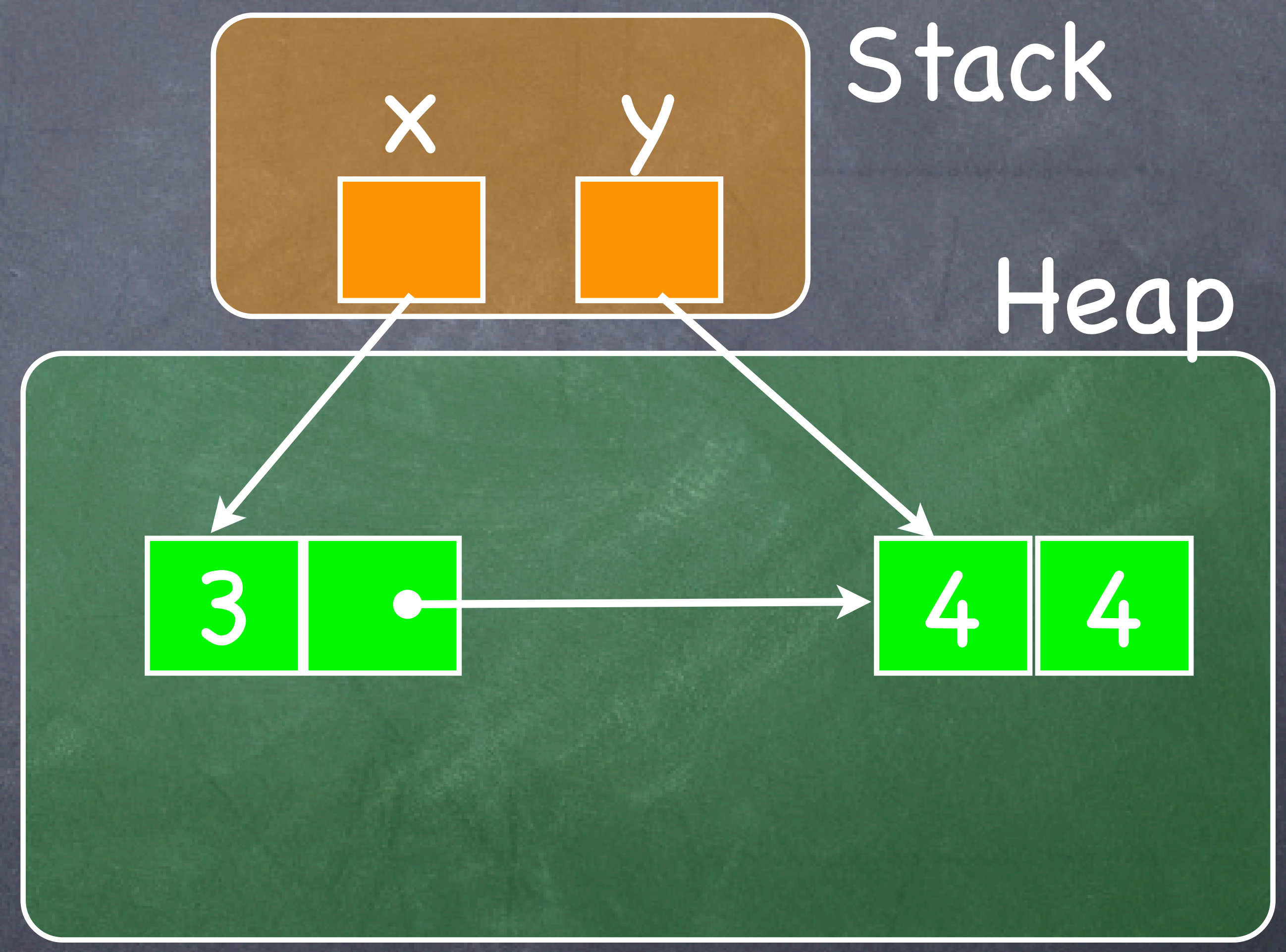
```
x := new(3,3);
y := new(4,4);
[x+1] := y;
[y+1] := x;
y := x+1;
dispose x;
y := [y];
```



Example Program

We are interested in pointer manipulating programs

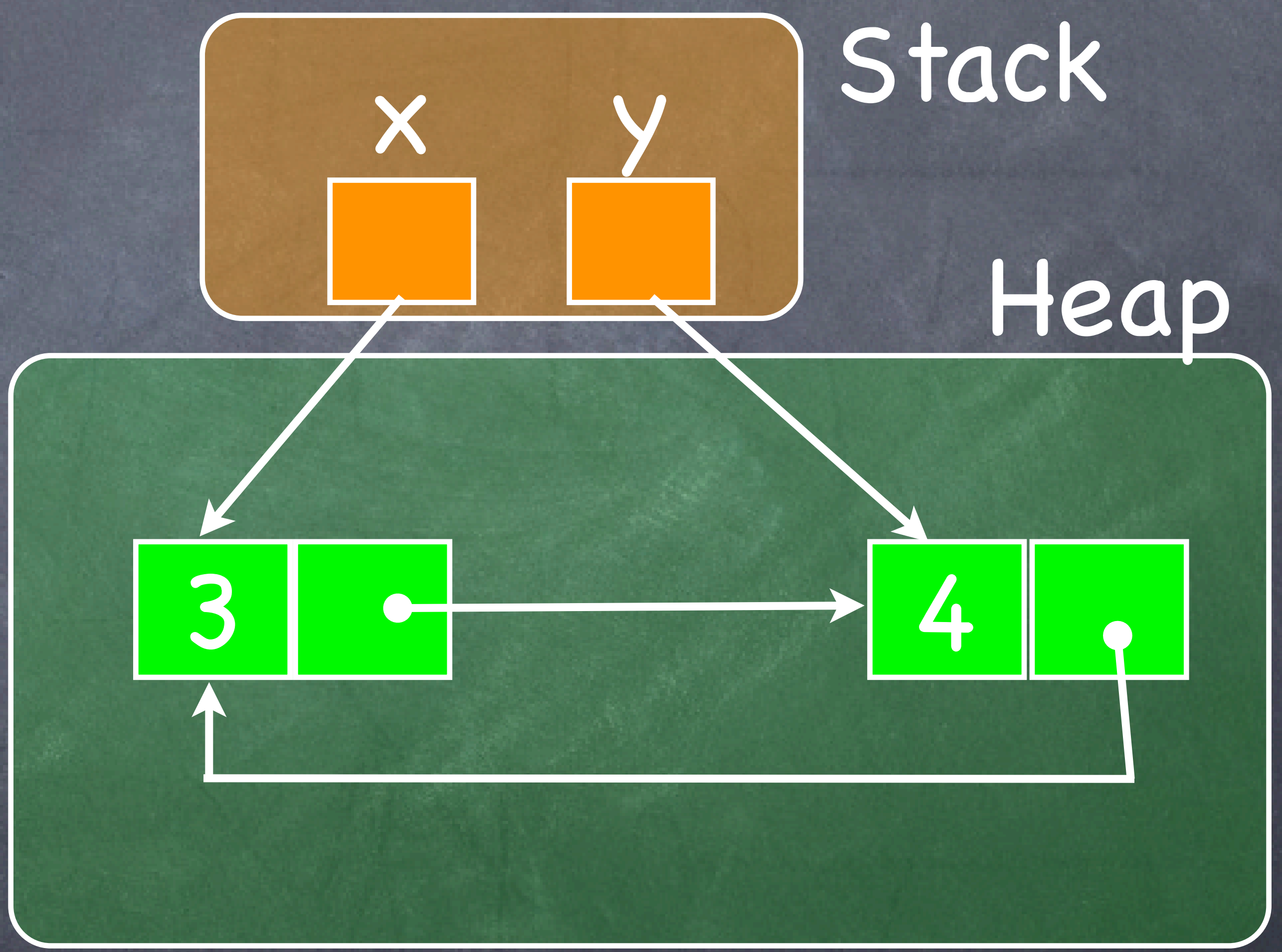
```
x := new(3,3);
y := new(4,4);
[x+1] := y;
[y+1] := x;
y := x+1;
dispose x;
y := [y];
```



Example Program

We are interested in pointer manipulating programs

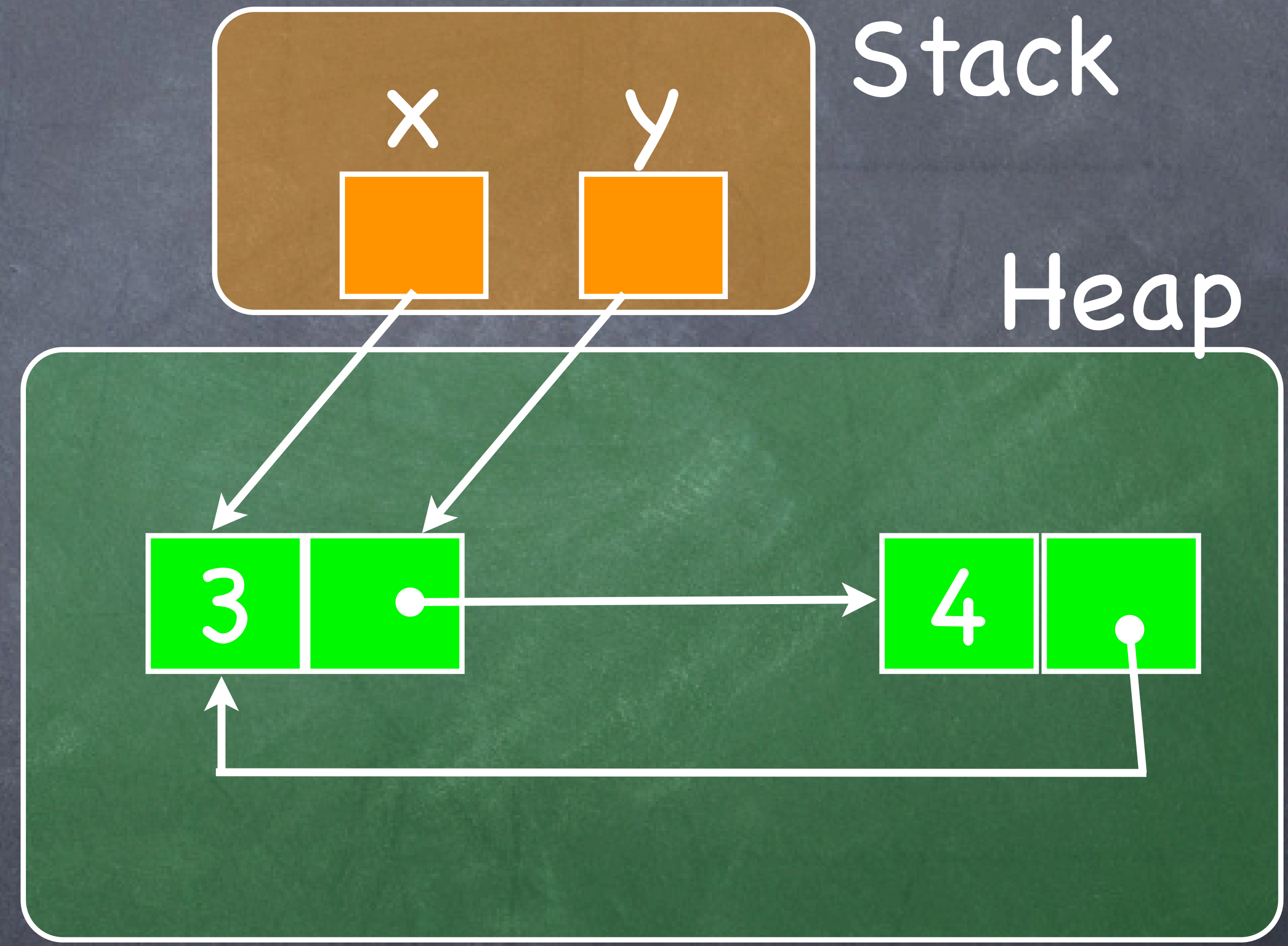
```
x := new(3,3);
y := new(4,4);
[x+1] := y;
[y+1] := x;
y := x+1;
dispose x;
y := [y];
```



Example Program

We are interested in pointer manipulating programs

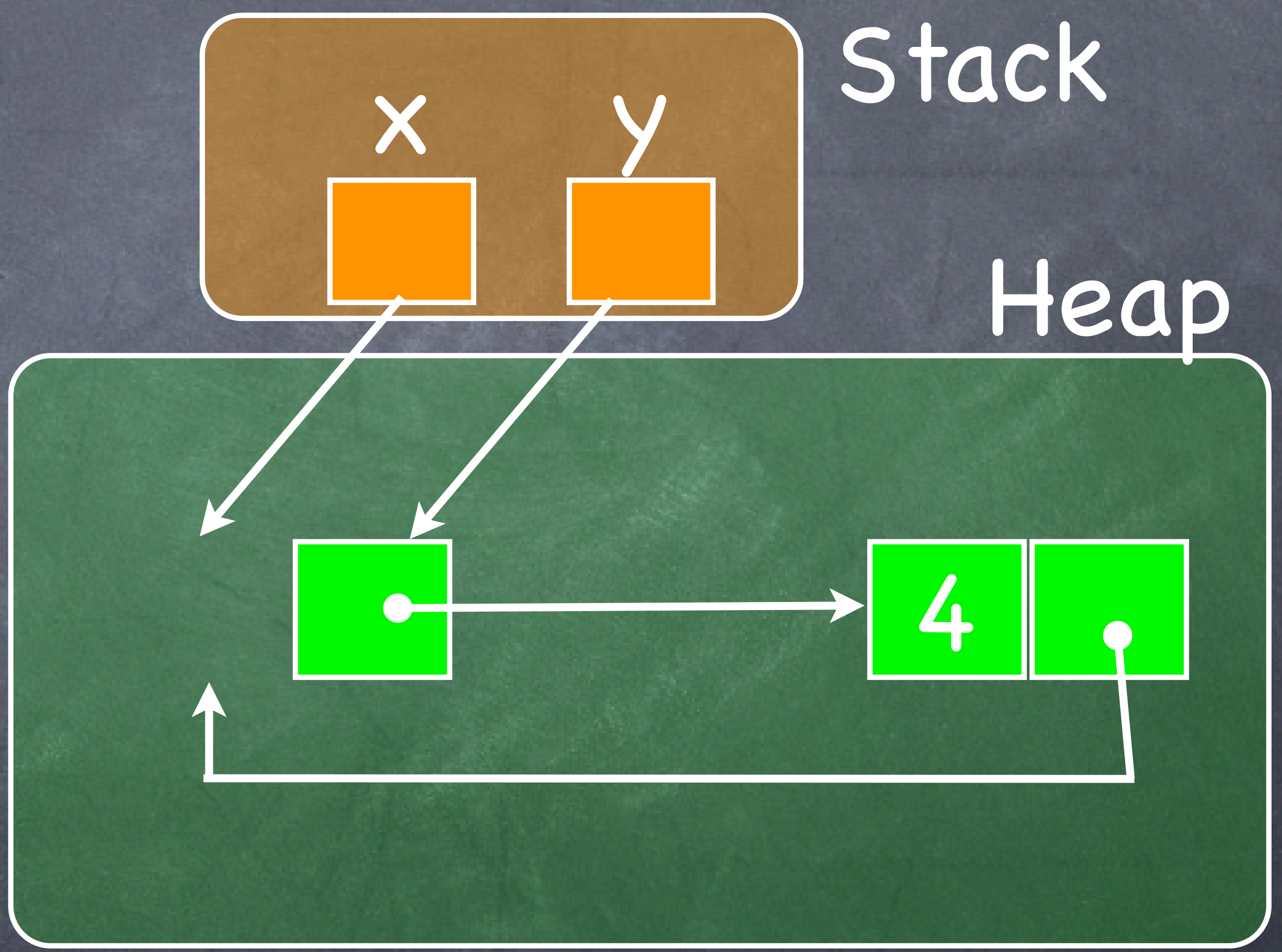
```
x := new(3,3);
y := new(4,4);
[x+1] := y;
[y+1] := x;
y := x+1;
dispose x;
y := [y];
```



Example Program

We are interested in pointer manipulating programs

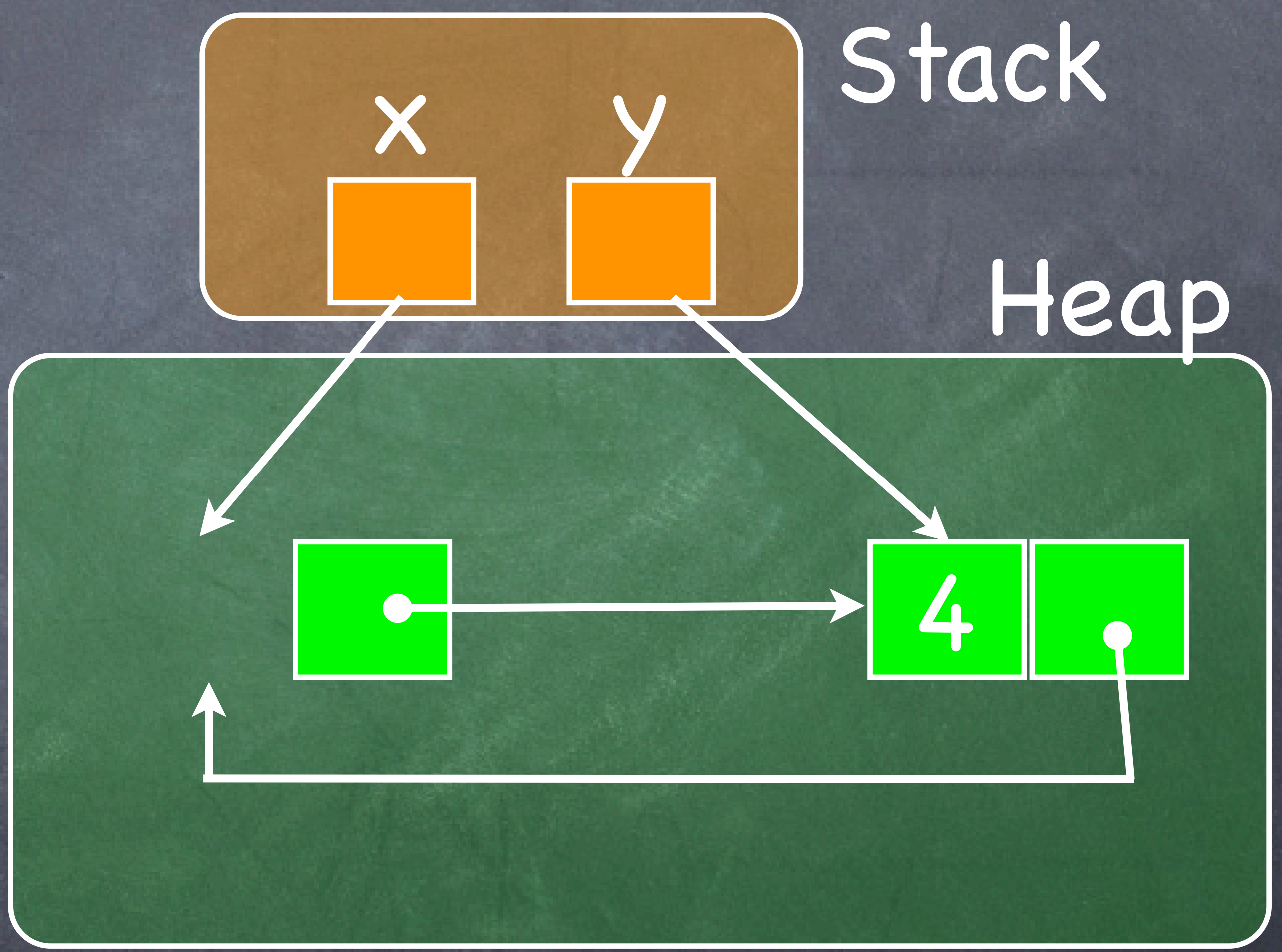
```
x := new(3,3);
y := new(4,4);
[x+1] := y;
[y+1] := x;
y := x+1;
dispose x;
y := [y];
```



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:

```
[y] := 4;
[z] := 5;
```

Guarantee([y] != [z])

We need to know that things are different. How?

Why Separation Logic?

Consider this code:

Add assertion?

```
Assume(y != z)
  [y] := 4;
  [z] := 5;
Guarantee([y] != [z])
```

We need to know that things are different. How?

Why Separation Logic?

Consider this code:

Assume(y != z)

Add assertion?

```
  [y] := 4;
  [z] := 5;
Guarantee([y] != [z])
```

We need to know that things are different. How?

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

Why Separation Logic?

Consider this code:

```
Assume([x] = 3)
Assume(y != z)
   [y] := 4;
   [z] := 5;
Guarantee([y] != [z])
Guarantee([x] = 3)
```

Add assertion?

We need to know that things are different. How?

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

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

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.

Storage Model

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

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

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

$$\mathsf{Stacks} \stackrel{def}{=} \mathsf{Vars} \rightarrow \mathsf{Vals}$$

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

Storage Model

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

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

$$\begin{aligned} \mathsf{Heaps} &\stackrel{def}{=} \mathsf{Locs} \rightharpoonup_{\mathsf{fin}} \mathsf{Vals} \\ \mathsf{Stacks} &\stackrel{def}{=} \mathsf{Vars} \to \mathsf{Vals} \\ \mathsf{States} &\stackrel{def}{=} \mathsf{Stacks} \times \mathsf{Heaps} \end{aligned}$$

Stack

x 7

y 42

Storage Model

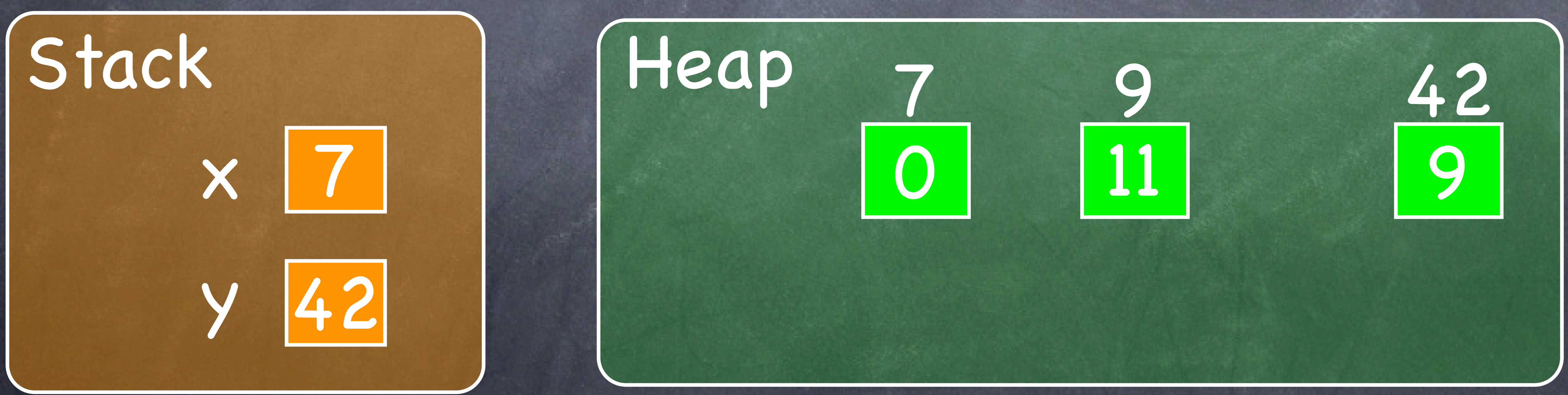
$$\mathsf{Vars} \stackrel{def}{=} \{x, y, z, \ldots\}$$

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

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

$$\mathsf{Stacks} \stackrel{def}{=} \mathsf{Vars} \rightarrow \mathsf{Vals}$$

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

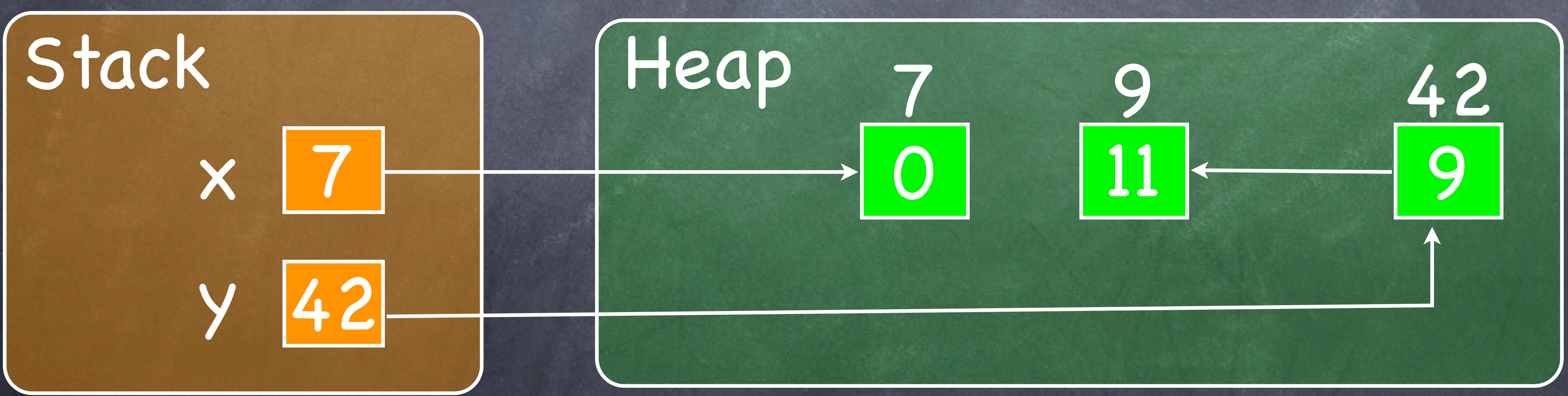


Storage Model

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

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

$$\begin{aligned} \mathsf{Heaps} &\stackrel{\mathit{def}}{=} \mathsf{Locs} \rightharpoonup_{\mathsf{fin}} \mathsf{Vals} \\ \mathsf{Stacks} &\stackrel{\mathit{def}}{=} \mathsf{Vars} \to \mathsf{Vals} \\ \mathsf{States} &\stackrel{\mathit{def}}{=} \mathsf{Stacks} \times \mathsf{Heaps} \end{aligned}$$



Assertions

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

Informal Meaning

Assertions

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

Informal Meaning

Assertions

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

Informal Meaning

Heap

Assertions

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

Informal Meaning

Heap

Assertions

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

Informal Meaning

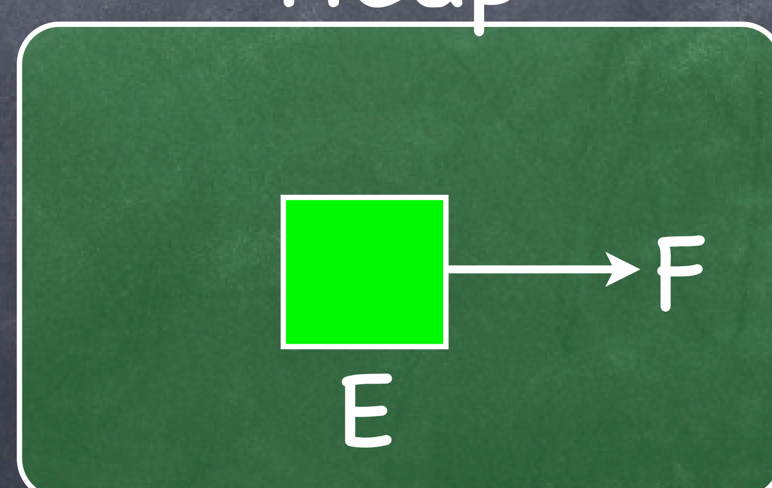
Heap

Assertions

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

Informal Meaning

Heap



Assertions

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

Informal Meaning

Heap

Assertions

$$
\begin{array}{rcl}
E, F & ::= & x \mid n \mid E{+}F \mid -E \mid \ldots \\
P, Q & ::= & E = F \mid E \geq F \mid E \mapsto F \\
 & \mid & \mathsf{emp} \mid \boxed{P * Q} \\
 & \mid & \mathsf{true} \mid P \wedge Q \mid \neg P \mid \forall x.\, P
\end{array}
$$

Heap-independent Exprs

Atomic Predicates

Separating Connectives

Classical Logic

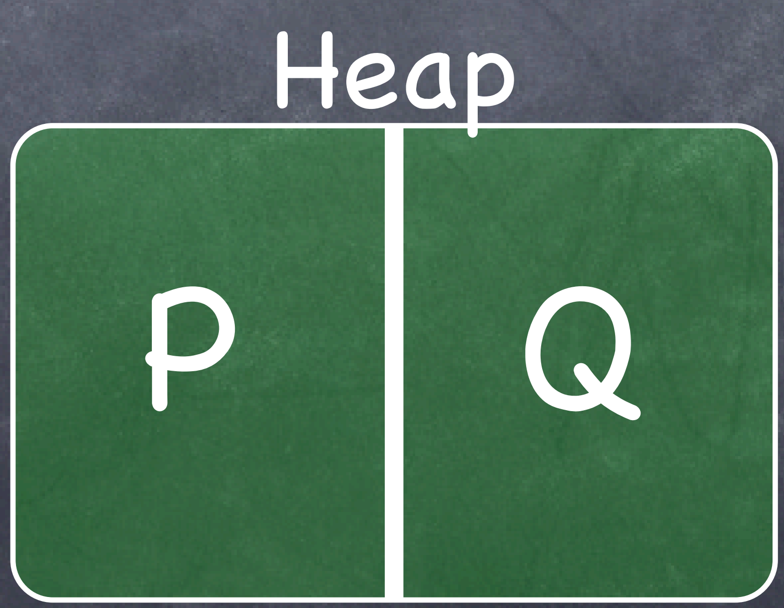
Informal Meaning

Heap

Assertions

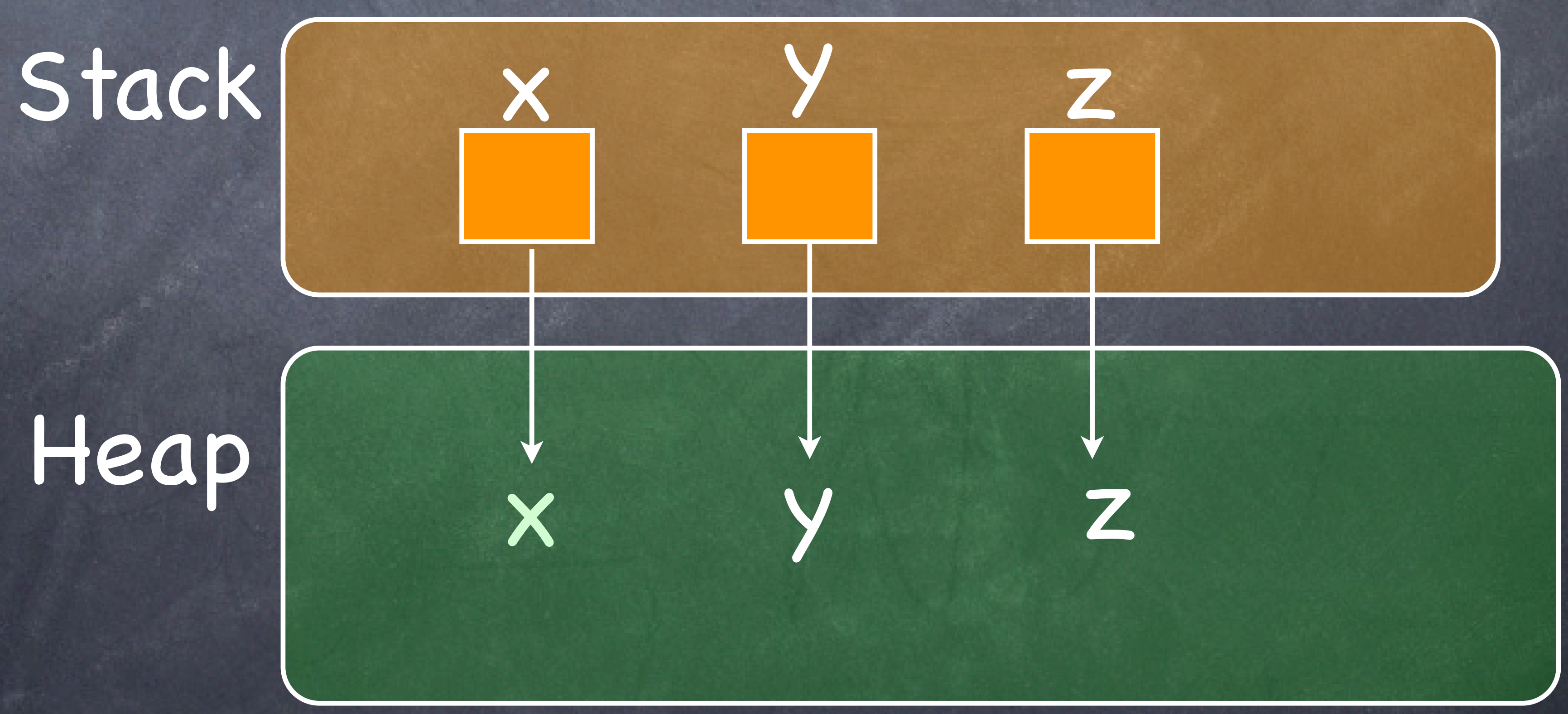
$$
\begin{array}{rclr}
E, F & ::= & x \mid n \mid E{+}F \mid -E \mid \ldots & \text{Heap-independent Exprs} \\
P, Q & ::= & E = F \mid E \geq F \mid E \mapsto F & \text{Atomic Predicates} \\
 & \mid & \mathsf{emp} \mid \boxed{P * Q} & \text{Separating Connectives} \\
 & \mid & \mathsf{true} \mid P \wedge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

Informal Meaning



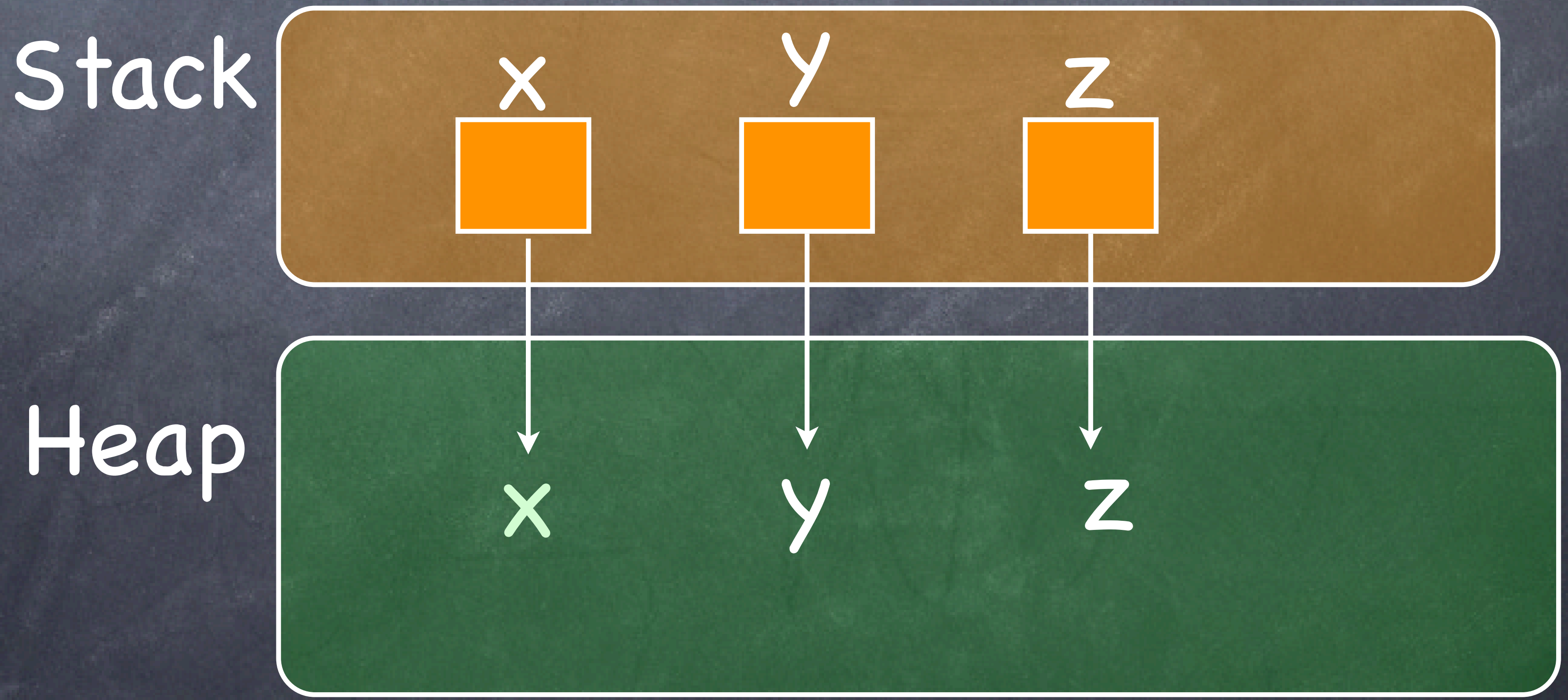
Examples

Formula: emp



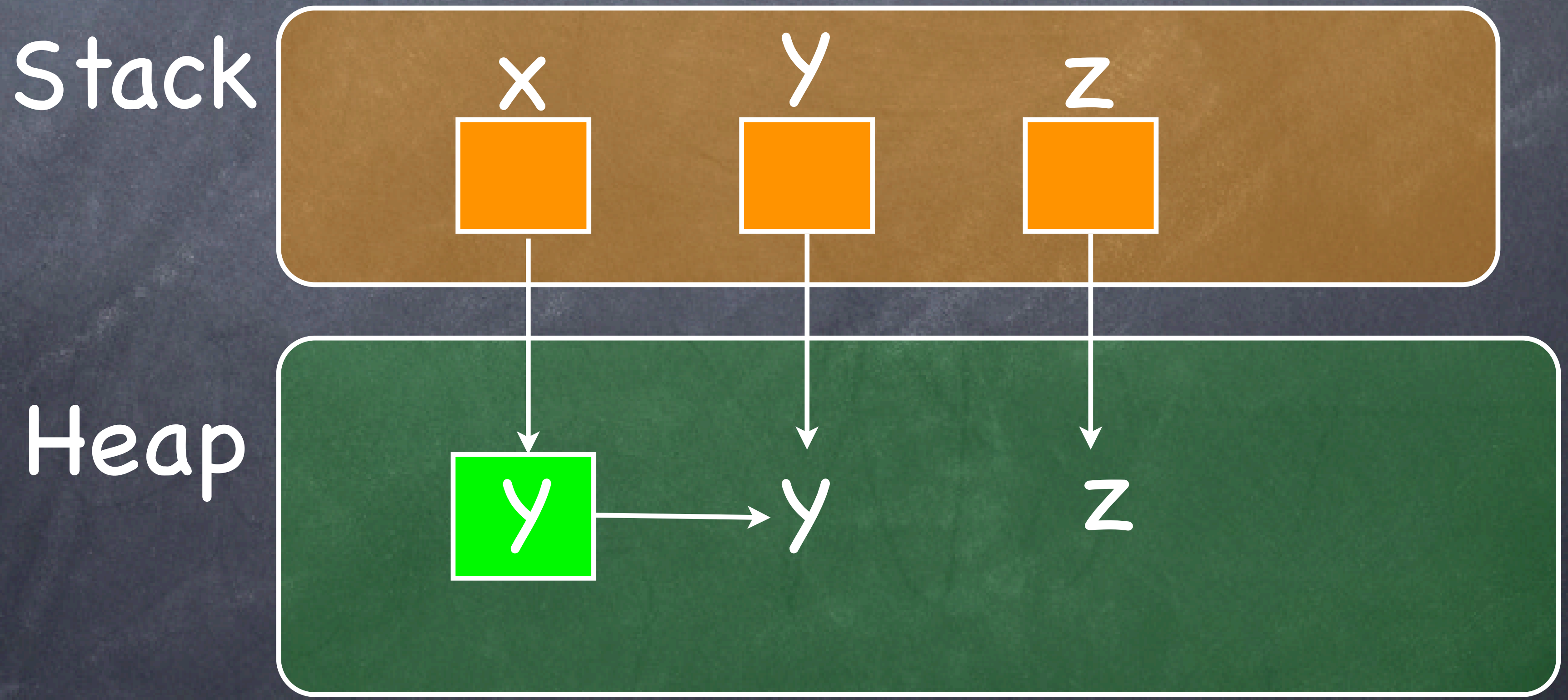
Examples

Formula: emp*x|->y



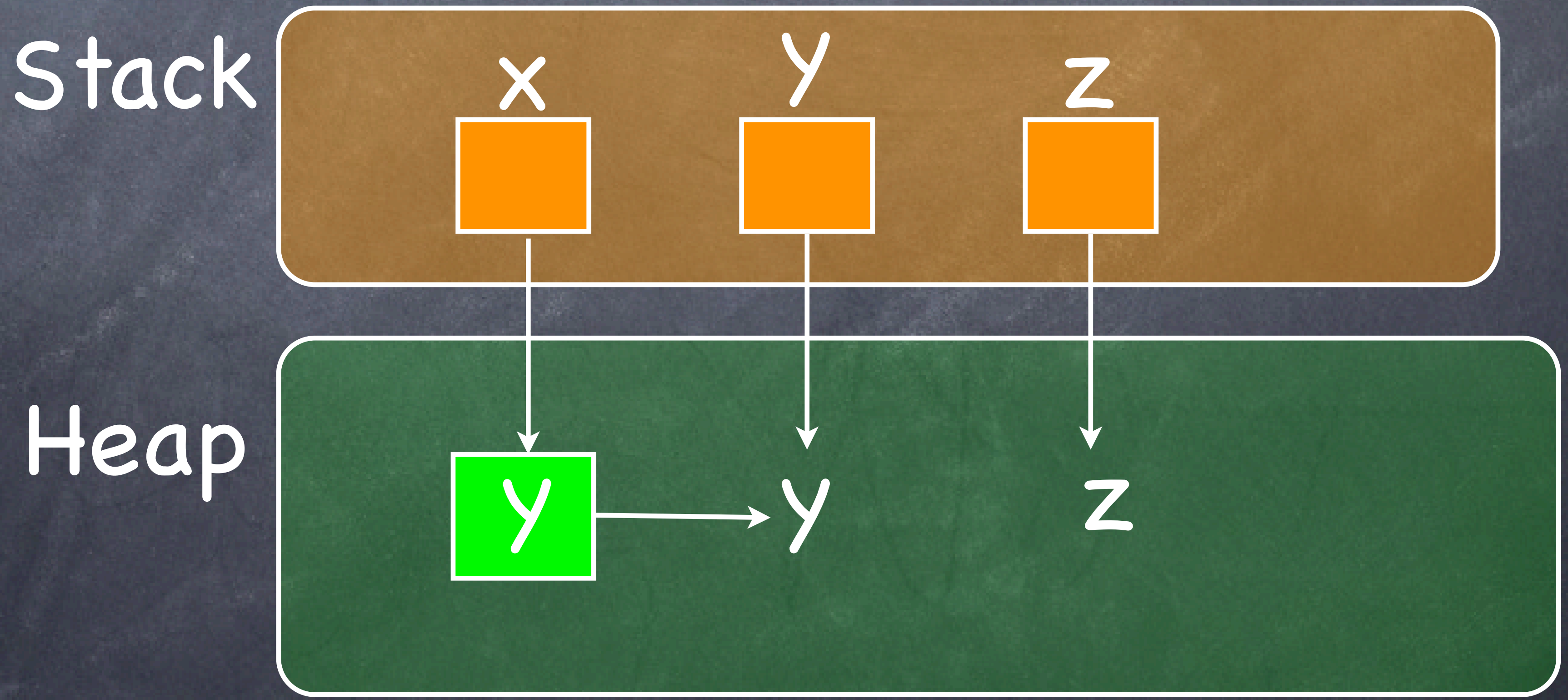
Examples

Formula: emp*x|->y



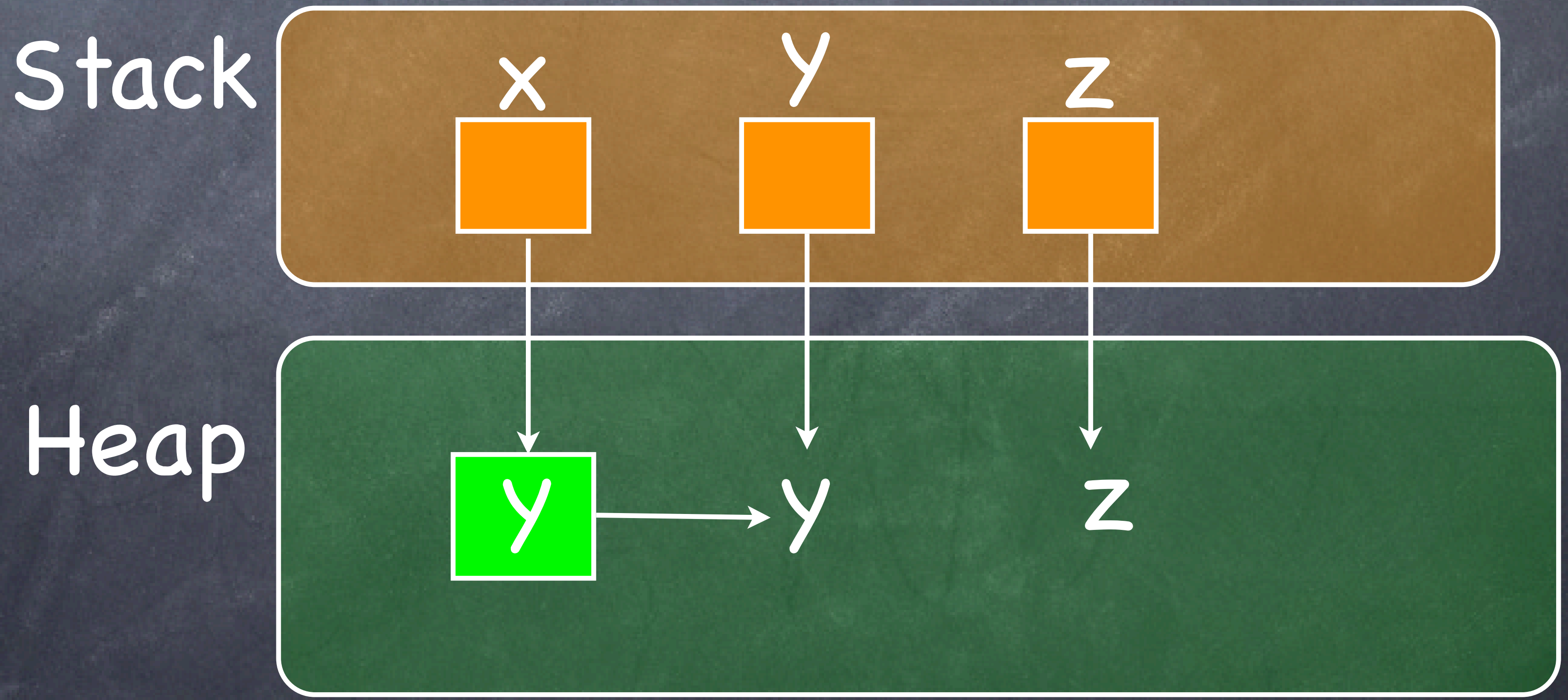
Examples

Formula: x|->y



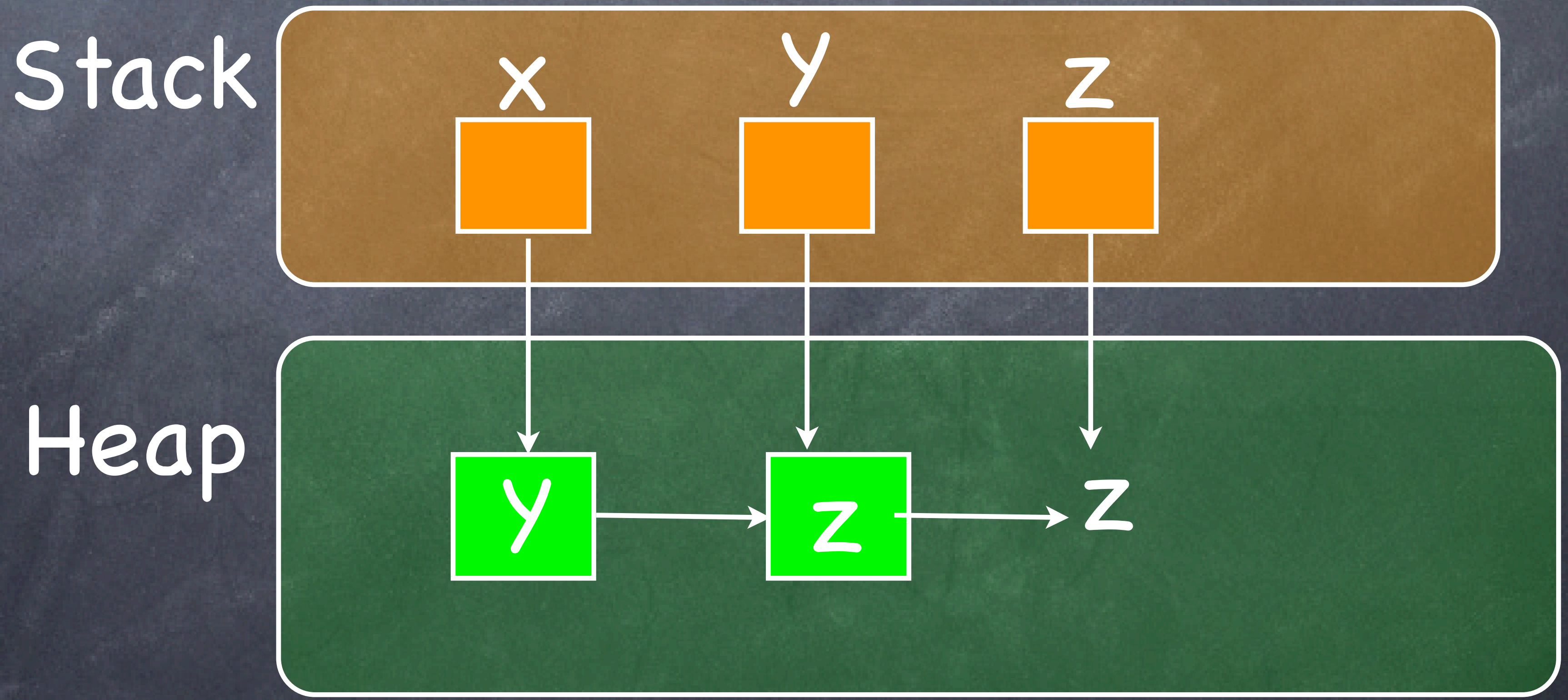
Examples

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



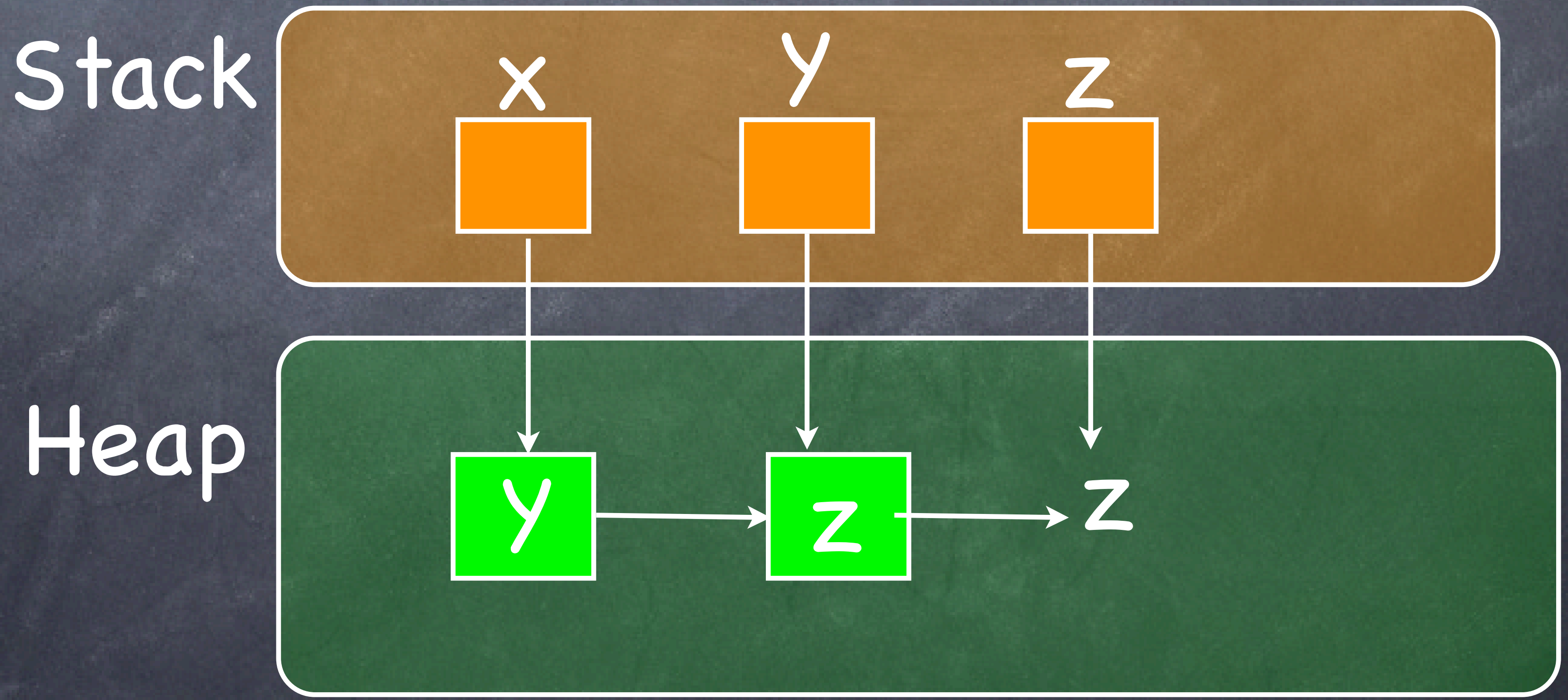
Examples

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



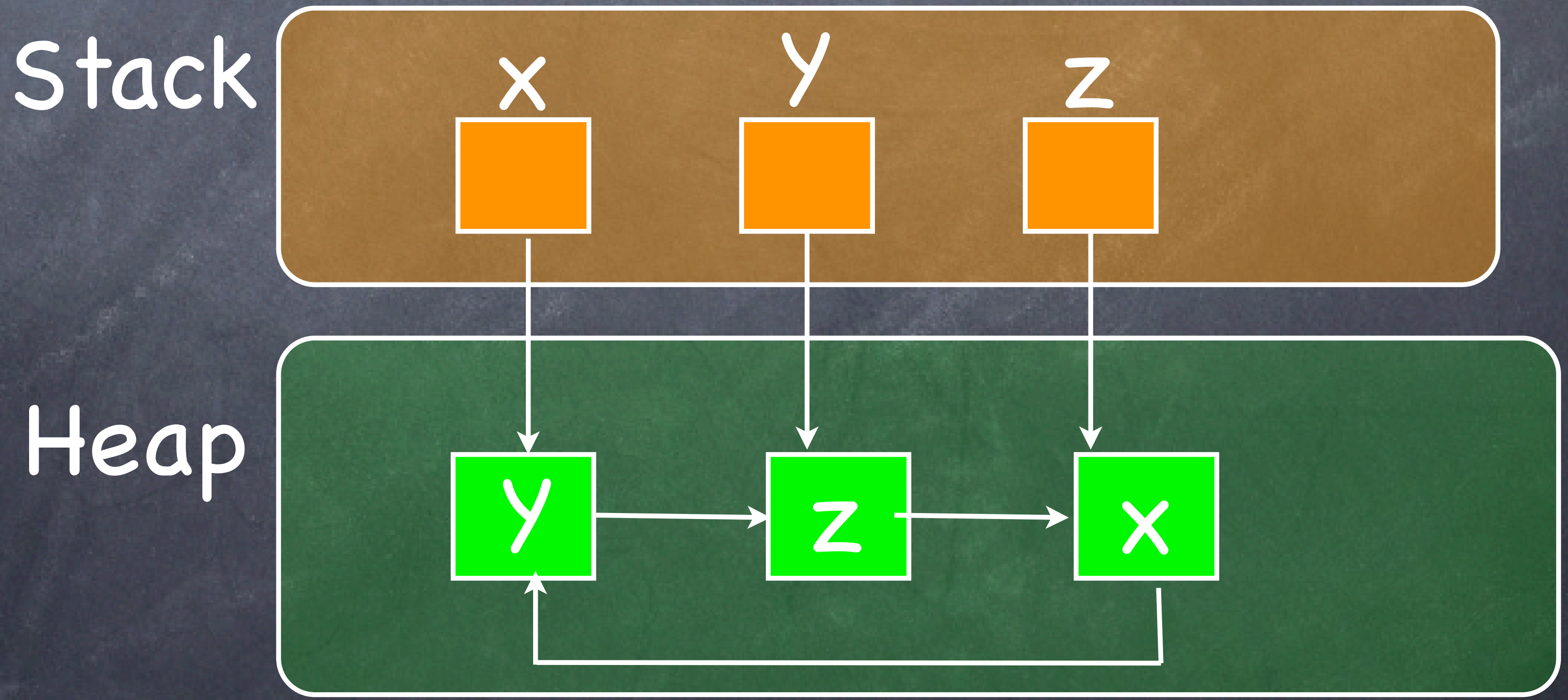
Examples

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



Examples

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



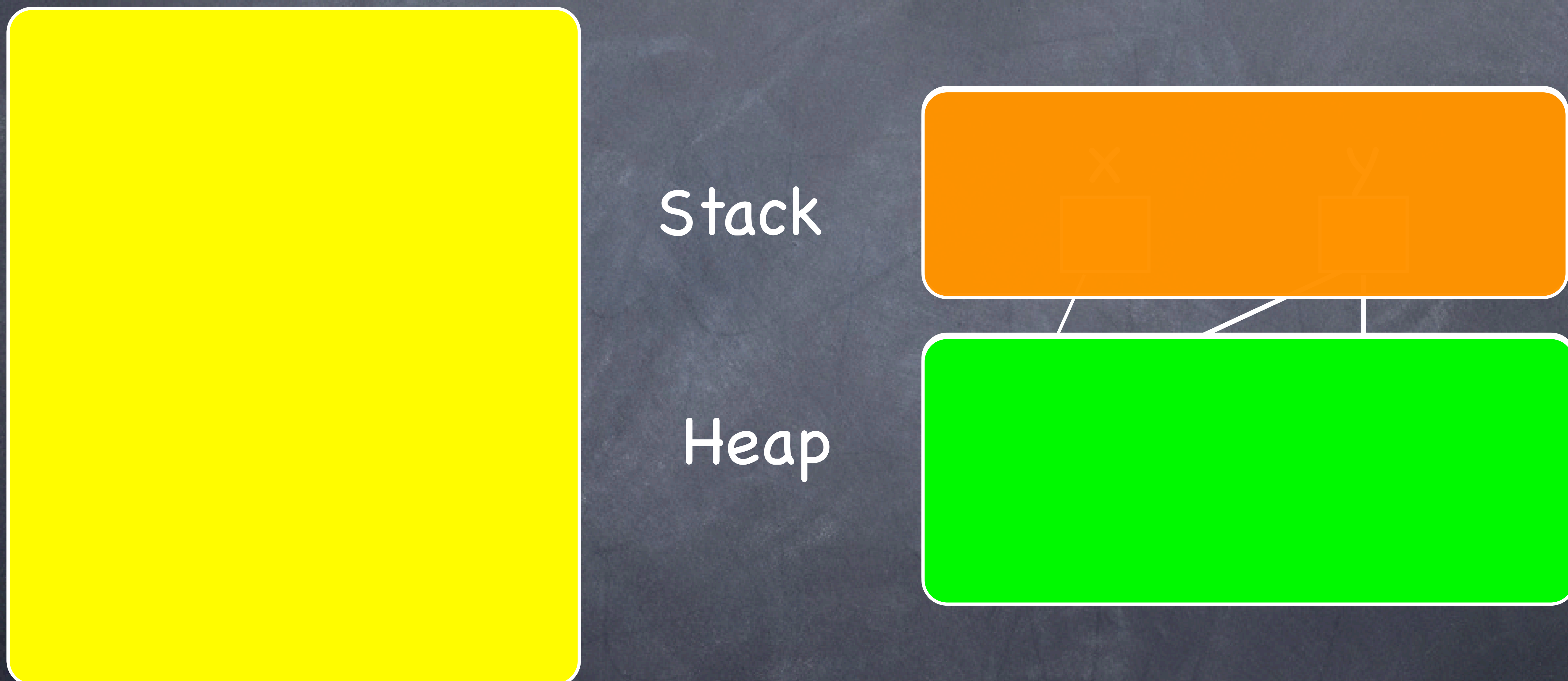
Semantics of Assertions

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

$$
\begin{array}{lll}
s, h \models E \mapsto F & \text{iff} & \mathrm{dom}(h) = \{[\![E]\!]_s\} \text{ and } h([\![E]\!]_s) = [\![F]\!]_s \\
s, h \models \mathsf{emp} & \text{iff} & \mathrm{dom}(h) = \emptyset \\
s, h \models P * Q & \text{iff} & \exists h_0, h_1.\, \mathrm{dom}(h_0) \cap \mathrm{dom}(h_1) = \emptyset \text{ and } h_0 \cdot h_1 = h \\
 & & \text{and } s, h_0 \models P \text{ and } s, h_1 \models Q \\
s, h \models P \wedge Q & \text{iff} & s, h \models P \text{ and } s, h \models Q
\end{array}
$$

where meaning of expressions $[\![E]\!] \; : \; \mathsf{Stacks} \rightarrow \mathsf{Vals}$

Example



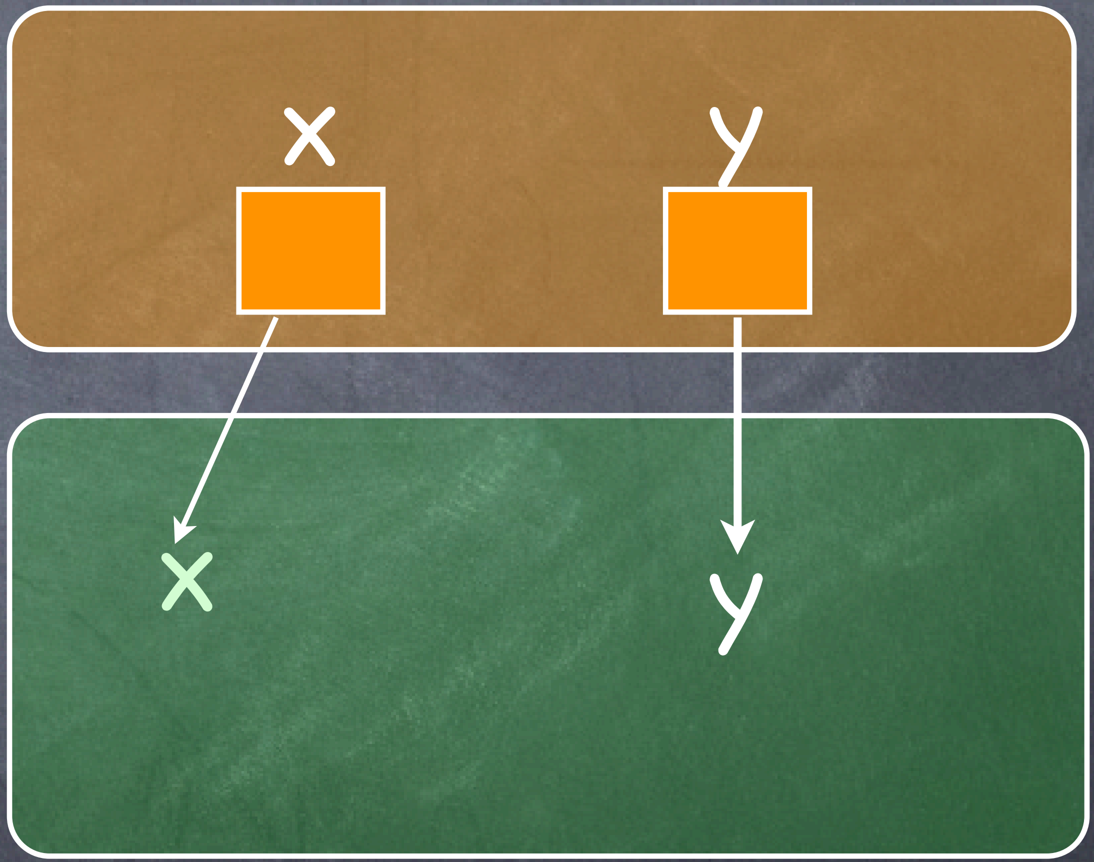
Abbreviation: E points to a record of several fields: $E \mapsto E_1, \ldots, E_n \triangleq E \mapsto E_1 * \cdots * E+n-1 \mapsto E_n$

Example

$x \mapsto 3, y$

Stack

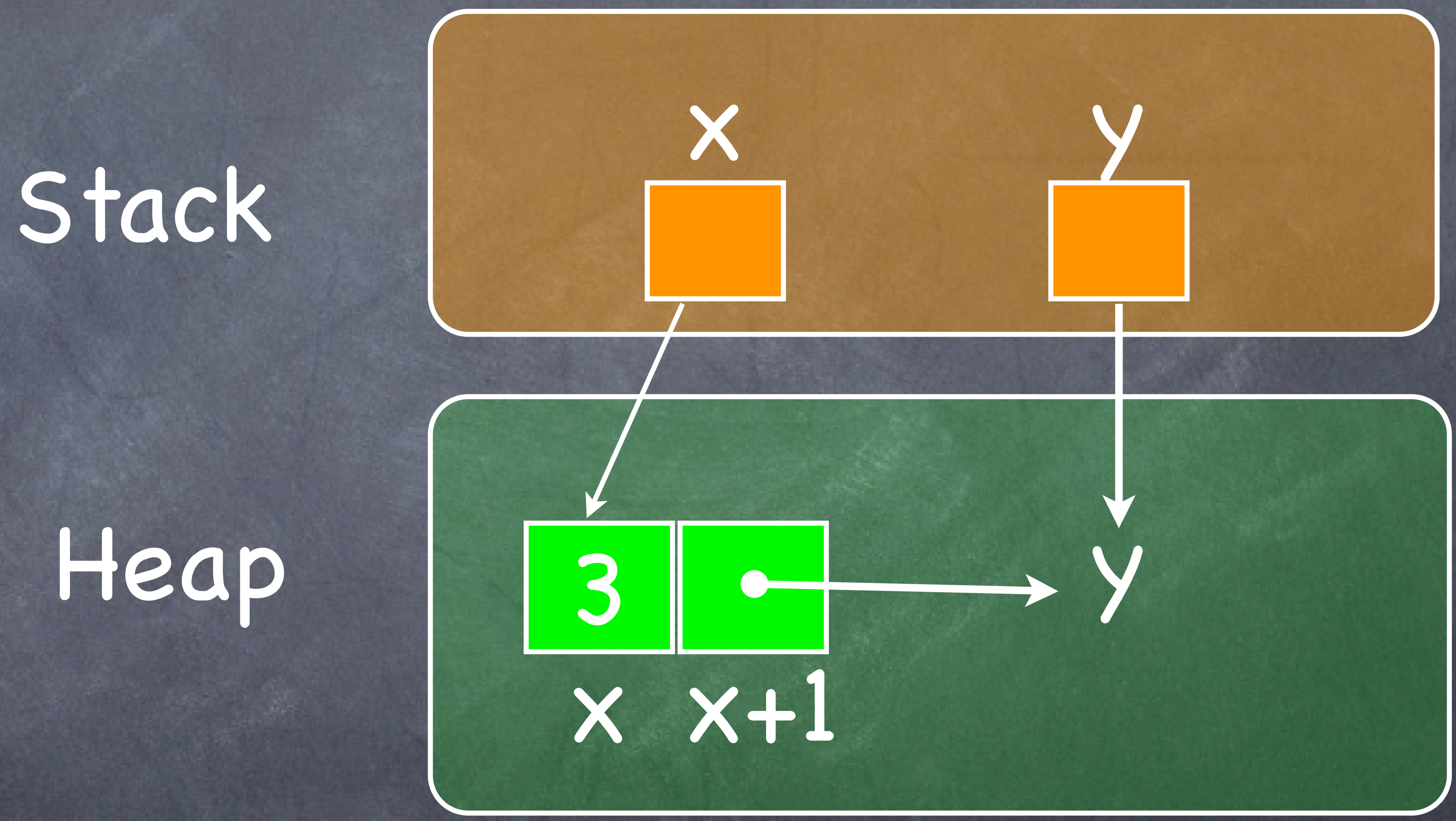
Heap



Abbreviation: 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

$x \mapsto 3, y$

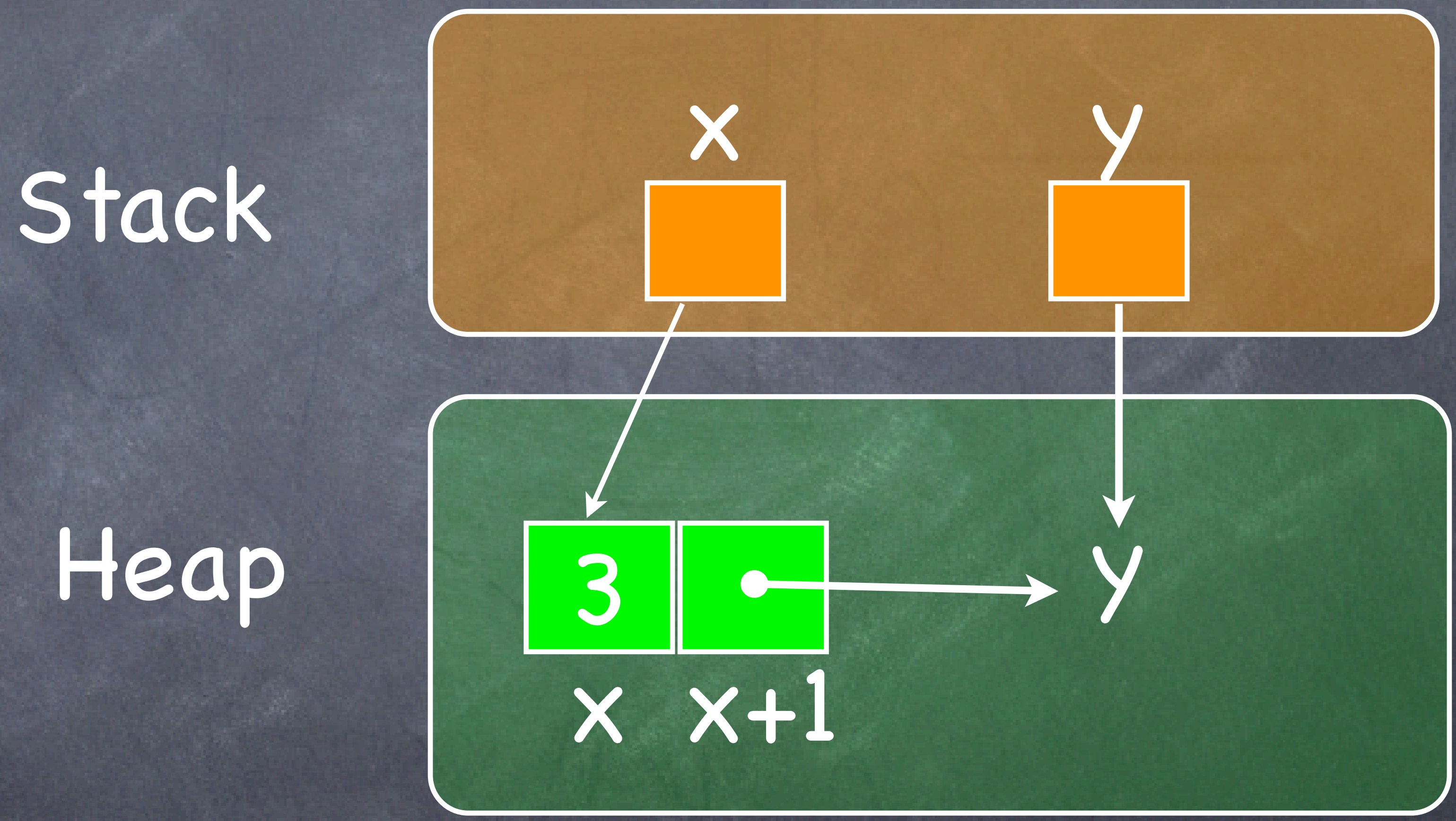


Abbreviation: E points to a record of several fields: $E \mapsto E_1, \ldots, E_n \triangleq E \mapsto E_1 * \cdots * E + n - 1 \mapsto E_n$

Example

$x \mapsto 3, y$

$y \mapsto 3, x$

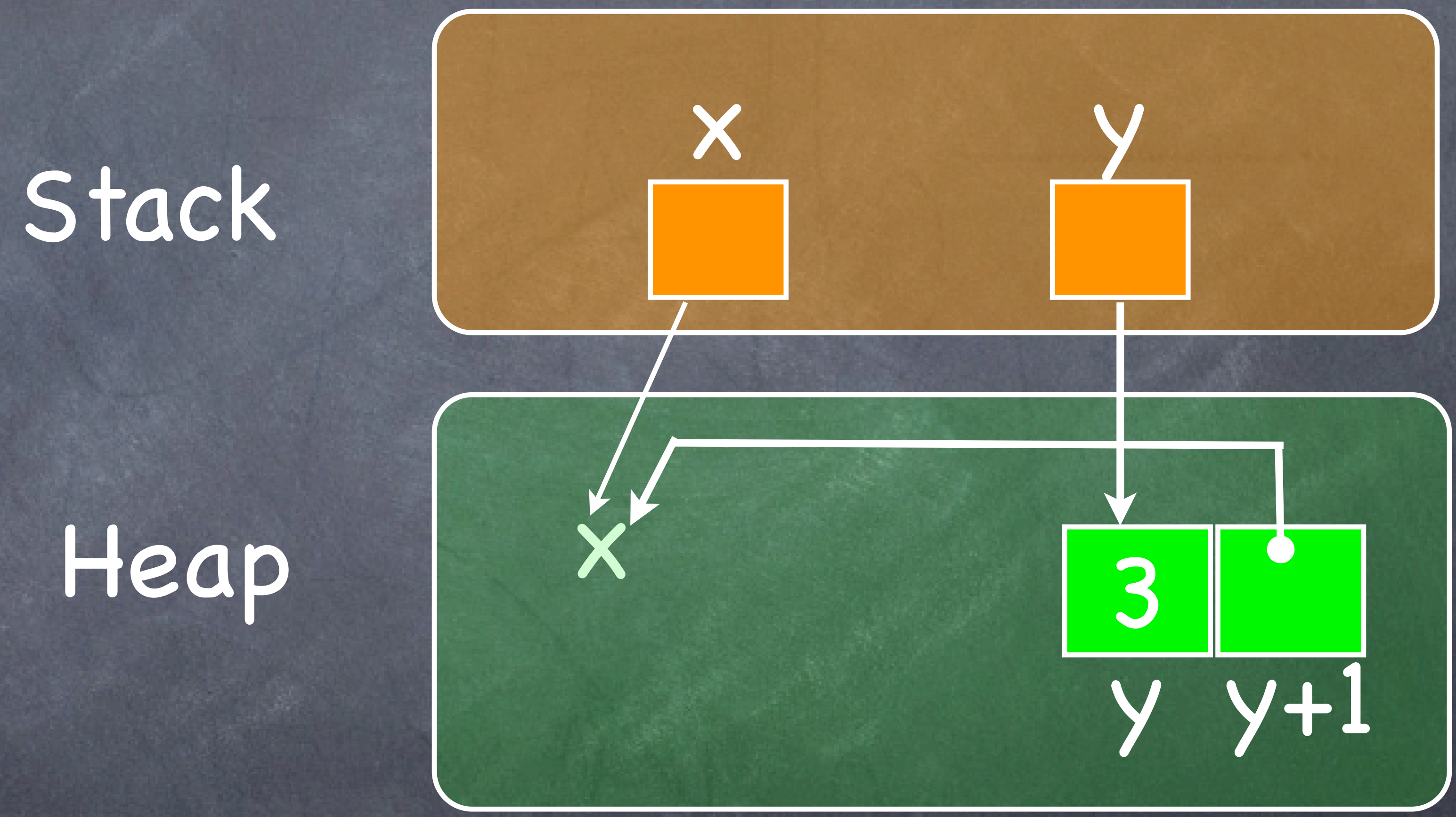


Abbreviation: E points to a record of several fields: $E \mapsto E_1, \ldots, E_n \triangleq E \mapsto E_1 * \cdots * E + n - 1 \mapsto E_n$

Example

$x \mapsto 3, y$

$y \mapsto 3, x$



Abbreviation: 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

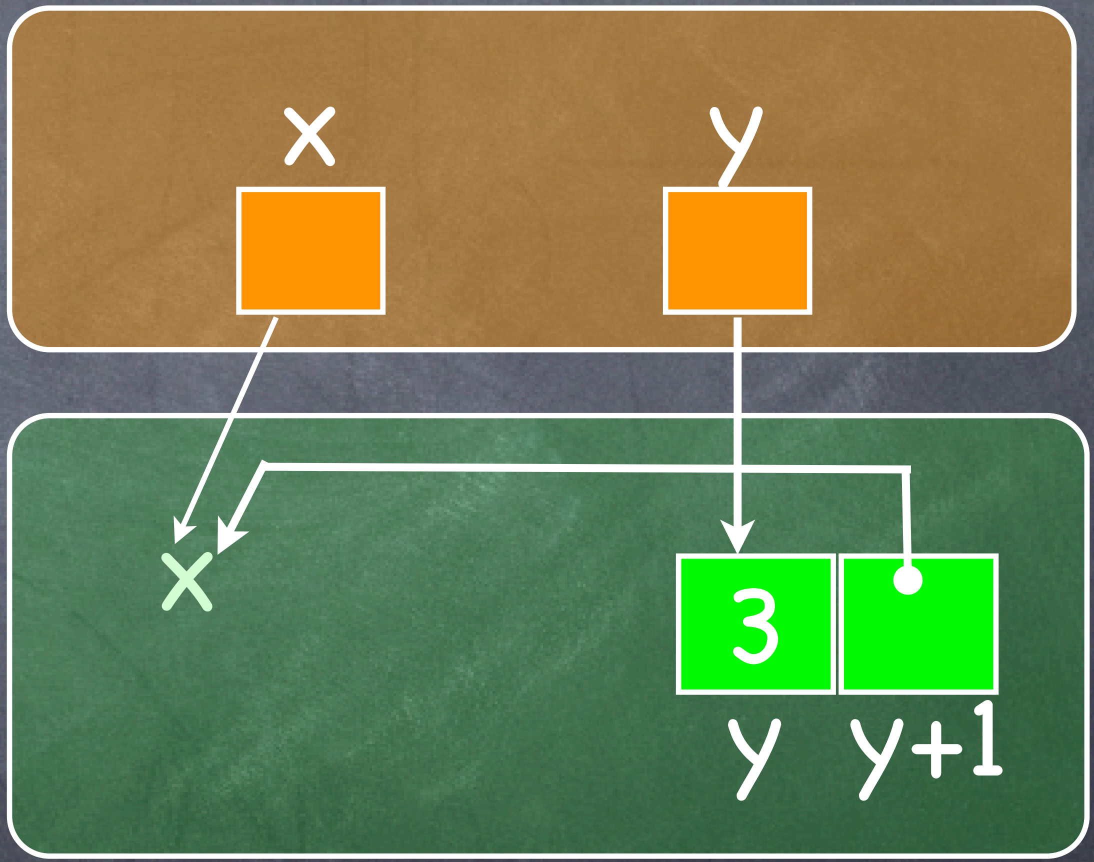
$$x \mapsto 3, y$$

$$y \mapsto 3, x$$

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

Stack

Heap



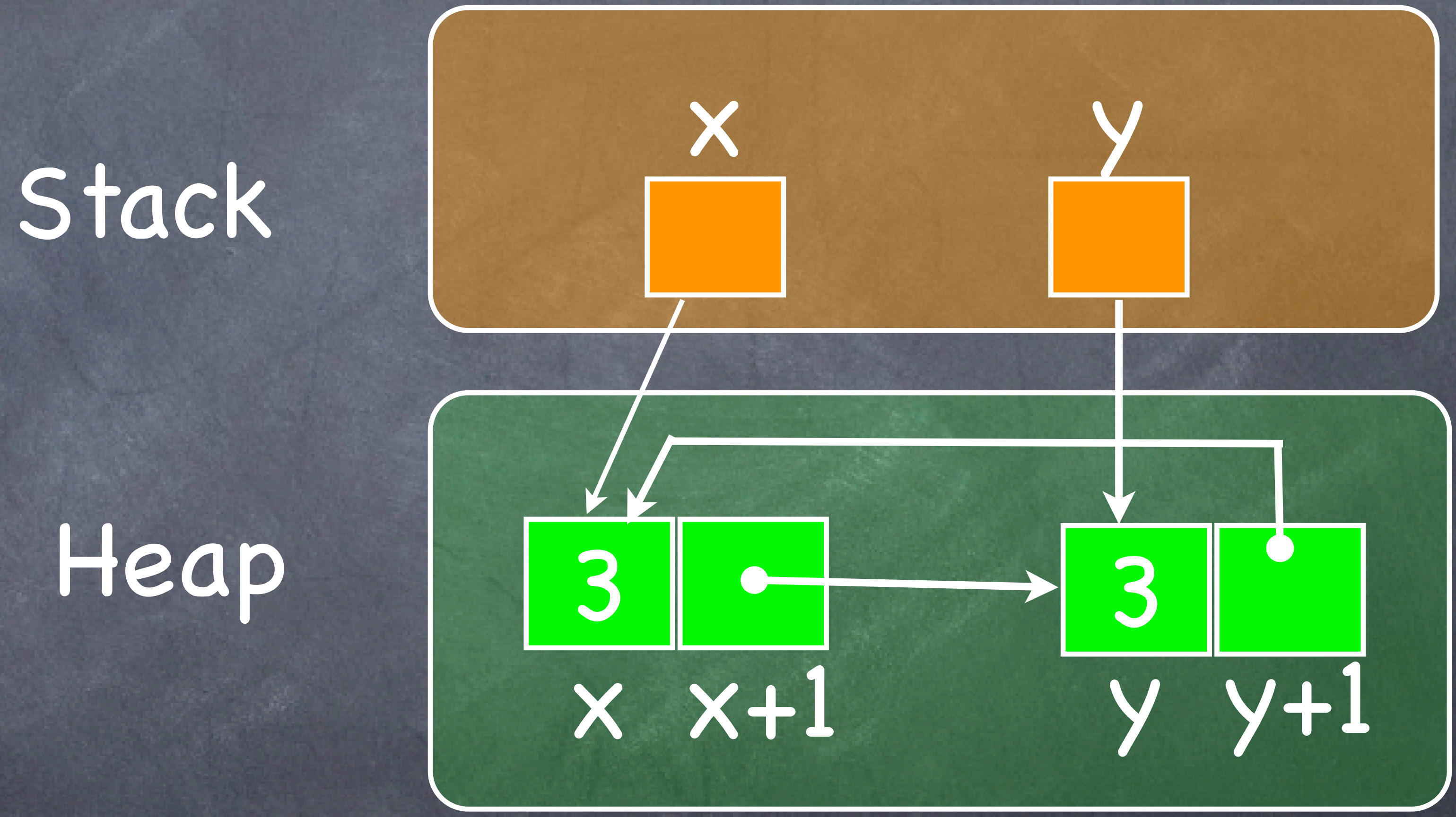
Abbreviation: E points to a record of several fields: $E \mapsto E_1, \ldots, E_n \triangleq E \mapsto E_1 * \cdots * E + n - 1 \mapsto E_n$

Example

$x \mapsto 3, y$

$y \mapsto 3, x$

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



Abbreviation: E points to a record of several fields: $E \mapsto E_1, \ldots, E_n \triangleq E \mapsto E_1 * \cdots * E + n - 1 \mapsto E_n$

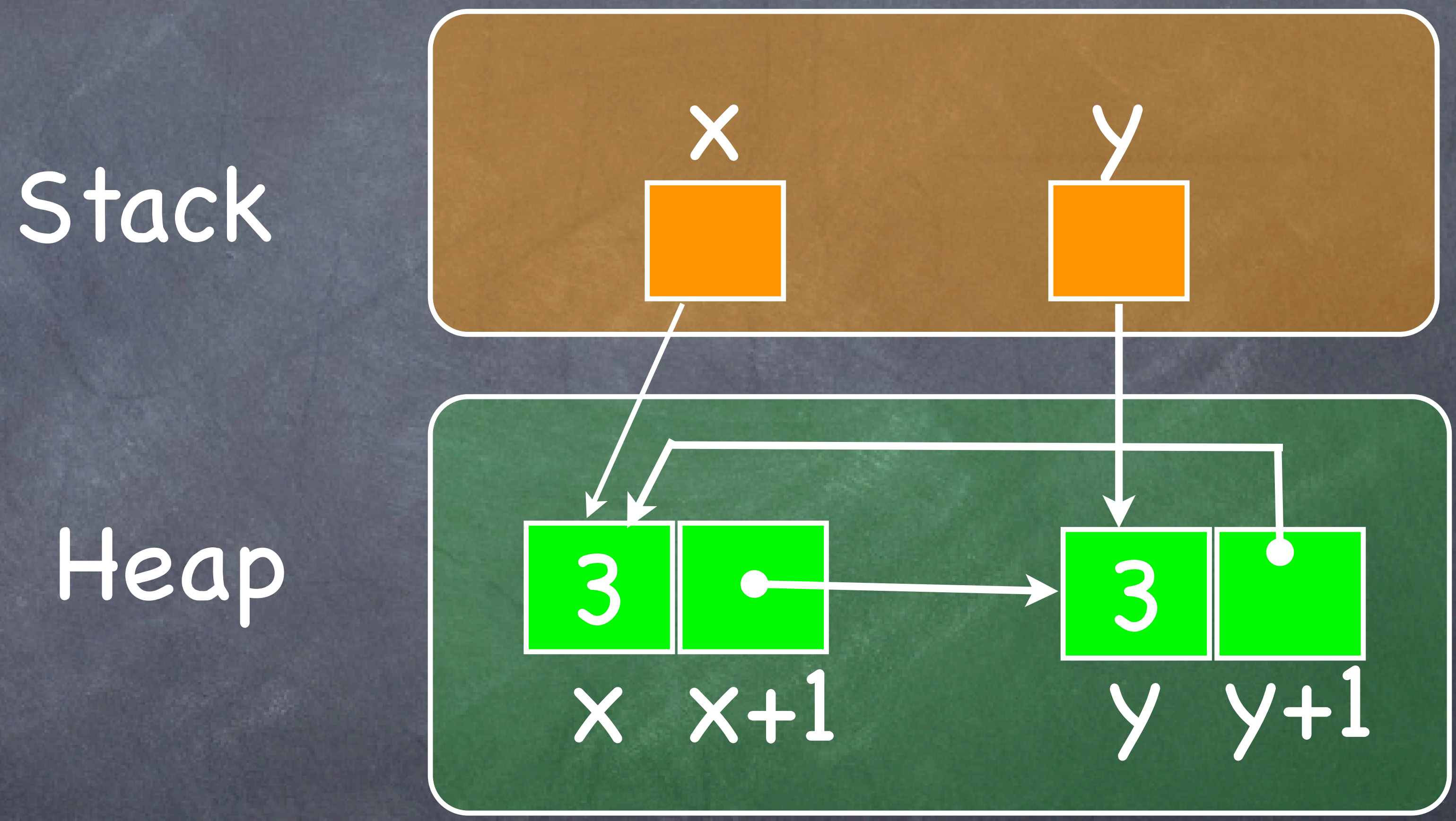
Example

$x \mapsto 3, y$

$y \mapsto 3, x$

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

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



Abbreviation: E points to a record of several fields: $E \mapsto E_1, \ldots, E_n \triangleq E \mapsto E_1 * \cdots * E + n - 1 \mapsto E_n$

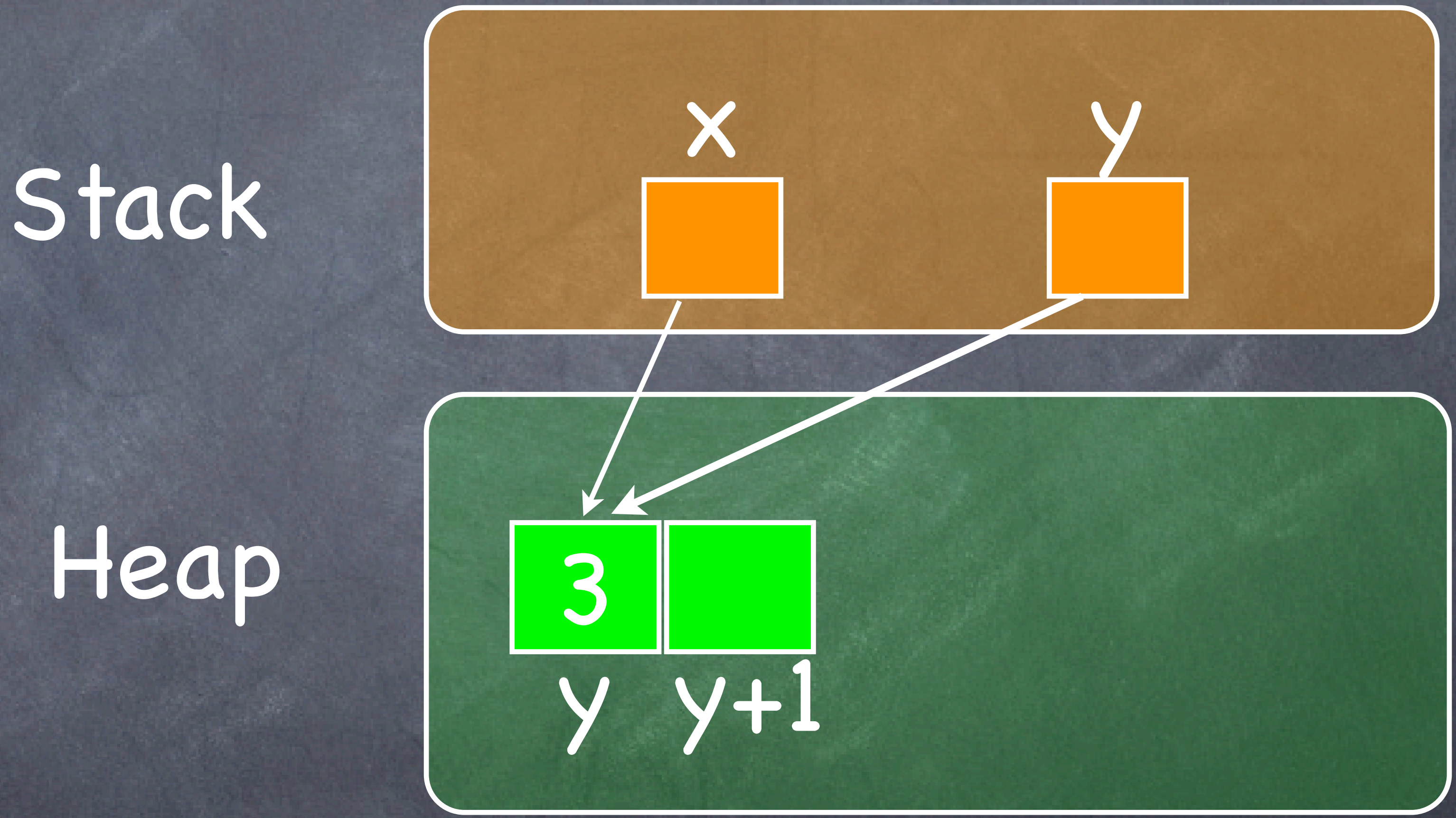
Example

$x \mapsto 3, y$

$y \mapsto 3, x$

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

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



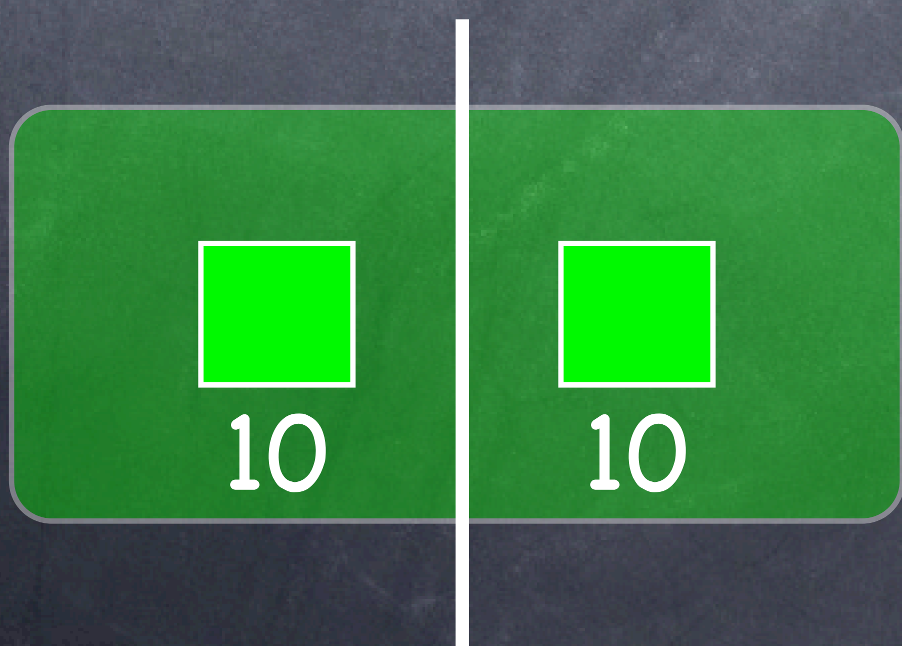
Abbreviation: E points to a record of several fields: $E \mapsto E_1, \ldots, E_n \triangleq E \mapsto E_1 * \cdots * E + n - 1 \mapsto E_n$

An inconsistency

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

An inconsistency

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



Try to be in two places
at the same time

...back to the real stuff:

Compositional Shape Analysis
by means of Bi-Abduction

Literature

- C. Calcagno, D. Distefano, P. O'Hearn and H. Yang. Compositional Shape Analysis by Means of Bi-Abduction. POPL 2009.
- D. Distefano. Attacking Large Industrial Code with Bi-Abductive Inference. FMICS 2009

A lot of real code out there uses pointer manipulation...

```
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
```

Is this correct?

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

We want to build tool that automatically answer such questions

Space Invader analyzer: overview

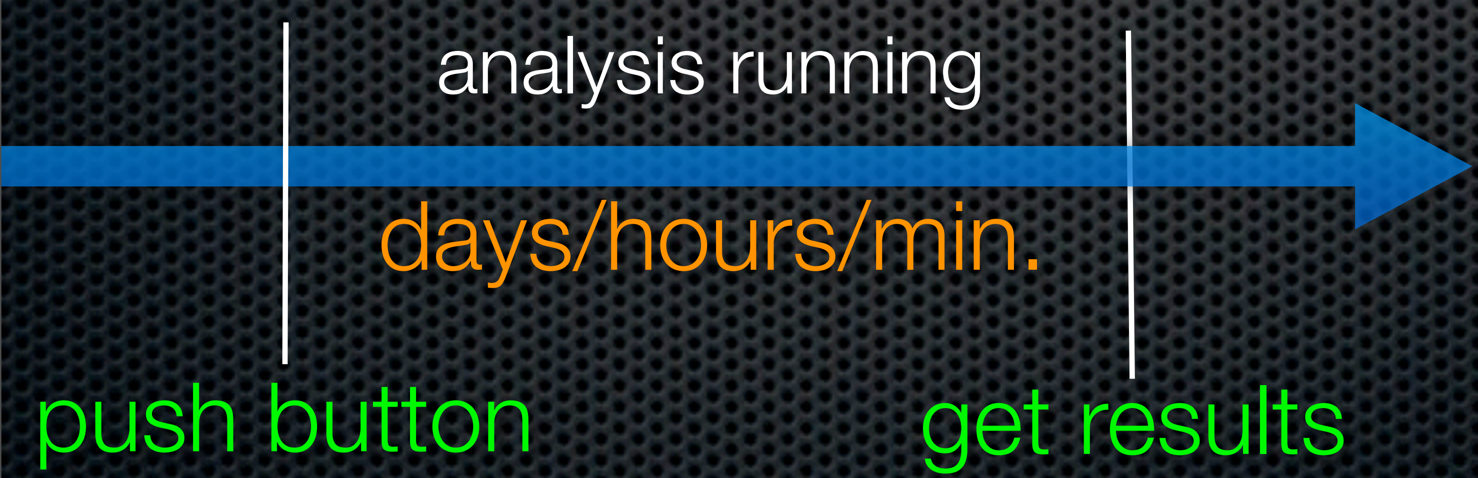
- Shape analyses discover deep properties about the heap: e.g., a variable points to a cyclic/acyclic doubly linked list,...
- Space Invader is Inter-procedural shape analysis for C programs
- Based on Separation Logic and Abstract interpretation to infer invariants
- Builds proofs or reports possible memory faults or memory leaks

Shape Analysis and Real Code

- So far shape analysis mostly applied to toy programs

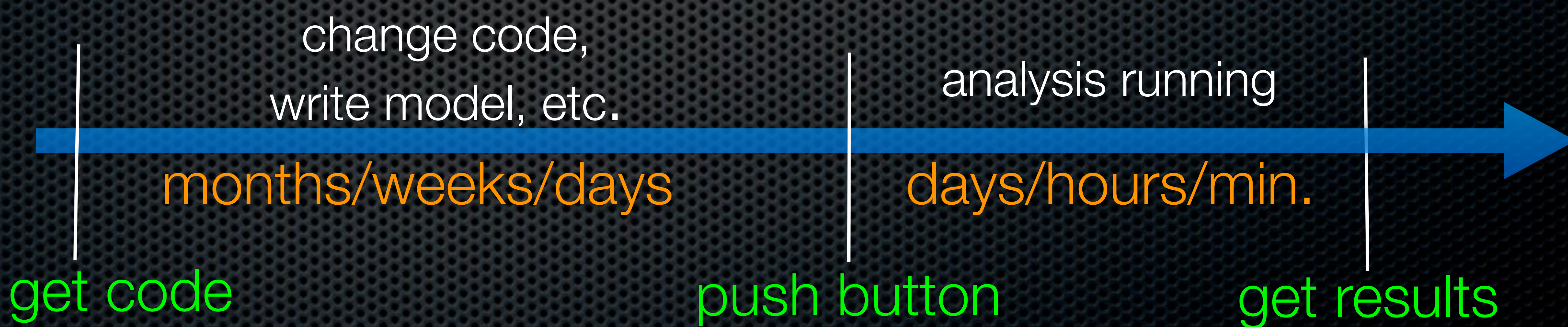
Shape Analysis and Real Code

- So far shape analysis mostly applied to toy programs



Shape Analysis and Real Code

- So far shape analysis mostly applied to toy programs



Fiction:

“no worries, device drivers use moslty lists”

```
typedef struct {
  PDEVICE_OBJECT          StackDeviceObject;
  PDEVICE_OBJECT          PortDeviceObject;
  PDEVICE_OBJECT          PhysicalDeviceObject;

  UNICODE_STRING          SymbolicLinkName;
  KSPIN_LOCK              ResetSpinLock;
  KSPIN_LOCK              CromSpinLock;
  KSPIN_LOCK              AsyncSpinLock;
  KSPIN_LOCK              IsochSpinLock;
  KSPIN_LOCK              IsochResourceSpinLock;


  BOOLEAN                 bShutdown;
  DEVICE_POWER_STATE      CurrentDevicePowerState;
  SYSTEM_POWER_STATE      CurrentSystemPowerState;

  ULONG                   GenerationCount;
  PASYNC_ADDRESS_DATA     Flink1;
  PASYNC_ADDRESS_DATA     Blink1;
  PBUS_RESET_IRP          Flink2;
  PBUS_RESET_IRP          Blink2;
  PCROM_DATA              Flink3;
  PCROM_DATA              Blink3;
  _PISOCH_DETACH_DATA     Flink4;
  _PISOCH_DETACH_DATA     Blink4;
  PISOCH_RESOURCE_DATA    Flink5;
  PISOCH_RESOURCE_DATA    Blink5;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

```
typedef struct ASYNC_ADDRESS_DATA {
  struct ASYNC_ADDRESS_DATA*  Flink1;
  struct ASYNC_ADDRESS_DATA*  Blink1;
  _PDEVICE_EXTENSION       DeviceExtension;
  PVOID                   Buffer;
  ULONG                   nLength;
  ULONG                   nAddressesReturned;
  PADDRESS_RANGE          AddressRange;
  HANDLE                  hAddressRange;
  PMDL                    pMdl;
} ASYNC_ADDRESS_DATA, *PASYNC_ADDRESS_DATA;

typedef struct BUS_RESET_IRP {
  struct BUS_RESET_IRP *Flink2;
  struct BUS_RESET_IRP *Blink2;
  PIRP            Irp;
} BUS_RESET_IRP, *PBUS_RESET_IRP;

typedef struct CROM_DATA {
  struct CROM_DATA *Flink3;
  struct CROM_DATA *Blink3;
  HANDLE          hCromData;
  PVOID           Buffer;
  PMDL            pMdl;
} CROM_DATA, *PCROM_DATA;

typedef struct ISOCH_RESOURCE_DATA {
  struct ISOCH_RESOURCE_DATA *Flink5;
  struct ISOCH_RESOURCE_DATA *Blink5;
  HANDLE          hResource;
} ISOCH_RESOURCE_DATA, *PISOCH_RESOURCE_DATA;
```

around 600 loc struct definitions

```
typedef struct {
  PDEVICE_OBJECT          StackDeviceObject;
  PDEVICE_OBJECT          PortDeviceObject;
  PDEVICE_OBJECT

  UNICODE_STRING
  KSPIN_LOCK              ResetSpinLock;
  KSPIN_LOCK              CromSpinLock;
  KSPIN_LOCK              AsyncSpinLock;
  KSPIN_LOCK              IsochSpinLock;
  KSPIN_LOCK              IsochResourceSpinLock;


  BOOLEAN                 bShutdown;
  DEVICE_POWER_STATE      CurrentDevicePowerState;
  SYSTEM_POWER_STATE      CurrentSystemPowerState;

  ULONG                   GenerationCount;
  PASYNC_ADDRESS_DATA     Flink1;
  PASYNC_ADDRESS_DATA     Blink1;
  PBUS_RESET_IRP          Flink2;
  PBUS_RESET_IRP          Blink2;
  PCROM_DATA              Flink3;
  PCROM_DATA              Blink3;
  _PISOCH_DETACH_DATA     Flink4;
  _PISOCH_DETACH_DATA     Blink4;
  PISOCH_RESOURCE_DATA    Flink5;
  PISOCH_RESOURCE_DATA    Blink5;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

```
typedef struct ASYNC_ADDRESS_DATA {
  struct ASYNC_ADDRESS_DATA*  Flink1;
  struct ASYNC_ADDRESS_DATA*  Blink1;
  _PDEVICE_EXTENSION       DeviceExtension;

                                  sReturned;
  PADDRESS_RANGE          AddressRange;
  HANDLE                  hAddressRange;
  PMDL                    pMdl;
} ASYNC_ADDRESS_DATA, *PASYNC_ADDRESS_DATA;

typedef struct BUS_RESET_IRP {
  struct BUS_RESET_IRP *Flink2;
  struct BUS_RESET_IRP *Blink2;
  PIRP            Irp;
} BUS_RESET_IRP, *PBUS_RESET_IRP;

typedef struct CROM_DATA {
  struct CROM_DATA *Flink3;
  struct CROM_DATA *Blink3;
  HANDLE          hCromData;
  PVOID           Buffer;
  PMDL            pMdl;
} CROM_DATA, *PCROM_DATA;

typedef struct ISOCH_RESOURCE_DATA {
  struct ISOCH_RESOURCE_DATA *Flink5;
  struct ISOCH_RESOURCE_DATA *Blink5;
  HANDLE          hResource;
} ISOCH_RESOURCE_DATA, *PISOCH_RESOURCE_DATA;
```

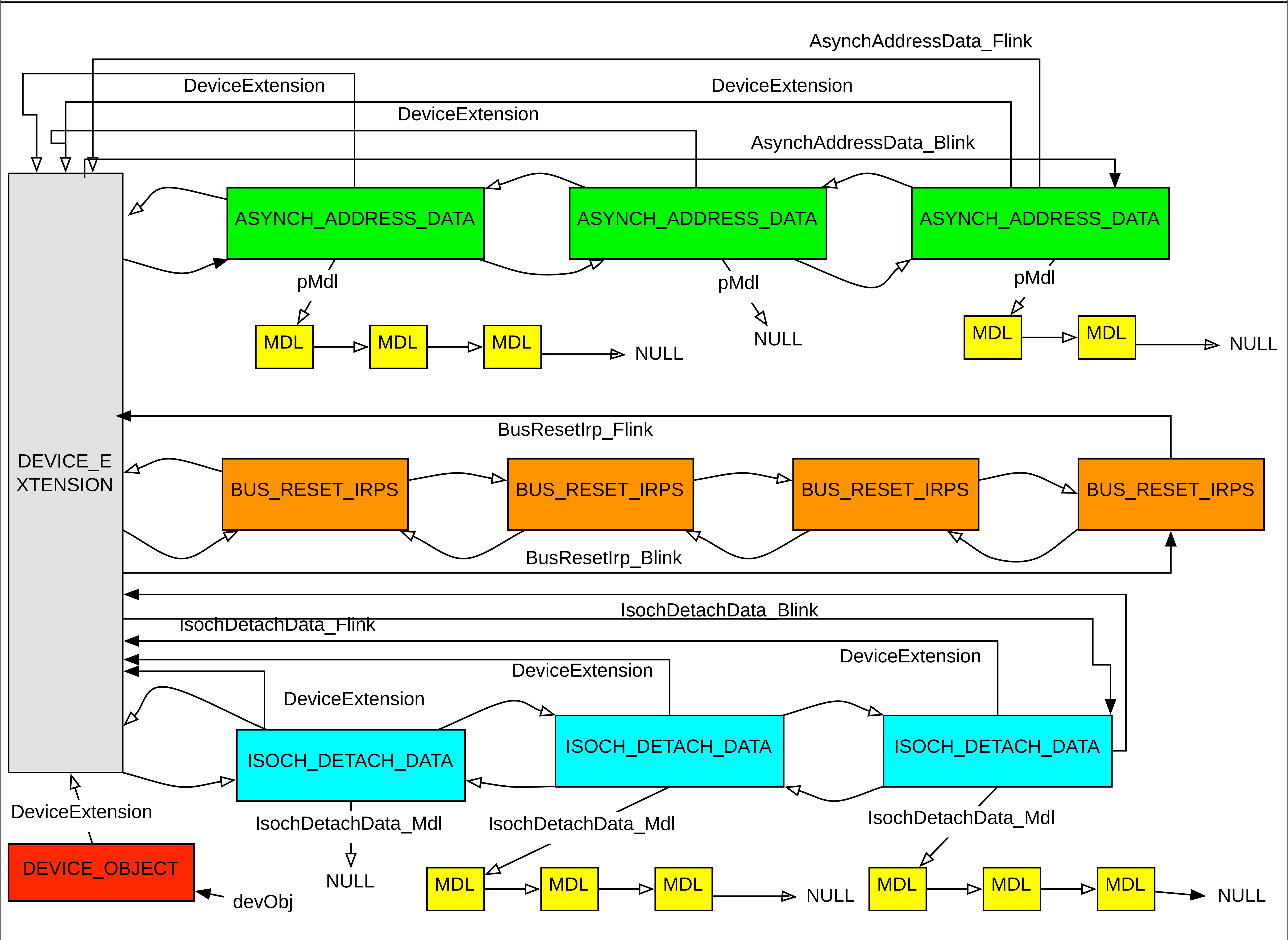
around 600 loc struct definitions

many big structs (around 20 fields) mutually pointing to aeach other in several way with several fields

```
typedef struct {
  PDEVICE_OBJECT          StackDeviceObject;
  PDEVICE_OBJECT          PortDeviceObject;
  PDEVICE_OBJECT

  UNICODE_STRING
  KSPIN_LOCK              ResetSpinLock;
  KSPIN_LOCK              CromSpinLock;
  KSPIN_LOCK              AsyncSpinLock;
  KSPIN_LOCK              IsochSpinLock;
  KSPIN_

  BOOLEA
  DEVICE
  SYSTEM

  ULONG
  PASYNC_
  PASYNC_ADDRESS_DATA     Blink1;
  PBUS_RESET_IRP          Flink2;
  PBUS_RESET_IRP          Blink2;
  PCROM_DATA              Flink3;
  PCROM_DATA              Blink3;
  _PISOCH_DETACH_DATA     Flink4;
  _PISOCH_DETACH_DATA     Blink4;
  PISOCH_RESOURCE_DATA    Flink5;
  PISOCH_RESOURCE_DATA    Blink5;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

```
typedef struct ASYNC_ADDRESS_DATA {
  struct ASYNC_ADDRESS_DATA*  Flink1;
  struct ASYNC_ADDRESS_DATA*  Blink1;
  _PDEVICE_EXTENSION          DeviceExtension;

                             sReturned;
  PADDRESS_RANGE             AddressRange;
  HANDLE                     hAddressRange;
  PMDL                       pMdl;
} ASYNC_ADDRESS_DATA, *PASYNC_ADDRESS_DATA;
```

```
  struct CROM_DATA *Blink3;
  HANDLE          hCromData;
  PVOID           Buffer;
  PMDL            pMdl;
} CROM_DATA, *PCROM_DATA;

typedef struct ISOCH_RESOURCE_DATA {
  struct ISOCH_RESOURCE_DATA *Flink5;
  struct ISOCH_RESOURCE_DATA *Blink5;
  HANDLE          hResource;
} ISOCH_RESOURCE_DATA, *PISOCH_RESOURCE_DATA;
```

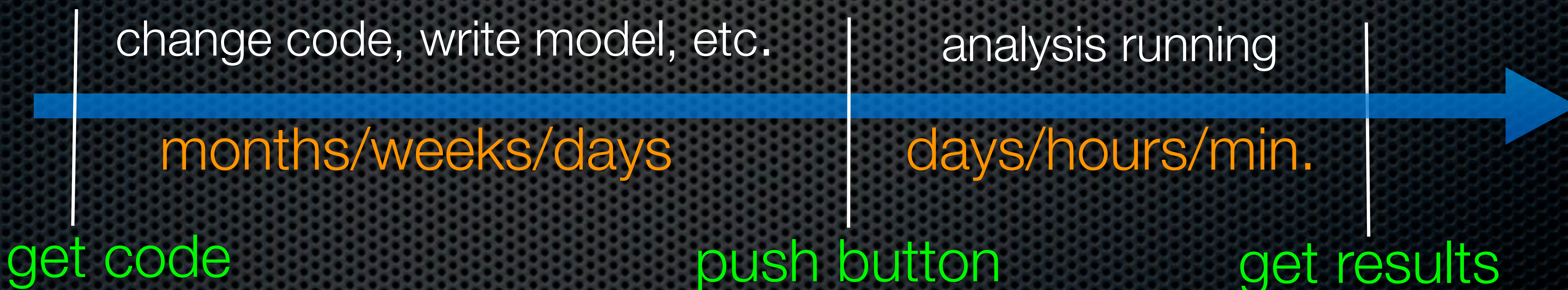

AsynchAddressData_Flink
DeviceExtension
DeviceExtension
DeviceExtension
AsynchAddressData_Blink
ASYNCH_ADDRESS_DATA
ASYNCH_ADDRESS_DATA
ASYNCH_ADDRESS_DATA
pMdl
pMdl
pMdl
MDL
MDL
MDL
NULL
NULL
MDL
MDL
NULL
DEVICE_EXTENSION
BusResetIrp_Flink
BUS_RESET_IRPS
BUS_RESET_IRPS
BUS_RESET_IRPS
BUS_RESET_IRPS
BusResetIrp_Blink
IsochDetachData_Blink
IsochDetachData_Flink
DeviceExtension
DeviceExtension
DeviceExtension
ISOCH_DETACH_DATA
ISOCH_DETACH_DATA
ISOCH_DETACH_DATA
DeviceExtension
IsochDetachData_Mdl
IsochDetachData_Mdl
IsochDetachData_Mdl
DEVICE_OBJECT
devObj
NULL
MDL
MDL
MDL
NULL
MDL
MDL
MDL
NULL

Fact:

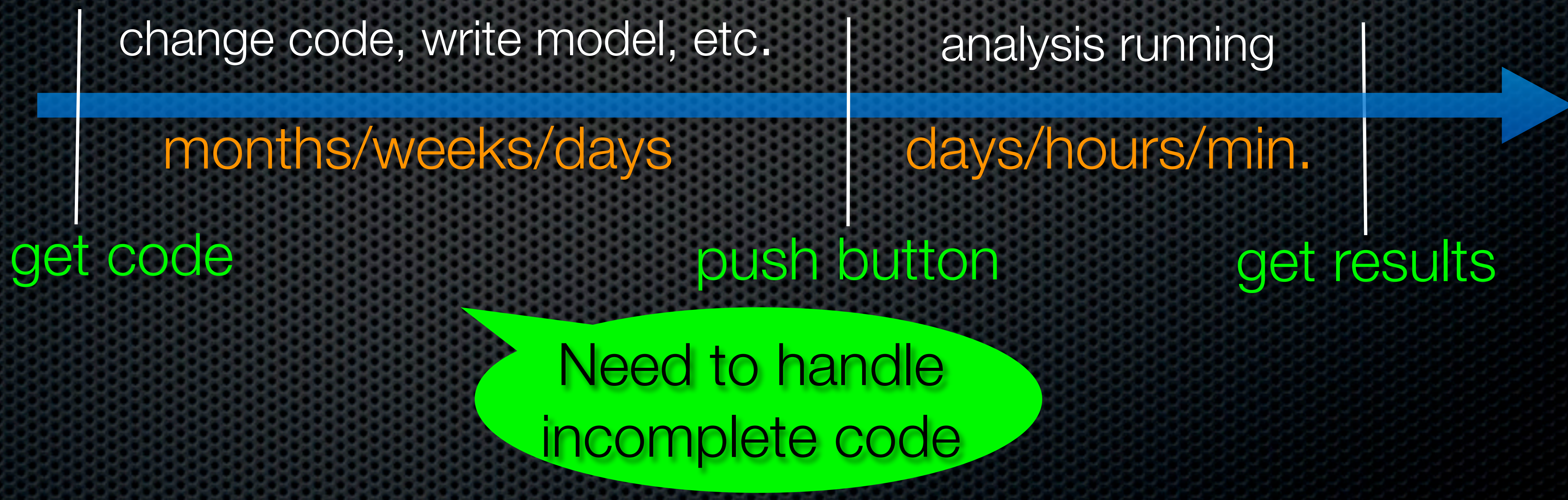
Real device drivers use lists in combination, resulting in more complicated data structures than those found in previous papers on shape analysis

Shape Analysis and Real Code

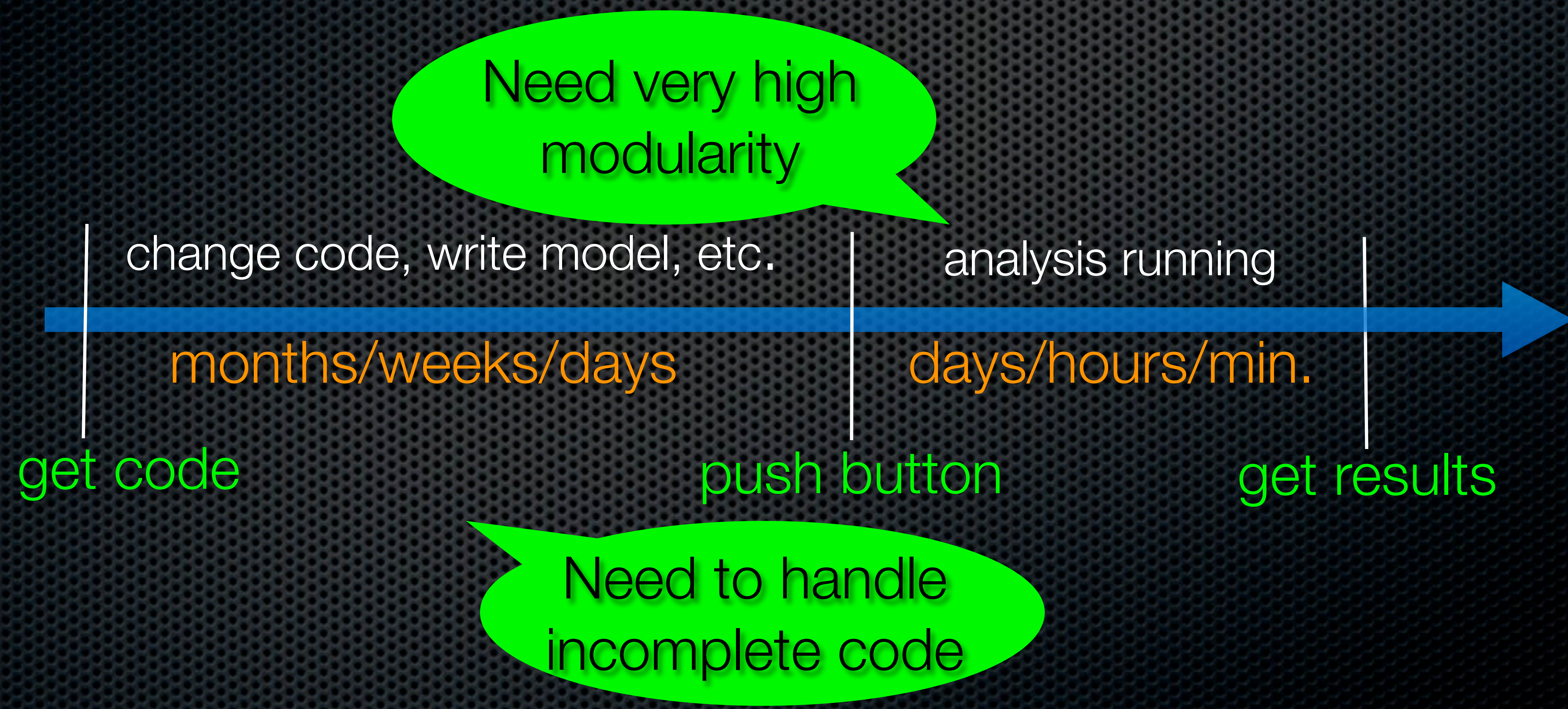
Shape Analysis and Real Code



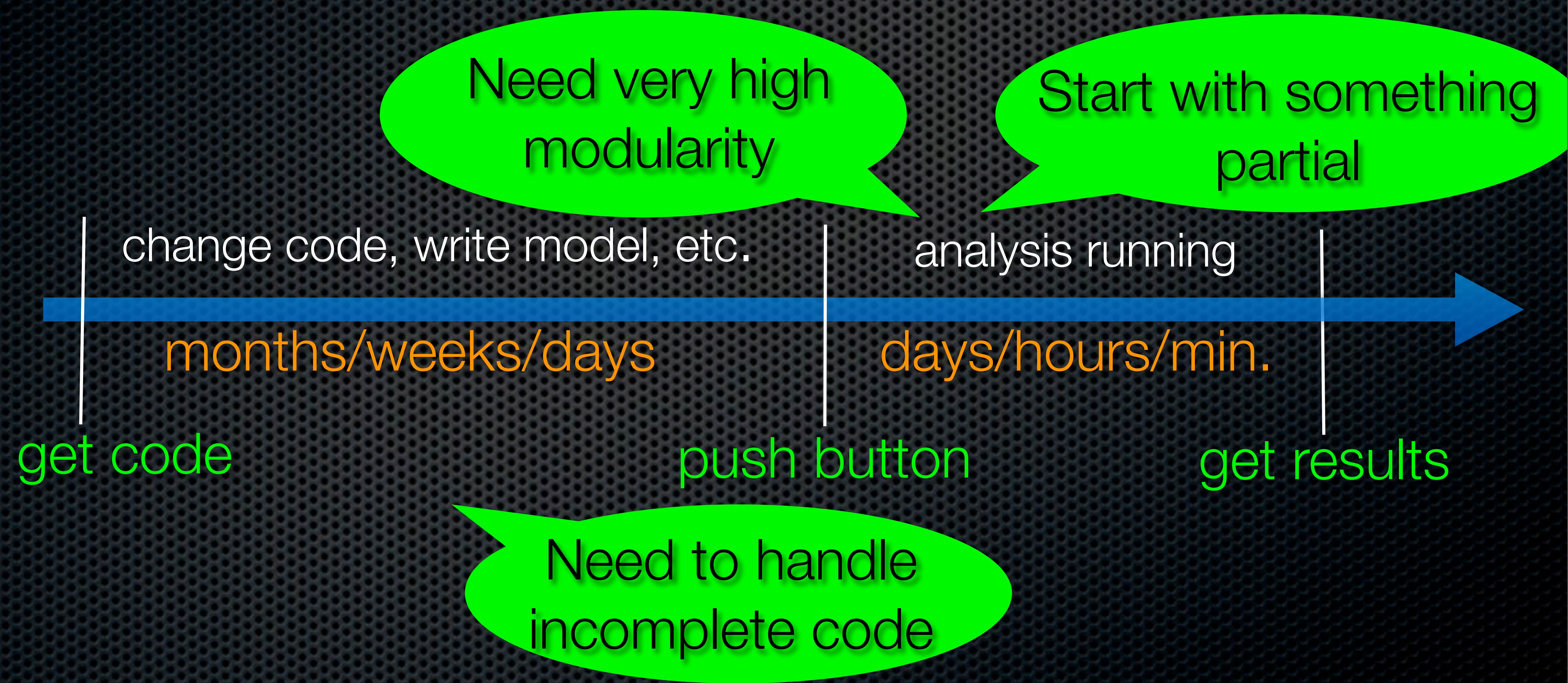
Shape Analysis and Real Code



Shape Analysis and Real Code



Shape Analysis and Real Code



Our response:
compositional Space Invader

✓Handles incomplete code
✓Admits partial results
✓Modular

Our response:

compositional Space Invader

- ✓ Handles incomplete code
- ✓ Admits partial results
- ✓ Modular

...demo!

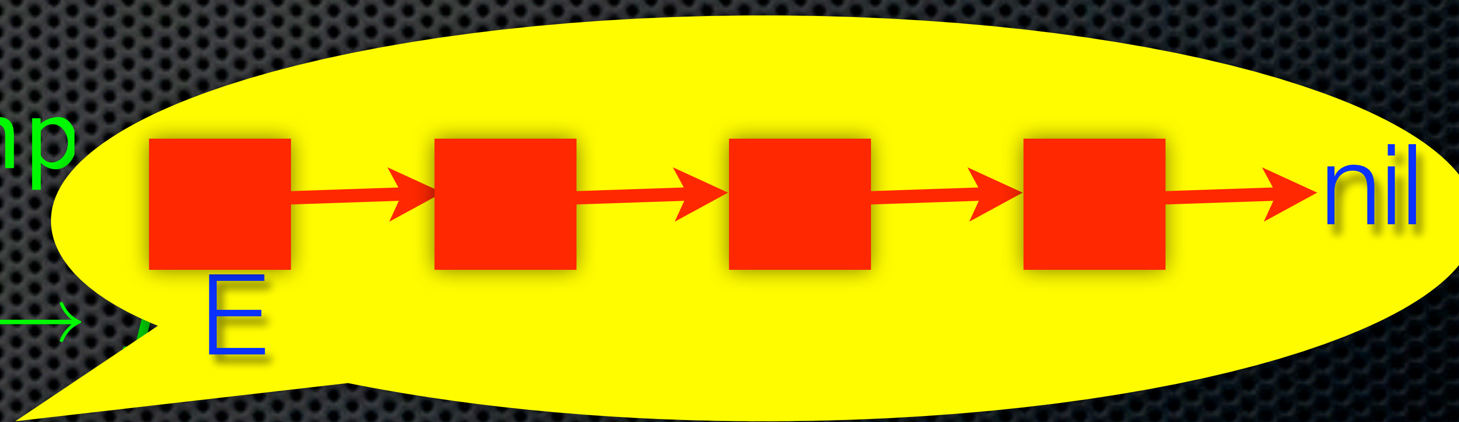
Basics

Notation

- Separation Logic's formulae to represent program states
- Some useful predicates:
  - The empty heap: emp
  - An allocated cell: $E \mapsto F$
  - A "complete" list: $\mathsf{list}(E)$
- P*Q means P and Q hold for disjoint portion of memory

Notation

- Separation Logic's formulae to represent program states
- Some useful predicates:
  - The empty heap: emp
  - An allocated cell: $E \mapsto$
  - A "complete" list: list(E)
- P*Q means P and Q hold for disjoint portion of memory



Notation

- Separation Logic's formulae to represent program states
- Some useful predicates:
  - The empty heap: **emp**
  - An allocated cell: $E \mapsto F$
  - A "complete" list: $\mathsf{list}(\mathsf{E})$
- P*Q means P and Q hold for disjoint portion of memory

Small specs

- Small specs encourage local reasoning and help to get small proofs
- When proving code involving procedures we use only their footprint

Example: use of small specs in proofs

```
{list(l1)*list(l2)}
Dispose(l1);

Dispose(l2);
```

Spec: `{list(l)} Dispose(l) {emp}`

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

Example: use of small specs in proofs

```
{list(l1)*list(l2)}
Dispose(l1);

Dispose(l2);
```

Spec: `{list(l)} Dispose(l) {emp}`

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

Example: use of small specs in proofs

```
{list(l1)*list(l2)}
Dispose(l1);
{emp*list(l2)}
Dispose(l2);
```

Spec: `{list(l)} Dispose(l) {emp}`

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

Example: use of small specs in proofs

```
{list(l1)*list(l2)}
Dispose(l1);
{list(l2)}
Dispose(l2);
```

Spec: `{list(l)} Dispose(l) {emp}`

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

Example: use of small specs in proofs

```
{list(l1)*list(l2)}
Dispose(l1);
{list(l2)}
Dispose(l2);
{emp}
```

Spec: `{list(l)} Dispose(l) {emp}`

$$\frac{\{P\}\ C\ \{Q\}}{\{P*R\}\ C\ \{Q*R\}}\quad \text{Frame Rule}$$

Novelties

Frame Inference

```
{list(l1)*list(l2)}
Dispose(l1);

Dispose(l2);
```

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

Spec: `{list(l)} Dispose(l) {emp}`

Frame Inference

```
{list(l1)*list(l2)}
Dispose(l1);

Dispose(l2);
```

$$\frac{\{P\}\ C\ \{Q\}}{\{P*R\}\ C\ \{Q*R\}}\quad \text{Frame Rule}$$

Spec: `{list(l)} Dispose(l) {emp}`

- In analysis to use the Frame Rule we need to compute R
- Frame inference problem: given A and B compute X such that $A \vdash B * X$

Frame Inference

```
{list(l1)*list(l2)}
Dispose(l1);

Dispose(l2);
```

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

Spec: `{list(l)} Dispose(l) {emp}`

- In analysis to use the Frame Rule we need to compute R
- Frame inference problem: given A and B compute X such that $A \vdash B * X$

Example:

`list(l1)*list(l2)` $\vdash$ `list(l1)* X`

Frame Inference

```
{list(l1)*list(l2)}
Dispose(l1);

Dispose(l2);
```

$$\frac{\{P\}\ C\ \{Q\}}{\{P*R\}\ C\ \{Q*R\}}\quad \text{Frame Rule}$$

Spec: `{list(l)} Dispose(l) {emp}`

- In analysis to use the Frame Rule we need to compute R
- Frame inference problem: given A and B compute X such that $A \vdash B * X$

Example:

`list(l1)*list(l2)` $\vdash$ `list(l1)*list(l2)`

Frame Inference

```
{list(l1)*list(l2)}
Dispose(l1);

Dispose(l2);
```

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

Spec: `{list(l)} Dispose(l) {emp}`

- In analysis to use the Frame Rule we need to compute R
- Frame inference problem: given A and B compute X such that $A \vdash B * X$

Example:

`list(l1)*list(l2)` $\vdash$ `list(l1)* list(l2)`

Frame Inference

```
{list(l1)*list(l2)}
Dispose(l1);
{emp*list(l2)}
Dispose(l2);
```

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

Spec: `{list(l)} Dispose(l) {emp}`

- In analysis to use the Frame Rule we need to compute R
- Frame inference problem: given A and B compute X such that $A \vdash B * X$

Example:

`list(l1)*list(l2)` $\vdash$ `list(l1)* list(l2)`

Abduction

Abductive Inference (Charles Peirce, circa 1900, writing about the scientific process)

"Abduction is the process of forming an explanatory hypothesis. It is the only logical operation which introduces any new idea"

"A man must be downright crazy to deny that science has made many true discoveries. But every single item of scientific theory which stands established today has been due to Abduction."

The Collected Papers of Charles Sanders Peirce, Volume V, Pragmatism and Pragmaticism

Abduction for Space Invader

Abduction for Space Invader

Abduction for Space Invader

Abduction Inference:

given A and B compute X such that

$$A * X \vdash B$$

Abduction for Space Invader

Abduction Inference:

given A and B compute X such that

$$A * X \vdash B$$

Example:

Spec: `{list(l1)*list(l2)} Dispose_Two_Lists(l1,l2) {emp}`

`list(l1)`

Abduction for Space Invader

Abduction Inference:

given A and B compute X such that

$$A * X \vdash B$$

Example:

Spec: `{list(l1)*list(l2)} Dispose_Two_Lists(l1,l2) {emp}`

$$\texttt{list(l1)*X} \vdash \texttt{list(l1)*list(l2)}$$

Abduction for Space Invader

Abduction Inference:

given A and B compute X such that

$$A * X \vdash B$$

Example:

Spec: `{list(l1)*list(l2)} Dispose_Two_Lists(l1,l2) {emp}`

`list(l1)*list(l2)` $\vdash$ `list(l1)*list(l2)`

Abduction is not enough

If heaps A and B are incomparable abduction and frame inference alone are not enough.

We need to synthesize both missing portion of state and leftover portion of state

Heap A

$$x \mapsto y \quad * \quad y \mapsto y'$$

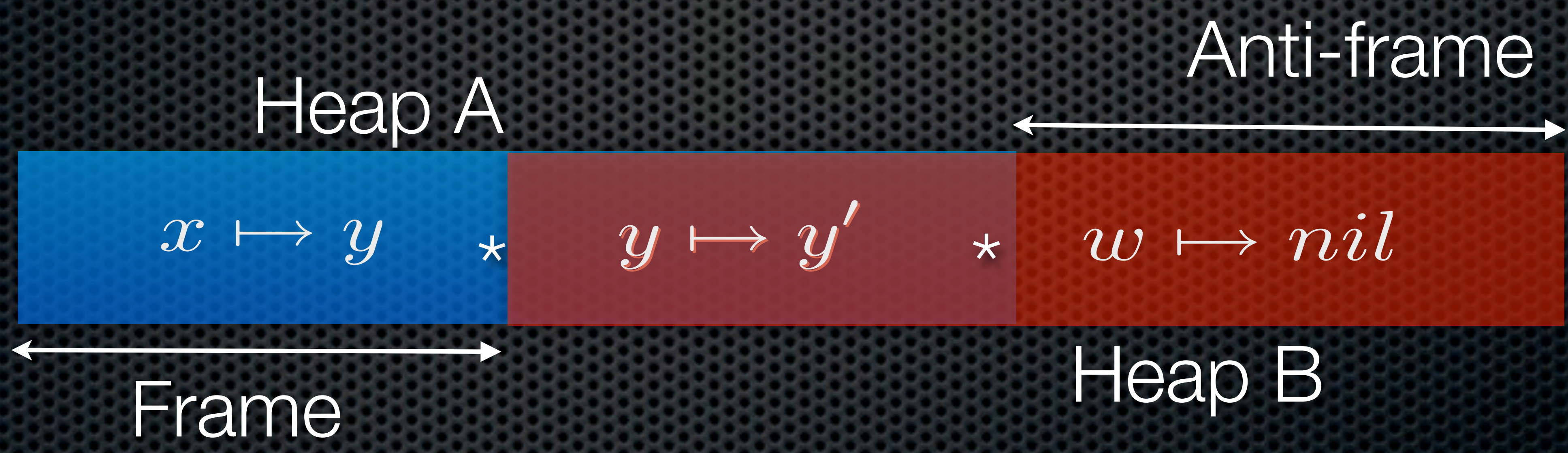
$$y \mapsto y' \quad * \quad w \mapsto nil$$

Heap B

Abduction is not enough

If heaps A and B are incomparable abduction and frame inference alone are not enough.

We need to synthesize both missing portion of state and leftover portion of state



Bi-Abduction

Synthesizing both missing portion of state (anti-frame) and leftover portion of state (frame) give rise to a new notion

Bi-Abduction:

given A and B compute ?antiframe and ?frame such that

$$A * ?antiframe \vdash B * ?frame$$

Bi-Abduction

Synthesizing both missing portion of state (anti-frame) and leftover portion of state (frame) give rise to a new notion

Bi-Abduction:

given A and B compute ?antiframe and ?frame such that

$$A * ?antiframe \vdash B * ?frame$$

Example:

$$x \mapsto 0 * z \mapsto 0 * ?antiframe \vdash \mathsf{list}(x) * \mathsf{list}(y) * ?frame$$

Bi-Abduction

Synthesizing both missing portion of state (anti-frame) and leftover portion of state (frame) give rise to a new notion

Bi-Abduction:

given A and B compute ?antiframe and ?frame such that

$$A * ?antiframe \vdash B * ?frame$$

Example:

$$x \mapsto 0 * z \mapsto 0 * \mathsf{list}(y) \vdash \mathsf{list}(x) * \mathsf{list}(y) * z \mapsto 0$$

Bi-Abduction

Synthesizing both missing portion of state (anti-frame) and leftover portion of state (frame) give rise to a new notion

Bi-Abduction:

given A and B compute ?antiframe and ?frame such that

$$A * ?antiframe \vdash B * ?frame$$

Example:

$$x \mapsto 0 * z \mapsto 0 * \mathsf{list}(y) \vdash \mathsf{list}(x) * \mathsf{list}(y) * z \mapsto 0$$

Our POPL'09 paper defines a theorem prover for Bi-Abduction

Bi-Abductive spec synthesis

Pre: list(x) * list(y)

```
void foo(list_item *x,list_item *y)
```

Post: list(x)

```
void p(list_item *y) { emp
  list_item *x, *z;
  x=malloc(sizeof(list_item)); x->tail = 0;   emp
  z=malloc(sizeof(list_item)); z->tail = 0;
  foo(x,y);
  foo(x,z);
}
```

Bi-abductive prover

Bi-Abductive spec synthesis

Pre: list(x) * list(y)

```
void foo(list_item *x,list_item *y)
```

Post: list(x)

```
void p(list_item *y) {    emp
  list_item *x, *z;
  x=malloc(sizeof(list_item)); x->tail = 0;    emp
  z=malloc(sizeof(list_item)); z->tail = 0;    x ↦ 0
  foo(x,y);
  foo(x,z);
}
```

Bi-abductive prover

Bi-Abductive spec synthesis

Pre: list(x) * list(y)

```
void foo(list_item *x,list_item *y)
```

Post: list(x)

```
void p(list_item *y) { emp
  list_item *x, *z;
  x=malloc(sizeof(list_item)); x->tail =  0;
  z=malloc(sizeof(list_item)); z->tail =  0;
  foo(x,y);
  foo(x,z);
}
```

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$

Bi-abductive prover

Bi-Abductive spec synthesis

Pre: list(x) * list(y)

```
void foo(list_item *x,list_item *y)
```

Post: list(x)

```
void p(list_item *y) {        emp
  list_item *x, *z;
  x=malloc(sizeof(list_item)); x->tail = 0;     emp
  z=malloc(sizeof(list_item)); z->tail = 0;     x ↦ 0
  foo(x,y);                                     x ↦ 0 * z ↦ 0
  foo(x,z);
}
```

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$

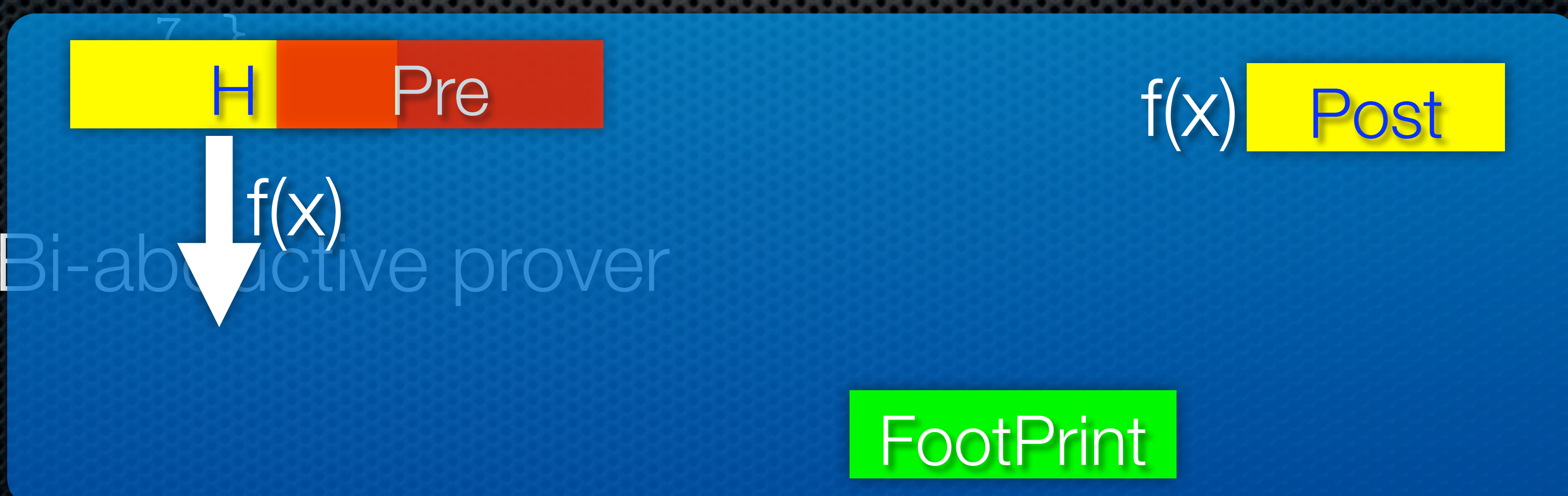
Bi-Abductive spec synthesis

Pre: list(x) * list(y)

```
void foo(list_item *x,list_item *y)
```

Post: list(x)

```
void p(list_item *y) {                                   emp
  list_item *x, *z;
  x=malloc(sizeof(list_item)); x->tail = 0;     emp
  z=malloc(sizeof(list_item)); z->tail = 0;     x ↦ 0
  foo(x,y);                                     x ↦ 0 * z ↦ 0
  foo(x,z);
}
```



Bi-Abductive spec synthesis

Pre: list(x) * list(y)

```
void foo(list_item *x,list_item *y)
```

Post: list(x)

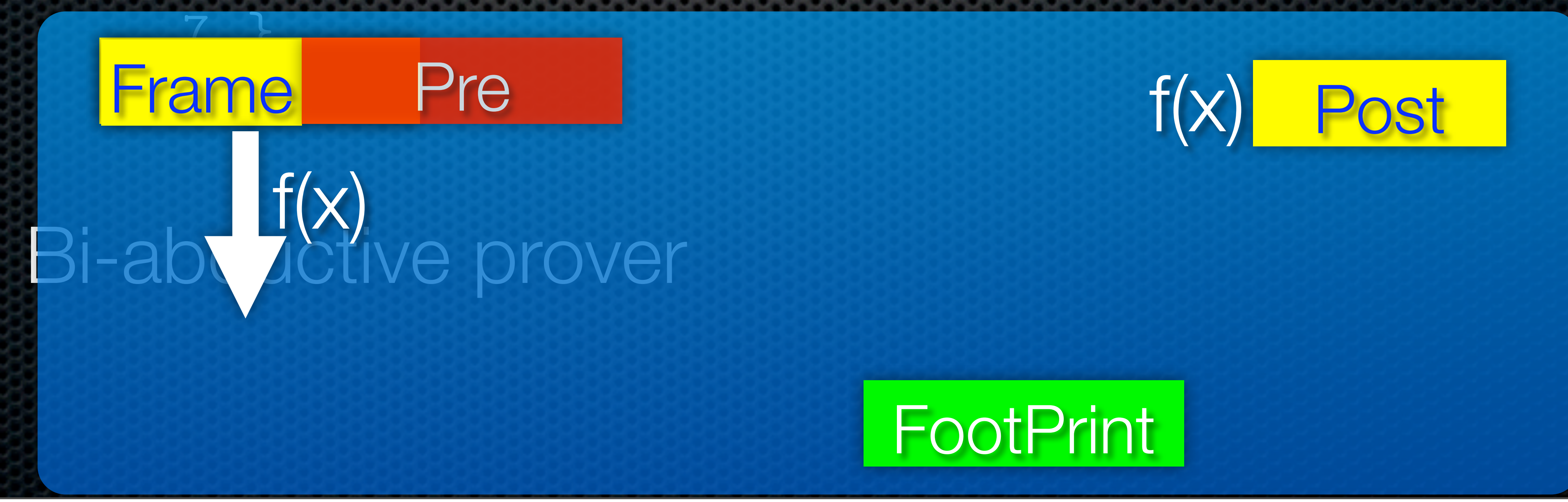
```
void p(list_item *y) {
  list_item *x, *z;
  x=malloc(sizeof(list_item)); x->tail = 0;
  z=malloc(sizeof(list_item)); z->tail = 0;
  foo(x,y);
  foo(x,z);
}
```

emp

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$



Bi-Abductive spec synthesis

Pre: list(x) * list(y)

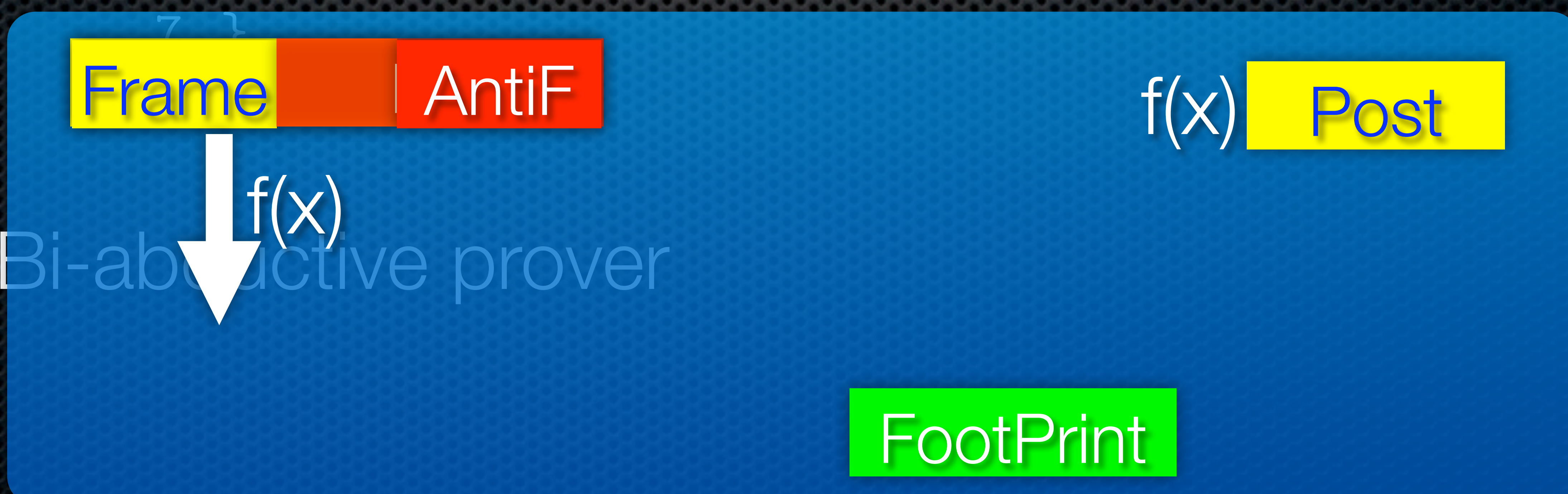
void foo(list_item *x,list_item *y)

Post: list(x)

```
void p(list_item *y) {                                  emp
  list_item *x, *z;
  x=malloc(sizeof(list_item)); x->tail = 0;             emp
  z=malloc(sizeof(list_item)); z->tail = 0;             x |-> 0
  foo(x,y);                                             x |-> 0 * z |-> 0
  foo(x,z);
}
```

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$



Bi-Abductive spec synthesis

Pre: list(x) * list(y)

void foo(list_item *x,list_item *y)

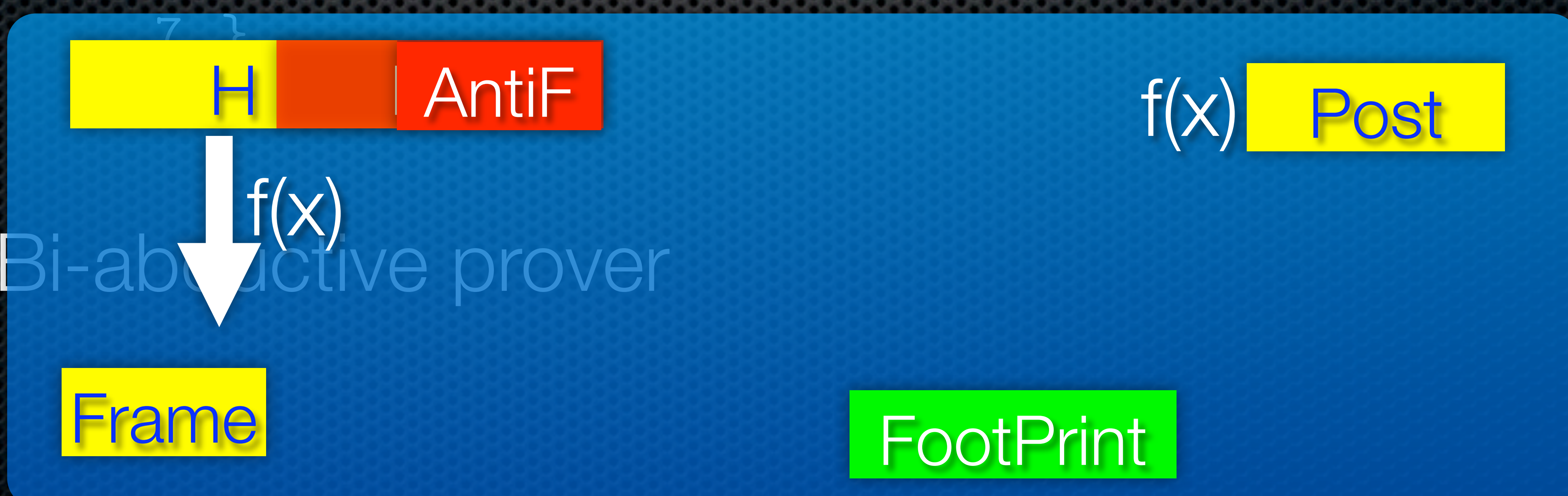
Post: list(x)

```
void p(list_item *y) {      emp
  list_item *x, *z;
  x=malloc(sizeof(list_item)); x->tail = 0;     emp
  z=malloc(sizeof(list_item)); z->tail = 0;     x |-> 0
  foo(x,y);                                     x |-> 0 * z |-> 0
  foo(x,z);
}
```

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$



Bi-Abductive spec synthesis

Pre: list(x) * list(y)

```
void foo(list_item *x,list_item *y)
```

Post: list(x)

```
void p(list_item *y) {
  list_item *x, *z;
  x=malloc(sizeof(list_item)); x->tail = 0;
  z=malloc(sizeof(list_item)); z->tail = 0;
  foo(x,y);
  foo(x,z);
}
```

emp

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$

Bi-Abductive spec synthesis

Pre: list(x) * list(y)

```
void foo(list_item *x,list_item *y)
```

Post: list(x)

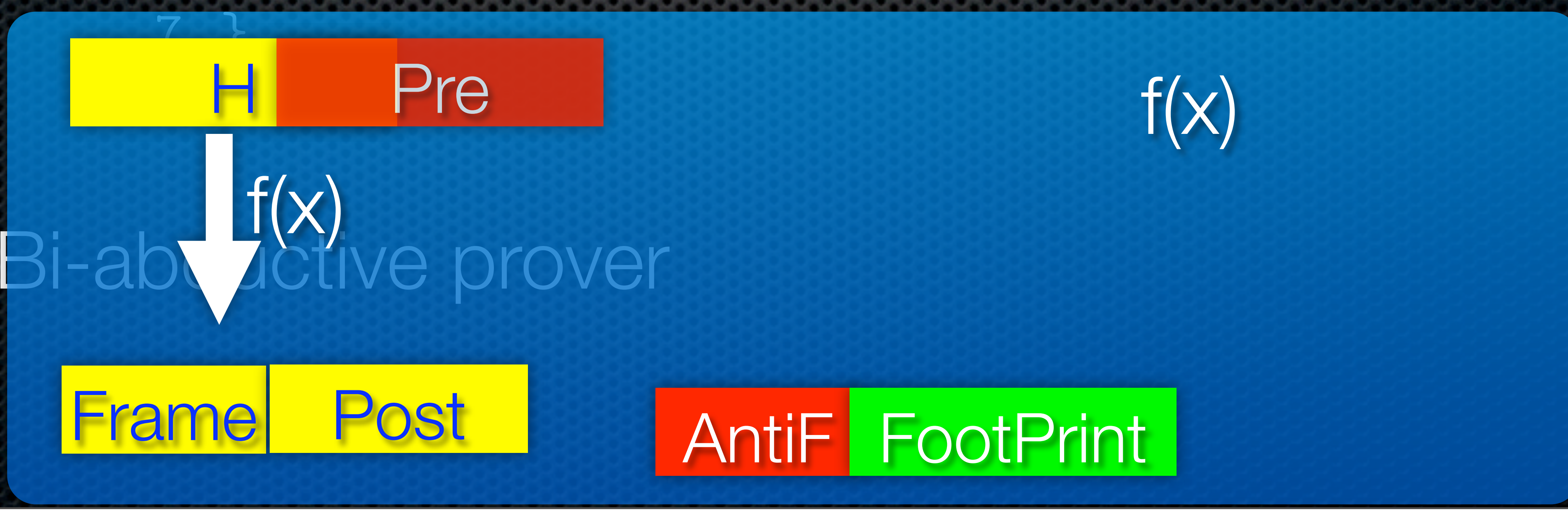
```
void p(list_item *y) {
  list_item *x, *z;
  x=malloc(sizeof(list_item)); x->tail = 0;
  z=malloc(sizeof(list_item)); z->tail = 0;
  foo(x,y);
  foo(x,z);
}
```

emp

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$



Bi-Abductive spec synthesis

Pre: list(x) * list(y)

```
void foo(list_item *x,list_item *y)
```

Post: list(x)

```
void p(list_item *y) {
  list_item *x, *z;
  x=malloc(sizeof(list_item)); x->tail = 0;
  z=malloc(sizeof(list_item)); z->tail = 0;
  foo(x,y);
  foo(x,z);
}
```

emp

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$

Bi-abductive prover

Bi-Abductive spec synthesis

Pre: list(x) * list(y)

```
void foo(list_item *x,list_item *y)
```

Post: list(x)

```
void p(list_item *y) {
  list_item *x, *z;
  x=malloc(sizeof(list_item)); x->tail =  0;
  z=malloc(sizeof(list_item)); z->tail =  0;
  foo(x,y);
  foo(x,z);
}
```

emp

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$

Bi-abductive prover

$$x \mapsto 0 * z \mapsto 0 * ?antiframe \vdash \mathsf{list}(x) * \mathsf{list}(y) * ?frame$$

Bi-Abductive spec synthesis

Pre: list(x) * list(y)

```
void foo(list_item *x,list_item *y)
```

Post: list(x)

```
void p(list_item *y) { emp
  list_item *x, *z;
  x=malloc(sizeof(list_item)); x->tail = 0;    emp
  z=malloc(sizeof(list_item)); z->tail = 0;    x ↦ 0
  foo(x,y);                                    x ↦ 0 * z ↦ 0
  foo(x,z);
}
```

Bi-abductive prover

$$x \mapsto 0 * z \mapsto 0 * \mathsf{list}(y) \vdash \mathsf{list}(x) * \mathsf{list}(y) * z \mapsto 0$$

Bi-Abductive spec synthesis

Pre: list(x) * list(y)

```
void foo(list_item *x,list_item *y)
```

Post: list(x)

```
void p(list_item *y) {
  list_item *x, *z;
  x=malloc(sizeof(list_item)); x->tail = 0;
  z=malloc(sizeof(list_item)); z->tail = 0;
  foo(x,y);
  foo(x,z);
}
```

$\mathsf{list}(y)$

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$

$\mathsf{list}(x) * z \mapsto 0$

Bi-abductive prover

$$x \mapsto 0 * z \mapsto 0 * \mathsf{list}(y) \vdash \mathsf{list}(x) * \mathsf{list}(y) * z \mapsto 0$$

Bi-Abductive spec synthesis

Pre: list(x) * list(y)

```
void foo(list_item *x,list_item *y)
```

Post: list(x)

```
void p(list_item *y) {
  list_item *x, *z;
  x=malloc(sizeof(list_item)); x->tail = 0;
  z=malloc(sizeof(list_item)); z->tail = 0;
  foo(x,y);
  foo(x,z);
}
```

$\mathsf{list}(y)$

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$

$\mathsf{list}(x) * z \mapsto 0$

Bi-abductive prover

$$\mathsf{list}(x) * z \mapsto 0 * ?antiframe \vdash \mathsf{list}(x) * \mathsf{list}(z) * ?frame$$

Bi-Abductive spec synthesis

Pre: list(x) * list(y)

```
void foo(list_item *x,list_item *y)
```

Post: list(x)

```
void p(list_item *y) {
  list_item *x, *z;
  x=malloc(sizeof(list_item)); x->tail = 0;
  z=malloc(sizeof(list_item)); z->tail = 0;
  foo(x,y);
  foo(x,z);
}
```

$\mathsf{list}(y)$

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$

$\mathsf{list}(x) * z \mapsto 0$

Bi-abductive prover

$$\mathsf{list}(x) * z \mapsto 0 * \mathsf{emp} \vdash \mathsf{list}(x) * \mathsf{list}(z) * \mathsf{emp}$$

Bi-Abductive spec synthesis

Pre: list(x) * list(y)

```
void foo(list_item *x,list_item *y)
```

Post: list(x)

```
void p(list_item *y) {
  list_item *x, *z;
  x=malloc(sizeof(list_item)); x->tail = 0;
  z=malloc(sizeof(list_item)); z->tail = 0;
  foo(x,y);
  foo(x,z);
}
```

$\mathsf{list}(y)$

emp

$x \mapsto 0$

$x \mapsto 0 * z \mapsto 0$

$\mathsf{list}(x) * z \mapsto 0$

$\mathsf{list}(x)$

Bi-abductive prover

$$\mathsf{list}(x) * z \mapsto 0 * \mathsf{emp} \vdash \mathsf{list}(x) * \mathsf{list}(z) * \mathsf{emp}$$

General Schema Compositional Analysis

For function in the program we compute tables of specs

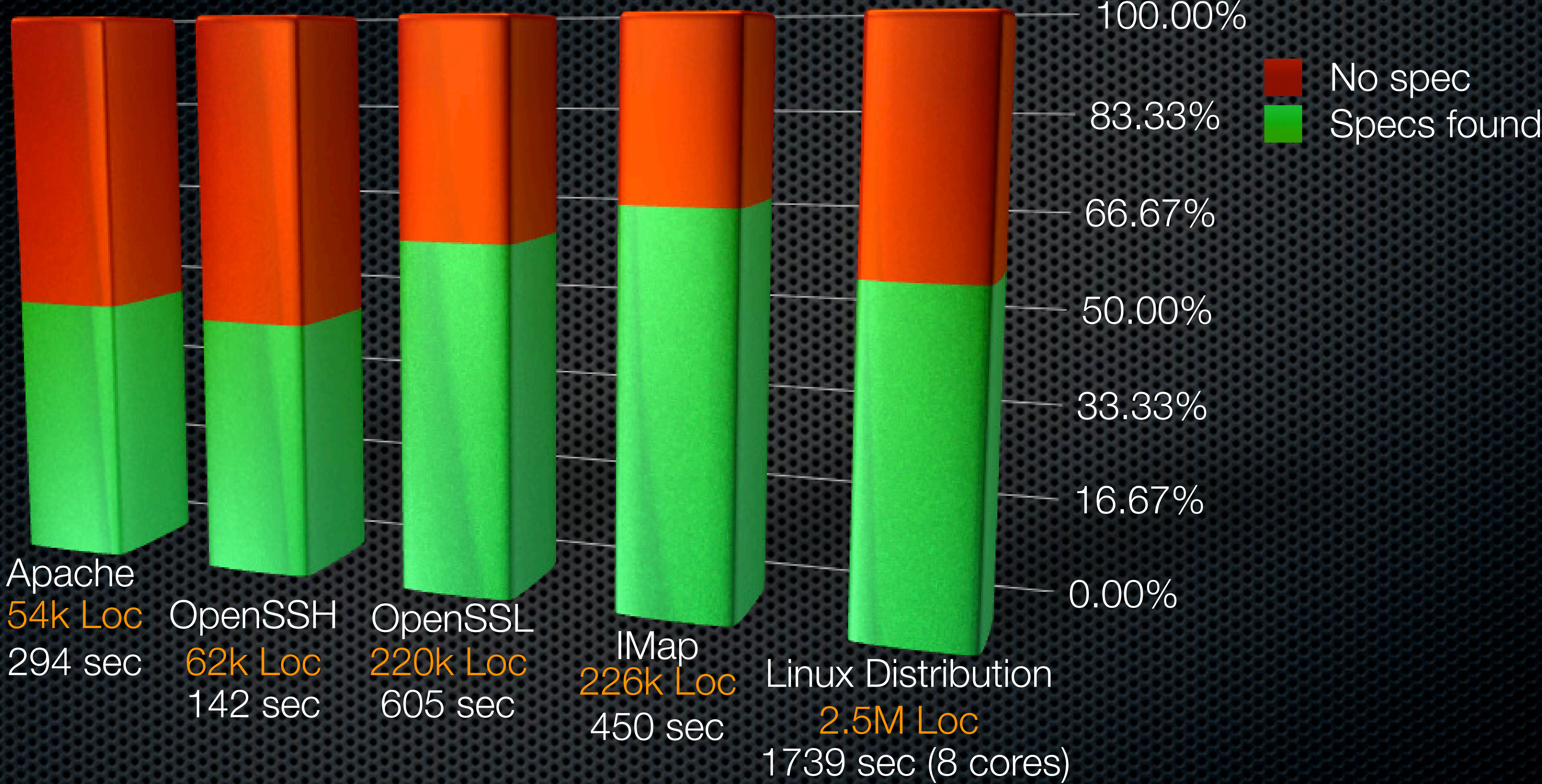
$$\{T_{f_1}, \ldots, T_{f_n}\}$$

Tables are sets of entries of type: $(pre, \{post_1, post_2, \ldots\})$

The computation follows the call graph (start from leaves)

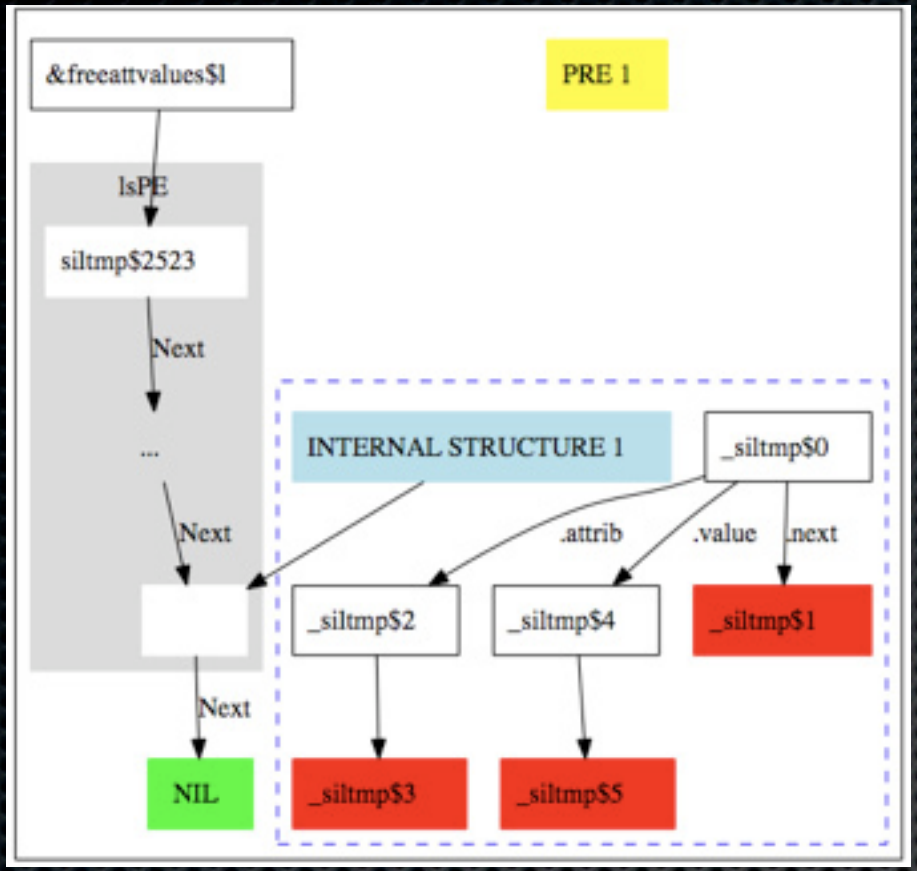
Recursive function are analyzed with an iterative method until it reaches fixed point

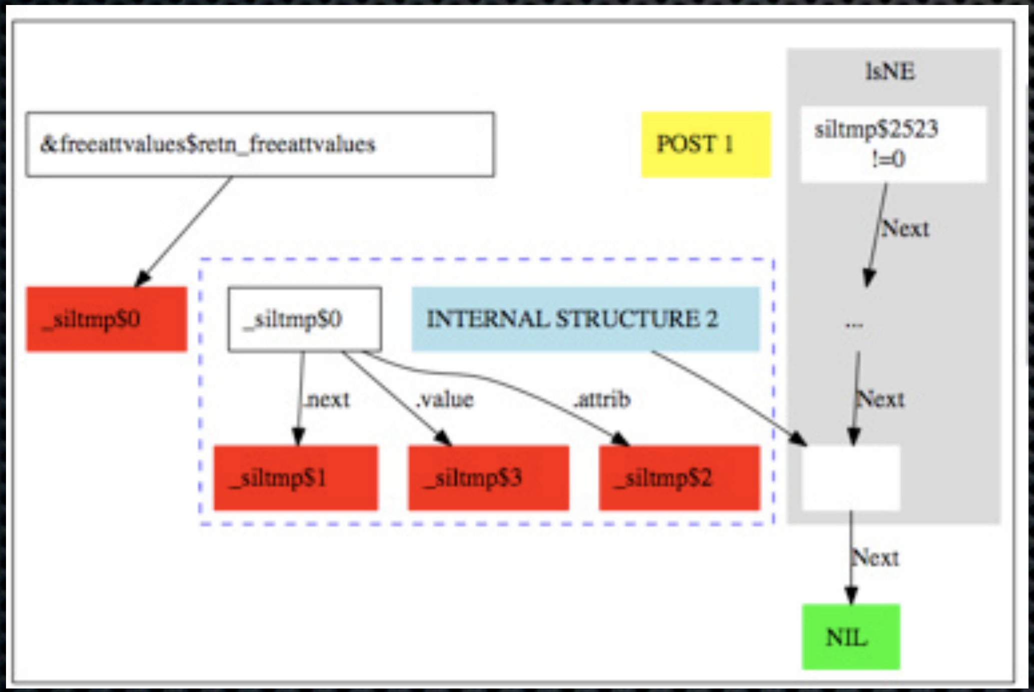
Running on really big code



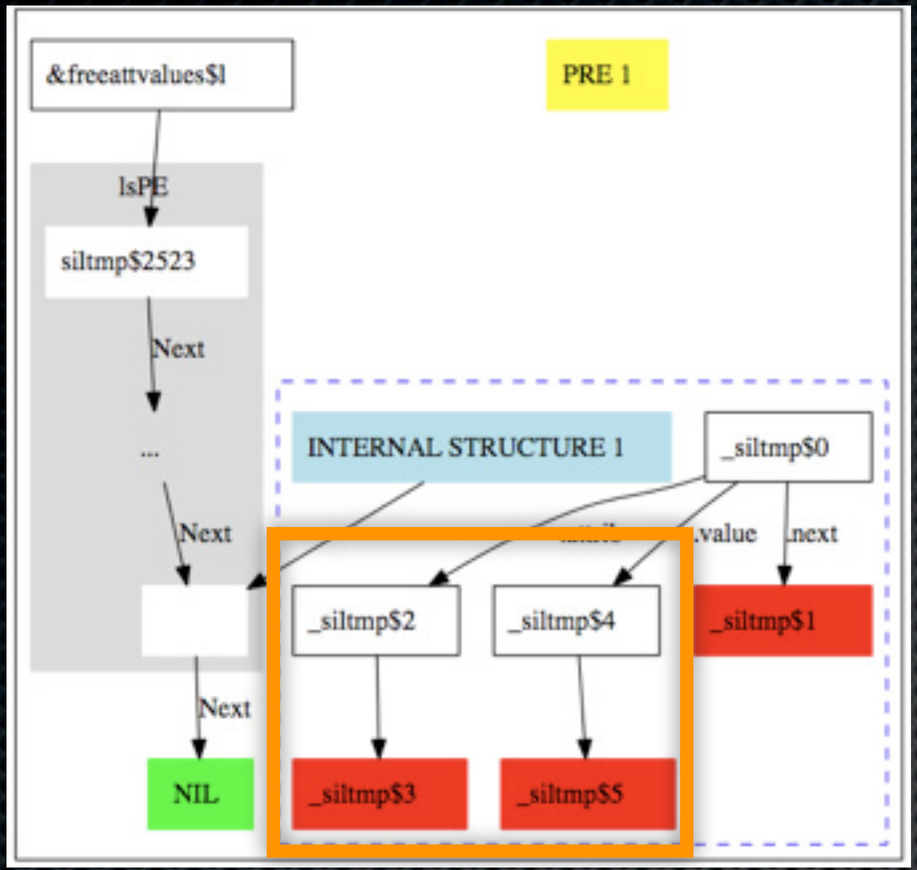
Test for precision: run on Firewire device driver and small recursive procedures handling nested data structures

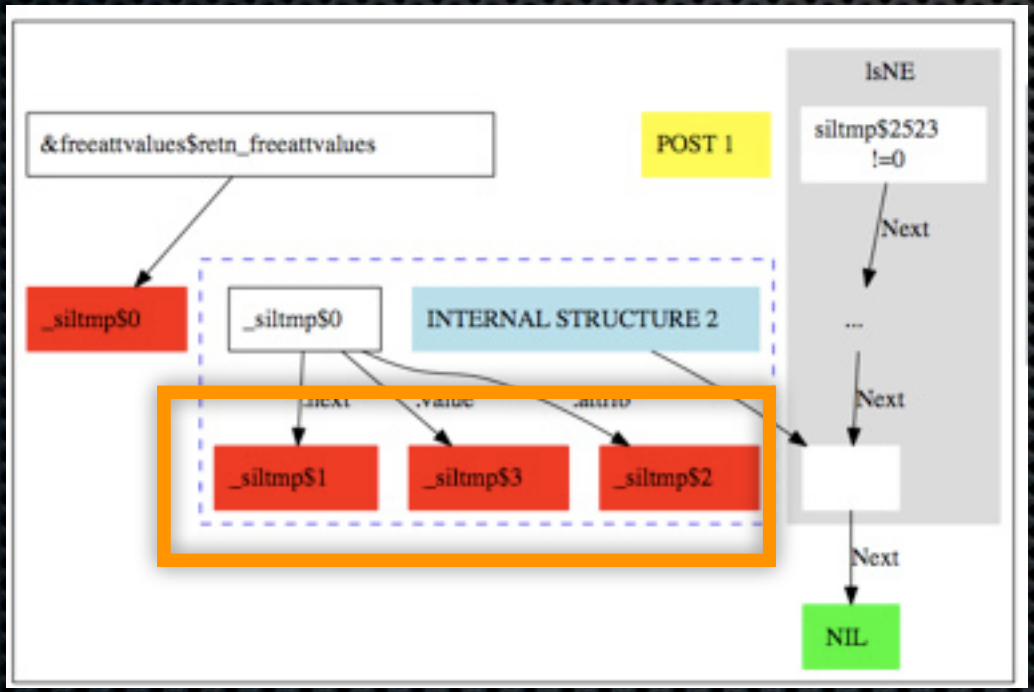
freeattvalues





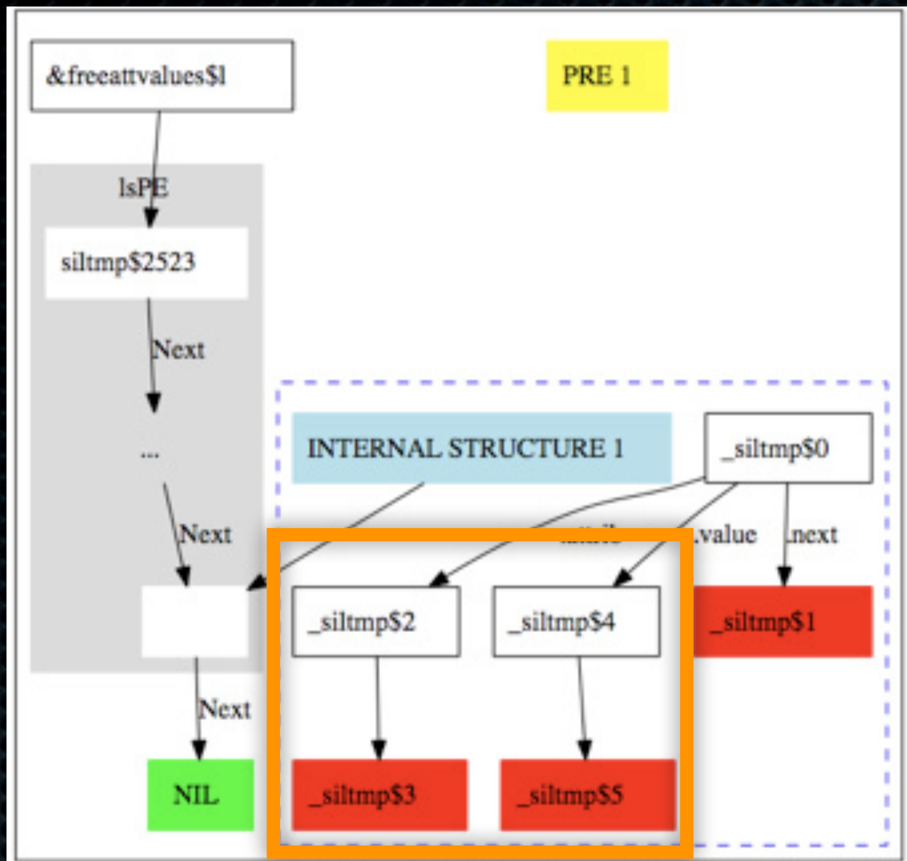
freeattvalues

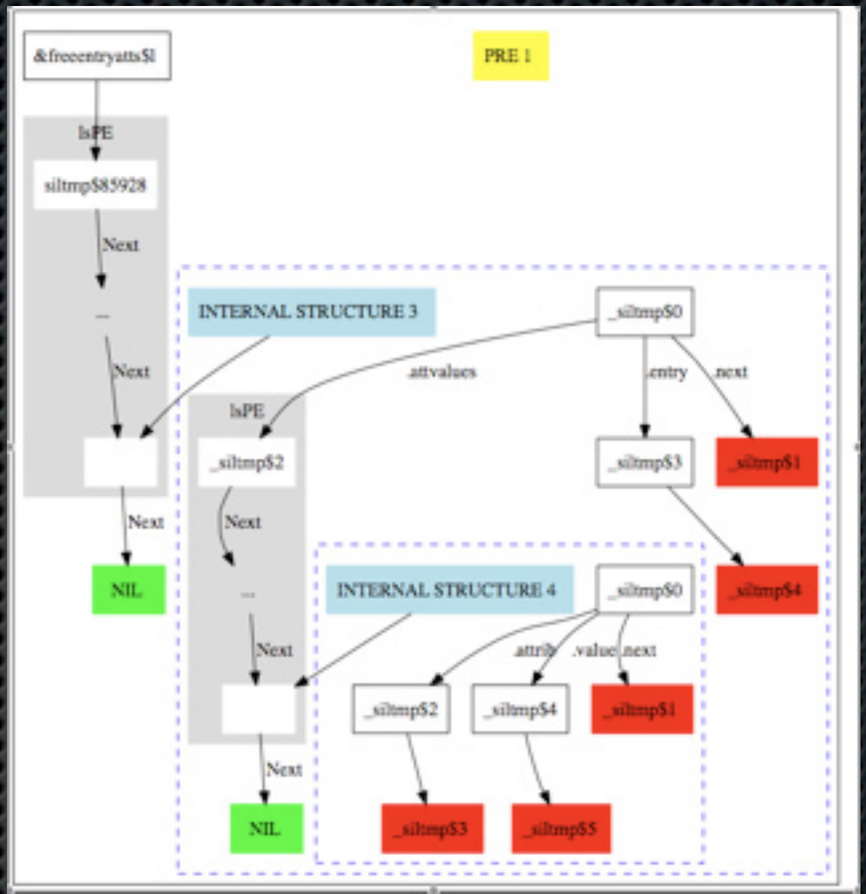


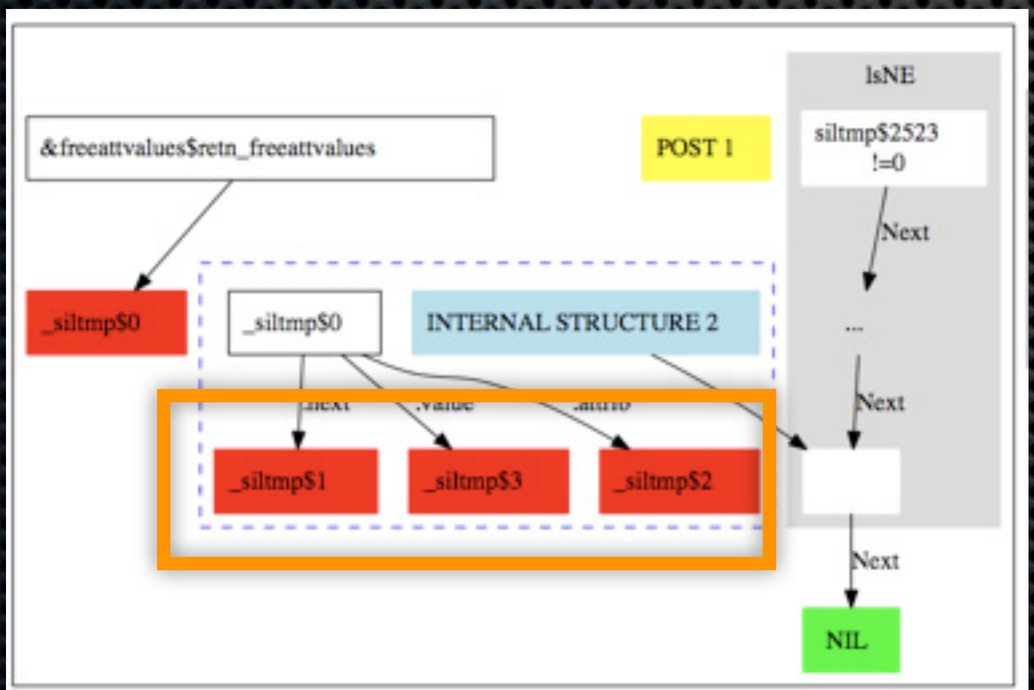


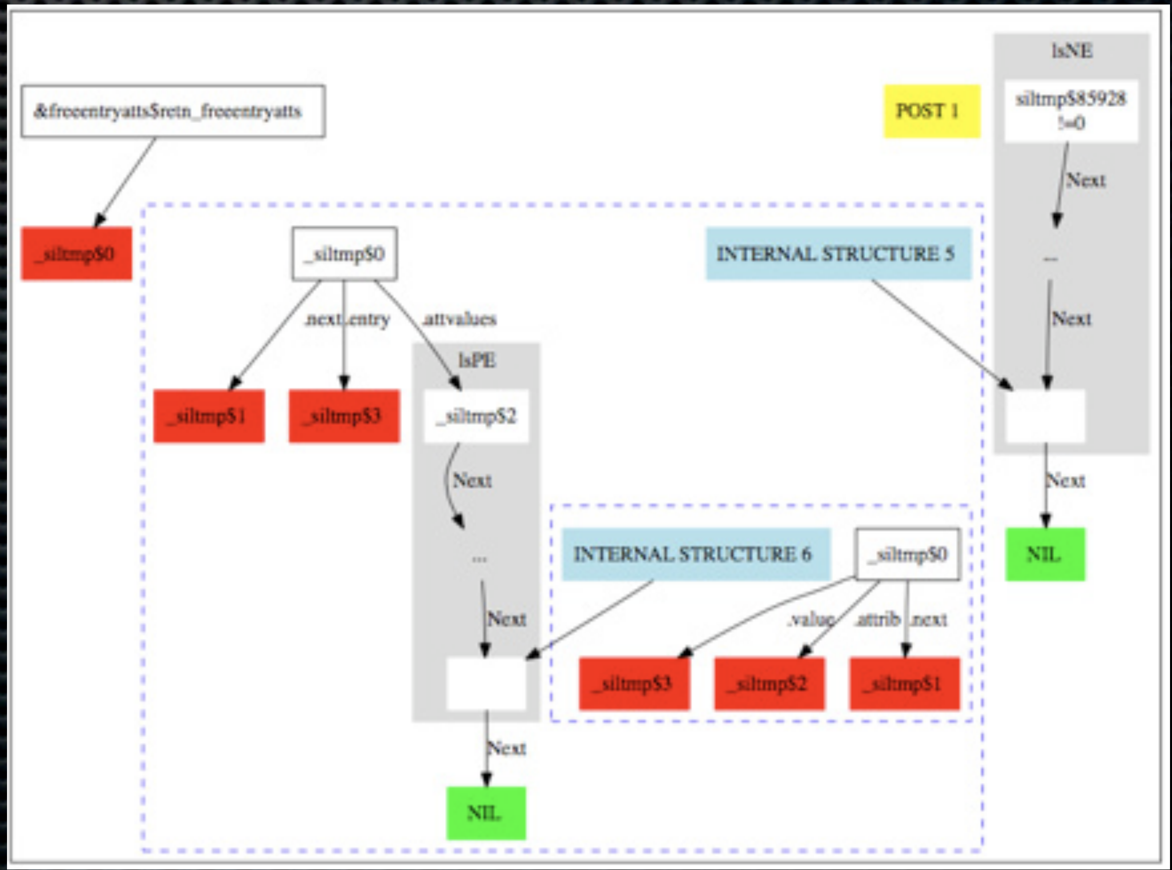
freeattvalues

freeentryatts

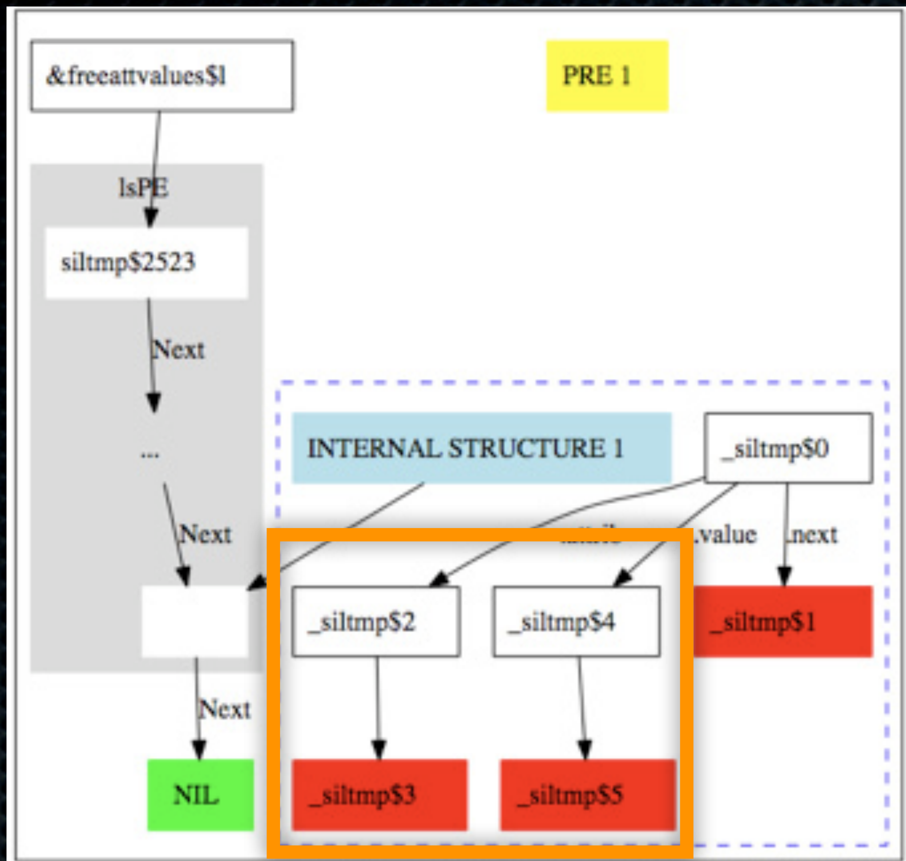


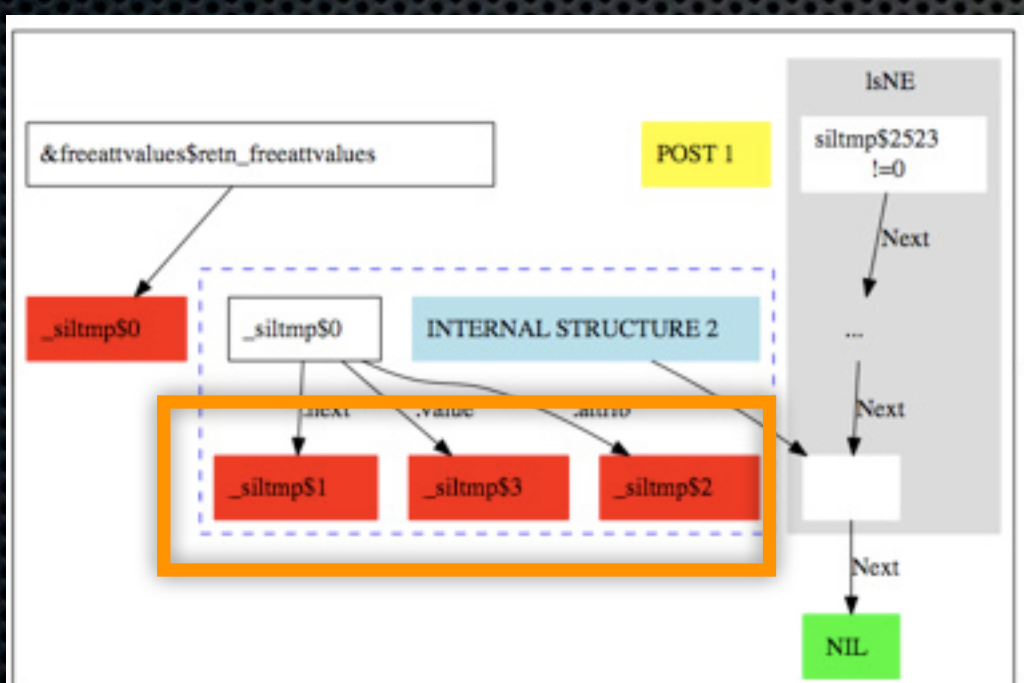




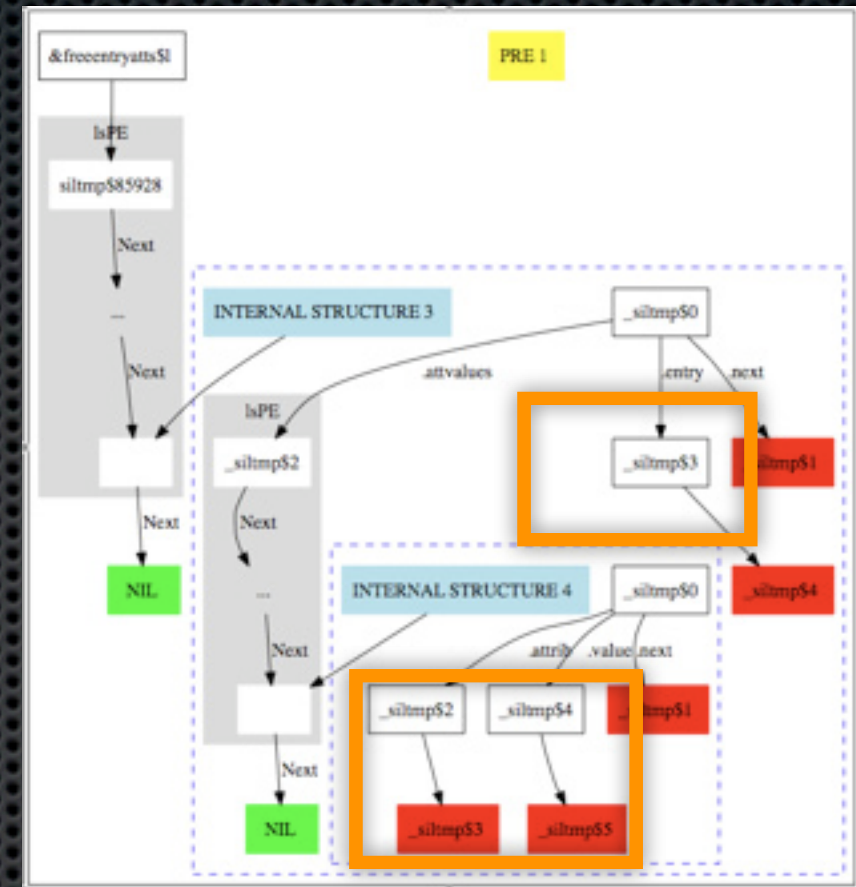


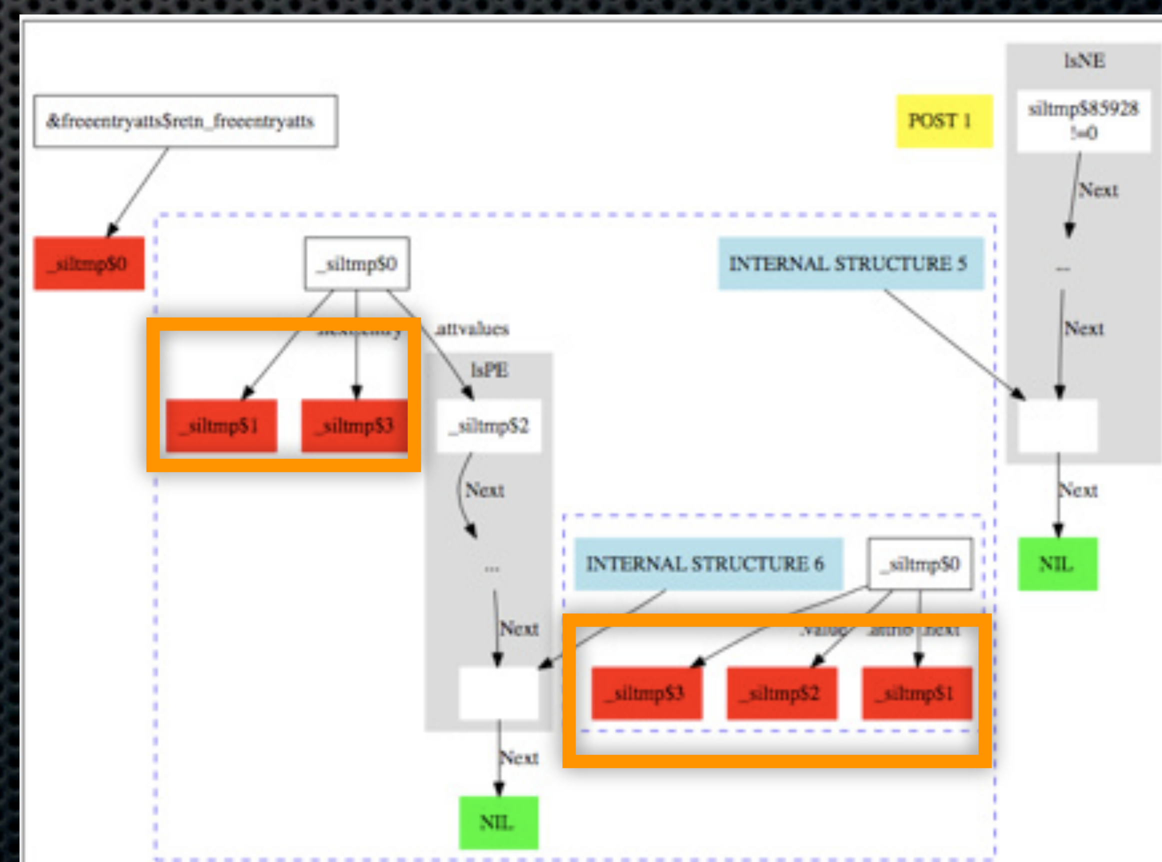
freeattvalues





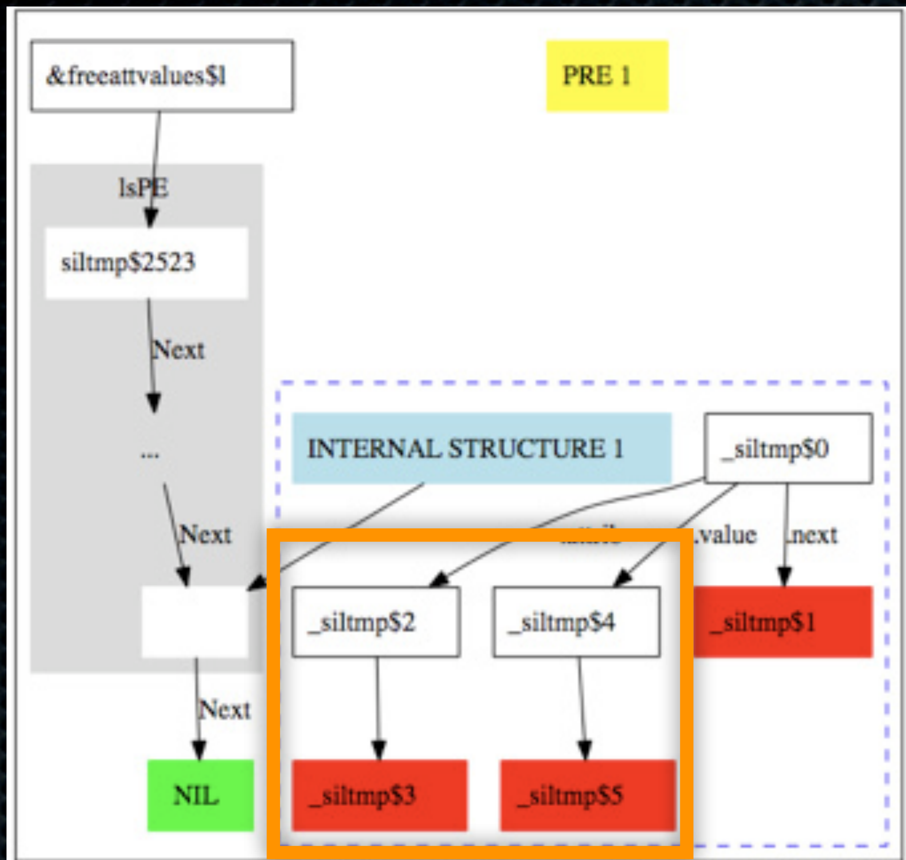
freeentryatts

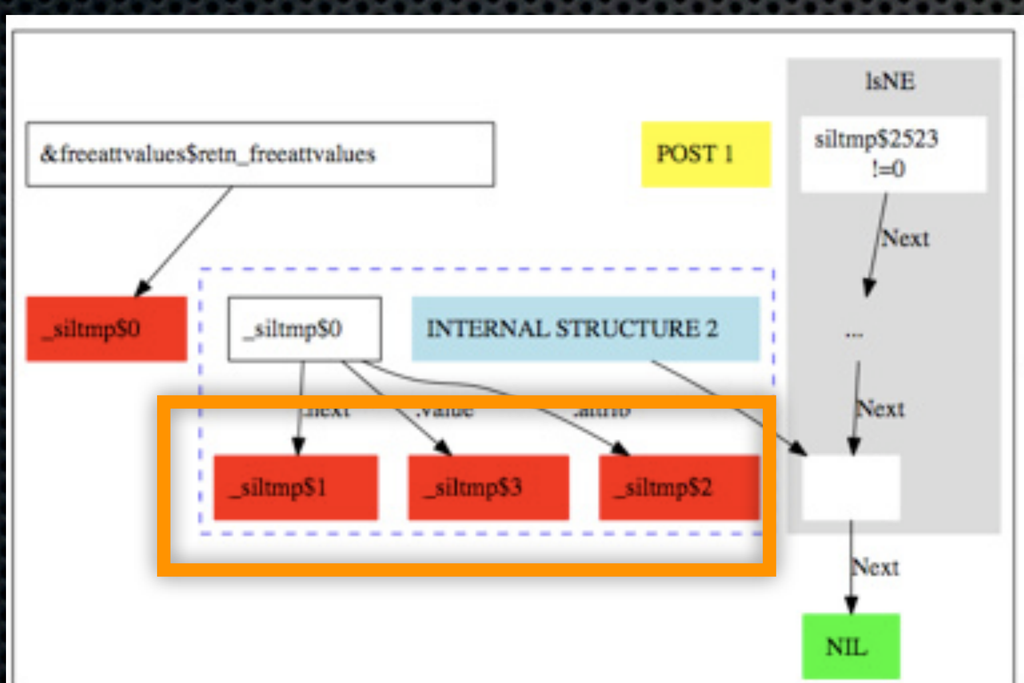


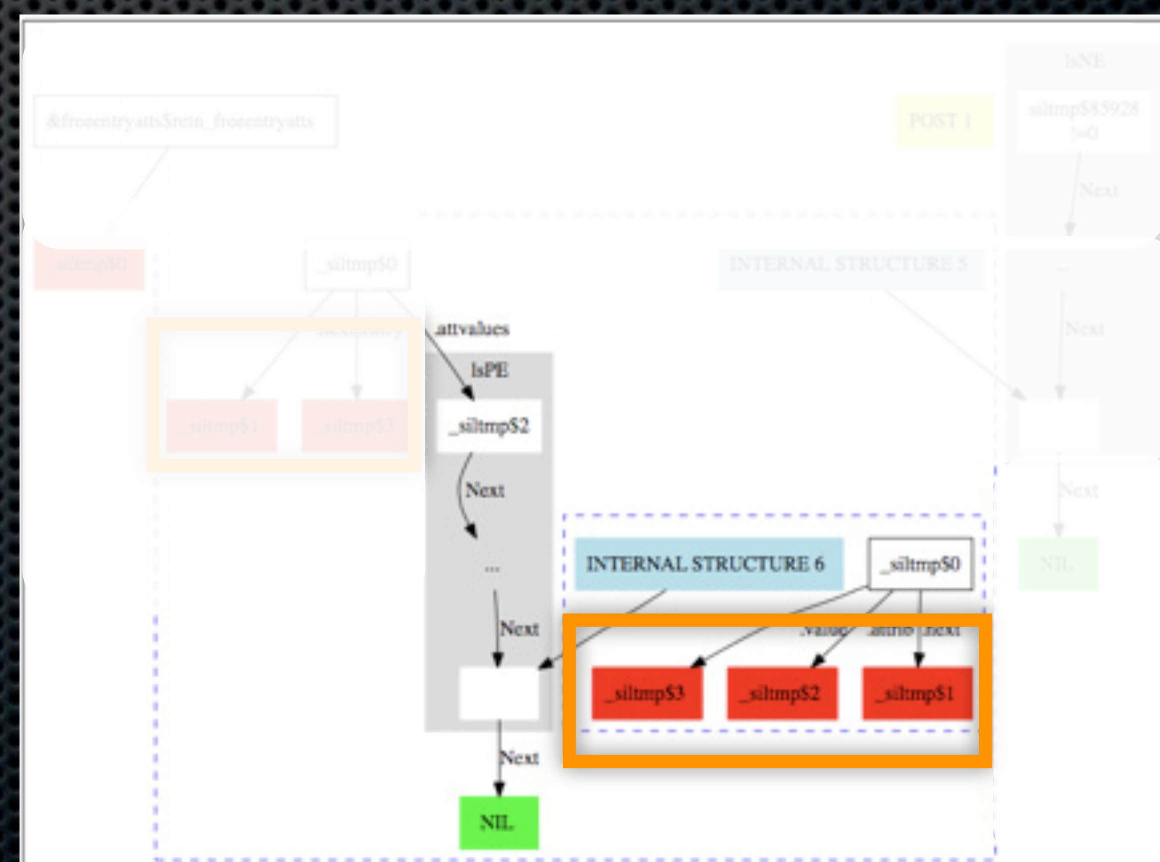


freeattvalues

freeentryatts

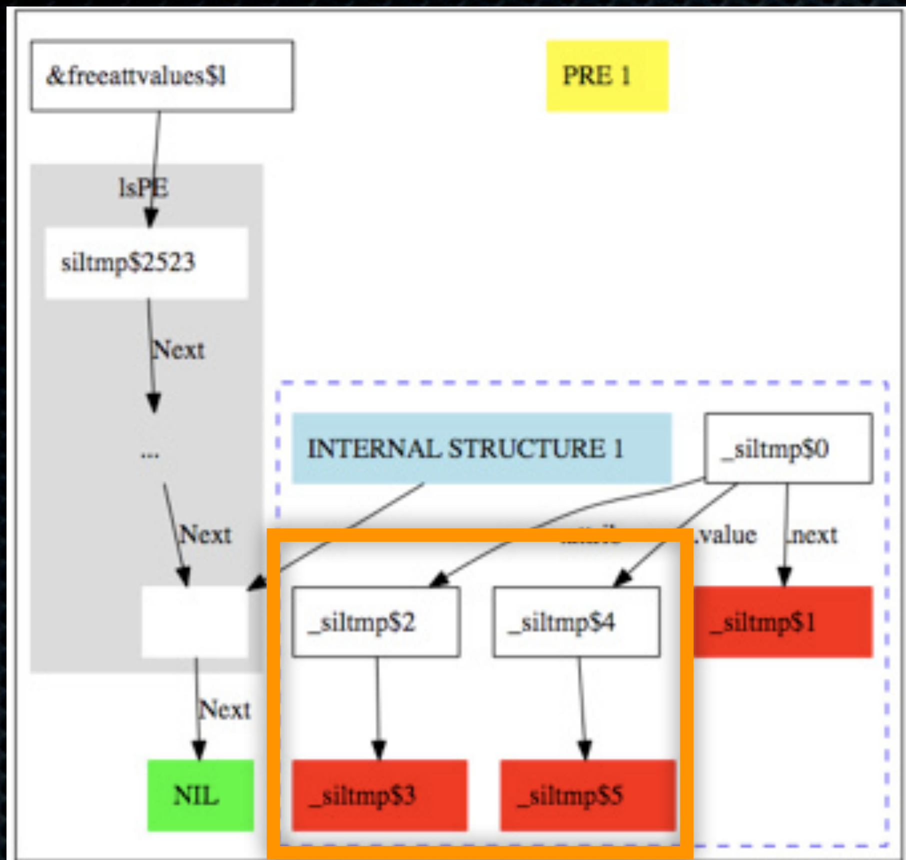


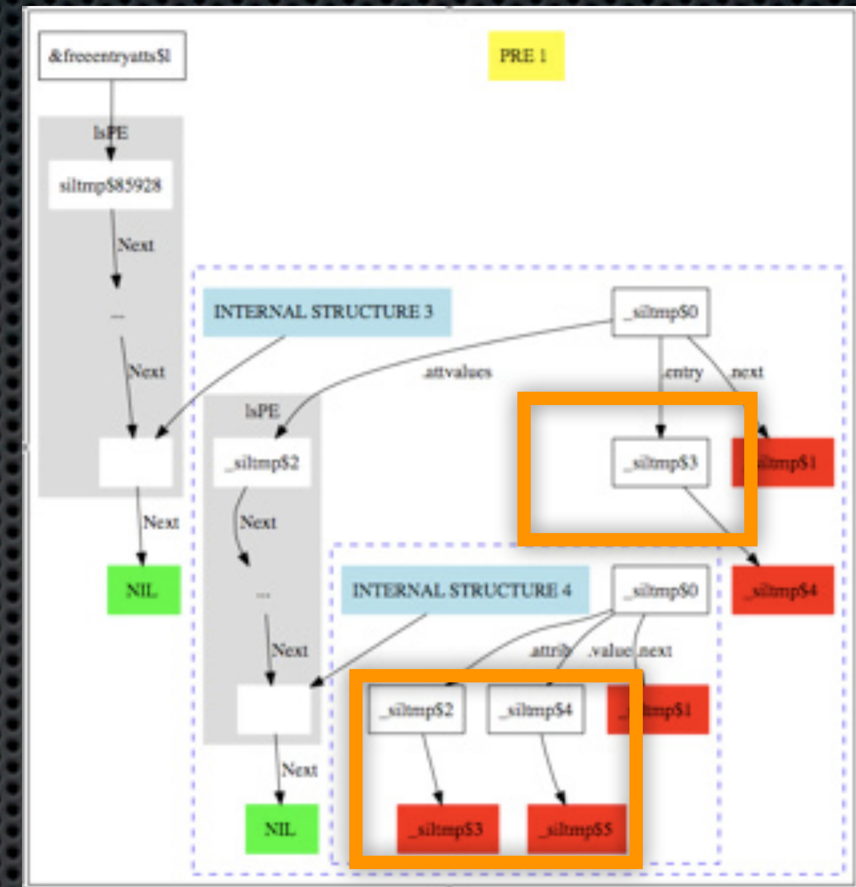


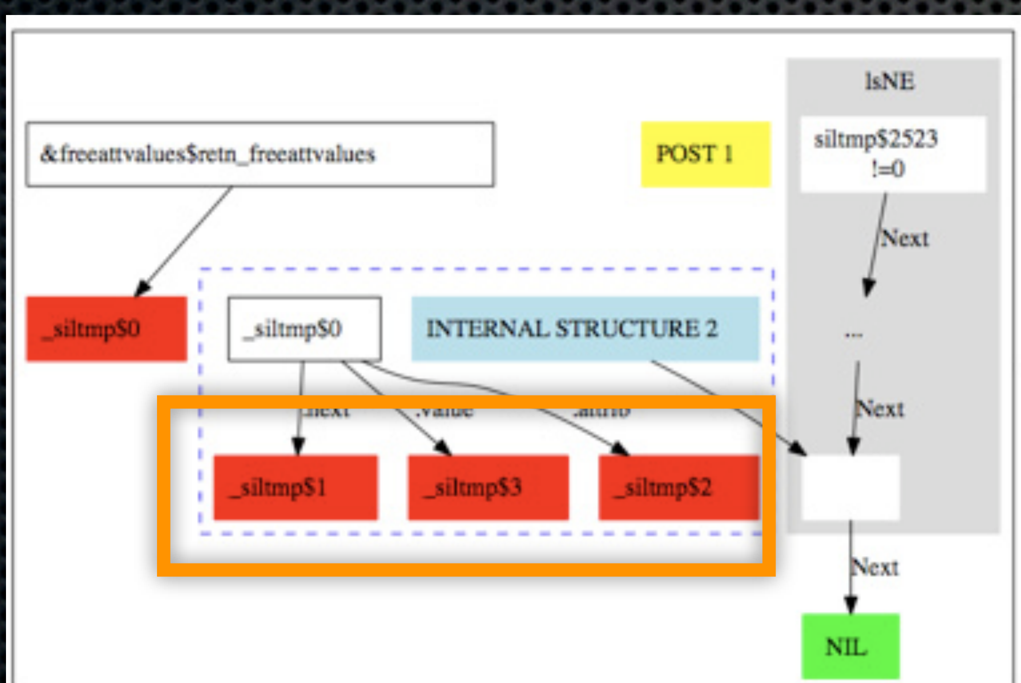


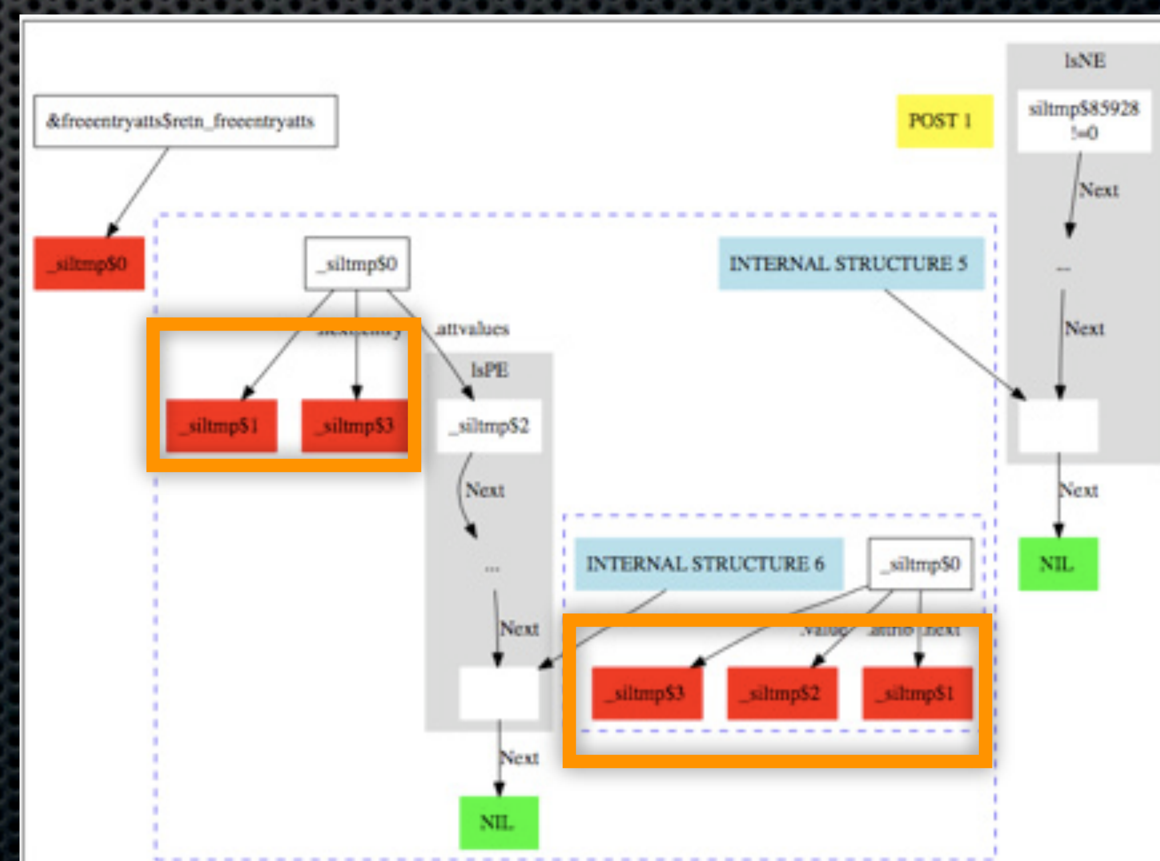
freeattvalues

freeentryatts









The bi-abduction manifesto

- Frame inference $A \vdash B * X$ allows an analyzer to use small specs
- Abduction $A * X \vdash B$ helps to synthesize small specs
- Their combination, bi-abduction

$$A * X \vdash B * X'$$

helps to achieve compositional bottom-up analysis. Furthermore it brings the benefits of local reasoning (as introduced in Separation Logic) to automatic program verification