An Introduction to Separation Logic (2/2)

Matthew Parkinson

Overview

First part (Last time) – Introduction

- Motivation
- In the beginning...
- The Logic
- Some examples

Second part (Today) – Harder stuff

- Modularity
- Concurrency
- ~~Decidability~~

Modularity

Overview

Frame rule allows modular proofs!

- Frame rule provides modularity in proofs
- Specifications are open to extension of new state.

Does not reflect the modularity of encapsulation of data-structures.

Overview

Frame rule allows modular proofs!

- Frame rule provides modularity in proofs
- Specifications are open to extension of new state.

Does not reflect the modularity of encapsulation of data-structures.

Two works in separation logic on adding modular procedure/function rules

- Static Modularity: Hypothetical Frame Rule [O'Hearn, Reynold and Yang, POPL 2004]
- Dynamic Modularity: Abstract Predicates [Parkinson and Bierman, POPL 2005]

We will use judgements of the form:

$$\Gamma \vdash \{P\}C\{Q\}$$

where

$$\Gamma \quad ::= \quad \{P\}\kappa\{Q\}, \Gamma \mid \epsilon$$

Procedure call

$$\Gamma, \{P\}\kappa\{Q\} \vdash \{P\}\kappa\{Q\}$$

Procedure definition

$$\frac{\Gamma, \{P_1\}\kappa_1\{Q_1\} \vdash \{P\}C\{Q\} \quad \Gamma \vdash \{P_1\}C_1\{Q_1\}}{\Gamma \vdash \{P\}let\ \kappa_1 = C_1\ in\ C\ end\{Q\}}$$

Will sometimes have parameters, or modifies clauses:

$$\Gamma \quad ::= \quad \{P\}\kappa(x)\{Q\}[X], \Gamma \mid \epsilon$$

Procedures with a hidden data-structure, e.g. Memory Manager with a free list.
Procedures with client specifications:

- $\{empty\}alloc\{x \mapsto \_, \_\}[x]$
- $\{x \mapsto \_, \_\}dealloc\{empty\}[]$

Expect to implement it using specifications involving free list:

- $\{list(f)\}alloc\{list(f) * x \mapsto \_, \_\}[x, f]$
- $\{x \mapsto \_, \_ * list(f)\}dealloc\{list(f)\}[f]$

Key Idea: verify procedures with additional state that client cannot use or depend on.

We want a rule of the following form

$$\frac{\Gamma, \{P_1\}\kappa_1\{Q_1\}[X_1], \ldots \vdash \{P\}C\{Q\} \qquad \Gamma \vdash \{P_1 * R\}C_1\{Q_1 * R\} \quad \ldots}{\Gamma \vdash \{P * R\}\ \texttt{let}\ \kappa_1 = C_1, \ldots\ \texttt{in}\ C\{Q * R\}}$$

where

- C does not modify free variables in R except using procedures $\kappa_1, \ldots, \kappa_n$.
- C_i only modifies X_i and Y
- Y is disjoint from P, Q and C.

Returning to the motivating example:

$$\frac{\begin{array}{c}\Gamma, \{emp\}malloc\{x \mapsto \_, \_\}[x], \ldots \vdash \{P\}C\{Q\} \\ \Gamma \vdash \{emp * list(f)\}C_1\{(x \mapsto \_, \_) * list(f)\} \\ \Gamma \vdash \{(x \mapsto \_, \_) * list(f)\}C_2\{emp * list(f)\}\end{array}}{\Gamma \vdash \{P * list(f)\}\ \texttt{let}\ \ malloc = C_1, free = C_2\ \ \texttt{in}\ \ C\{Q * list(f)\}}$$

So far:

- "Ownership transfer"
- Static modularity: single instance of hidden data structure

Cannot instantiate several memory managers, e.g. connection pooling.

$\left\{empty\right\}$ `consPool(db)` $\left\{cpool(ret, db)\right\}$

$\left\{cpool(x, db)\right\}$ `getConn(x)` $\left\{cpool(x, db) * conn(ret, db)\right\}$

$\left\{cpool(x, db) * conn(y, db)\right\}$ `freeConn(x,y)` $\left\{cpool(x, db)\right\}$

Abstract Predicates

Need to scope predicate definitions to support ADTs

Need to scope predicate definitions to support ADTs

1. Define judgements

$$\Lambda; \Gamma \vdash \{P\}C\{Q\}$$

where Λ is an abstract predicate environment.

e.g.

$$\Lambda := (\alpha_1(\overline{x_1}) \stackrel{\text{def}}{=} P_1), \ldots, \ (\alpha_n(\overline{x_n}) \stackrel{\text{def}}{=} P_n)$$

A mapping from predicate name, α, to formula, P.

Need to scope predicate definitions to support ADTs

1. Define judgements
$$\Lambda; \Gamma \vdash \{P\}C\{Q\}$$
2. The predicate definitions are used with the rule of consequence

$$\frac{\Lambda \models P \Rightarrow P' \quad \Lambda; \Gamma \vdash \{P'\}C\{Q'\} \quad \Lambda \models Q' \Rightarrow Q}{\Lambda; \Gamma \vdash \{P\}C\{Q\}}$$

with two axioms: open and close

$$(\alpha(\overline{x}) \stackrel{\text{def}}{=} P), \Lambda \models \alpha(\overline{E}) \Rightarrow P[\overline{E}/\overline{x}]$$

$$(\alpha(\overline{x}) \stackrel{\text{def}}{=} P), \Lambda \models P[\overline{E}/\overline{x}] \Rightarrow \alpha(\overline{E})$$

Intuitively, open and close can be seen as pack and unpack for abstract data types.

3. Provide rule to allow predicates to be introduced and scoped

$$\frac{\Lambda';\ \Gamma,\ \{P_1\}k_1(\overline{x_1})\{Q_1\},\cdots \vdash \{P\}C\{Q\} \qquad \Lambda,\ \Lambda';\ \Gamma \vdash \{P_1\}C_1\{Q_1\} \qquad \cdots}{\Lambda';\Gamma \vdash \{P\}\mathtt{let}\ \ k_1(\overline{x_1}) = C_1,\cdots\ \ \mathtt{in}\ \ C\{Q\}}$$

C is verified without knowing the definitions in Λ hence it can not modify things defined in it except through $\{k_1,\ldots\}$.

Example: Connection Pool

We define two abstract predicates for the connection pool module: $cpool$ and $clist$.

$$cpool(x, db) \stackrel{\textbf{def}}{=} \exists i.x \mapsto i, db * clist(i, db)$$

$$clist(x, db) \stackrel{\textbf{def}}{=} x \doteq null \vee (\exists ij.x \mapsto i, j \; * conn(i, db) * clist(j, db))$$

where $E \doteq E'$ is a shorthand for $E = E' \wedge empty$.

Connection pool

```
let
  consPool db :=
    (newvar p; p=cons(null,db); return p)

  getConn x := (newvar n,c,l,p; l=[x];
    if (l == null) then
      p=[x+1]; c=consConn(p)
    else (c=[l]; n=[l+1]; dispose(l);
          dispose(l+1); [x]=n);
    return c)

  freeConn x y :=
    (newvar t,n; t=[x]; n=cons(y,t); [x]=n)
in
  C
```

Let us verify the constructor

$$\{empty\}$$
$$\{empty\}$$
$$p = cons(null, db); return\ p;$$
$$\{ret \mapsto null, db\}$$
$$\{Pool(ret, db)\}$$

Here we must prove

$$\Lambda \models (ret \mapsto null, db) \Rightarrow Pool(this, db)$$

From the definition of $ConnList$ and $Pool$ we know

$$\Lambda \models empty \Rightarrow ConnList(null, db)$$
$$\Lambda \models (ret \mapsto null, db) * ConnList(null, db) \Rightarrow Pool(this, db)$$

Client Example

$$
\begin{array}{l}
\{empty\} \\
\quad x = consPool(db); \\
\quad y = getConn(x); \\
\quad UseConn(y); \\
\quad freeConn(x, y); \\
\quad UseConn(y) \\
\{true\}
\end{array}
$$

Client Example

$$\{empty\}$$
$$\quad x = consPool(db);$$
$$\{Pool(x, db)\}$$
$$\quad y = getConn(x);$$
$$\quad UseConn(y);$$
$$\quad freeConn(x, y);$$
$$\quad UseConn(y)$$
$$\{true\}$$

Client Example

$$\{empty\}$$
$$x = consPool(db);$$
$$\{Pool(x, db)\}$$
$$y = getConn(x);$$
$$\{Pool(x, db) * Conn(y, db)\}$$
$$UseConn(y);$$
$$freeConn(x, y);$$
$$UseConn(y)$$
$$\{true\}$$

Client Example

$$\{empty\}$$
$$\quad x = consPool(db);$$
$$\{Pool(x, db)\}$$
$$\quad y = getConn(x);$$
$$\{Pool(x, db) * Conn(y, db)\}$$
$$\quad \{Conn(y, db)\}$$
$$\quad UseConn(y);$$
$$\quad \{Conn(y, db)\}$$
$$\{Pool(x, db) * Conn(y, db)\}$$
$$\quad freeConn(x, y);$$
$$\quad UseConn(y)$$
$$\{true\}$$

$$
\begin{array}{l}
\{empty\} \\
\quad x = consPool(db); \\
\{Pool(x, db)\} \\
\quad y = getConn(x); \\
\{Pool(x, db) * Conn(y, db)\} \\
\quad \{Conn(y, db)\} \\
\quad UseConn(y); \\
\quad \{Conn(y, db)\} \\
\{Pool(x, db) * Conn(y, db)\} \\
\quad freeConn(x, y); \\
\{{\color{blue}Pool(x, db)}\} \\
\quad UseConn(y) \\
\{true\}
\end{array}
$$

Client Example

$$\{empty\}$$
$$\quad x = consPool(db);$$
$$\{Pool(x, db)\}$$
$$\quad y = getConn(x);$$
$$\{Pool(x, db) * Conn(y, db)\}$$
$$\quad \{Conn(y, db)\}$$
$$\quad UseConn(y);$$
$$\quad \{Conn(y, db)\}$$
$$\{Pool(x, db) * Conn(y, db)\}$$
$$\quad freeConn(x, y);$$
$$\{Pool(x, db)\}$$
$$\quad \color{red}{UseConn(y)}$$
$$\{true\}$$

Client Example

$$\{empty\}$$
$$\quad x = consPool(db);$$
$$\{Pool(x, db)\}$$
$$\quad y = getConn(x);$$
$$\{Pool(x, db) * Conn(y, db)\}$$
$$\quad \{Conn(y, db)\}$$
$$\quad UseConn(y);$$
$$\quad \{Conn(y, db)\}$$
$$\{Pool(x, db) * Conn(y, db)\}$$
$$\quad freeConn(x, y);$$
$$\{Pool(x, db)\}$$
$$\quad \color{red}{UseConn(y)}$$
$$\{true\}$$

X

Now let us consider specifying C-style **malloc** and **free**

$$\{empty\}\mathbf{malloc(n)}\{(ret \mapsto \_) * \ldots * (ret + n - 1 \mapsto \_)\}$$

$$\{(x \mapsto \_) * \ldots * (x + n - 1 \mapsto \_)\}\mathbf{free(x)}\{empty\}$$

but what is n in **free**'s specification? We don't want to write **free(x,n)** as that isn't how the code is written.

Now let us consider specifying C-style **malloc** and **free**

$$\{empty\}\mathbf{malloc}(\mathbf{n})\{(ret \mapsto \_) * \ldots * (ret + n - 1 \mapsto \_)\}$$

$$\{(x \mapsto \_) * \ldots * (x + n - 1 \mapsto \_)\}\mathbf{free}(\mathbf{x})\{empty\}$$

but what is n in **free**'s specification? We don't want to write **free(x,n)** as that isn't how the code is written.

Write specifications using APs:

$$\{empty\}\mathbf{malloc}(\mathbf{n})\{Block(ret, n) * ret \mapsto \_ * \ldots ret + n - 1 \mapsto \_\}$$

$$\{Block(x, n) * x \mapsto \_ * \ldots * x + n - 1 \mapsto \_\}\mathbf{free}(\mathbf{x})\{empty\}$$

Here $Block$ is not known to the client and can be defined as

$$Block(x, n) \stackrel{\text{def}}{=} x - 1 \mapsto n$$

Concurrency

- Read sharing

$$E \mapsto_z E' * E \mapsto_{z'} E' \Leftrightarrow E \mapsto_{z+z'} E'$$

- Parallel rule

$$\frac{\{P_1\}C_1\{Q_1\} \quad \{P_2\}C_2\{Q_2\}}{\{P_1 * P_2\}C_1||C_2\{Q_1 * Q_2\}}$$

- Resource rule

$$\frac{\{P\}C\{Q\}}{\{P * I_r\}\text{resource } r \text{ in } C\{Q * I_r\}}$$

- CCR rule

$$\frac{\{P * I_r \wedge B\}C\{Q * I_r\}}{\{P\}\text{with } r \text{ when } B \text{ in } C\{Q\}}$$

Some side-conditions but we can ignore them in this talk

$\{emp\}$

```
lock() {
  local status = false;
  while (! status)
    status = CAS(LOCK,0,1);
}
```

$\{P\}$

$\{P\}$

```
unlock() {
  LOCK=0;
}
```

$\{emp\}$

Example: simple lock (1/3)

$\{emp\}$

```
lock() {
  local status = false;
  while (!status)
    with r do{
      status = CAS(LOCK,0,1);
    }
}
```

$\{P\}$

$\{P\}$

```
unlock() {
  with r do{ LOCK=0; }
}
```

$\{emp\}$

- $I_r \stackrel{def}{=} (\text{LOCK} = 0 \wedge P) \vee (\text{LOCK} = 1 \wedge emp)$
- $P * P \Rightarrow false$

Example: simple lock (2/3)

$$\frac{\{I_r\}\text{status = CAS(LOCK,0,1);}\{I_r * (status \dot{\Rightarrow} P)\}}{\{emp\}\text{with r do \{ status = CAS(LOCK,0,1);\}}\{(status \dot{\Rightarrow} P)\}}$$

$$\frac{\dfrac{\{(\text{LOCK} = 1 \wedge emp) * P\}\text{LOCK = 0;}\{(\text{LOCK} = 0 \wedge P) * emp\}}{\{I_r * P\}\text{LOCK = 0;}\{I_r\}}}{\{P\}\text{with r do\{LOCK = 0;\}}\{emp\}}$$

- $I_r \stackrel{def}{=} (\text{LOCK} = 0 \wedge P) \vee (\text{LOCK} = 1 \wedge emp)$
- Proof requires $P * P \Rightarrow false$
- $I_r * P \Rightarrow \text{LOCK} = 1 \wedge P$

Example: simple lock (3/3)

$\{emp\}$

lock ();

$\{10 \mapsto \_\}$

t = [10] + 1;

$\{10 \mapsto t - 1\}$

[10] = t ;

$\{10 \mapsto t\}$

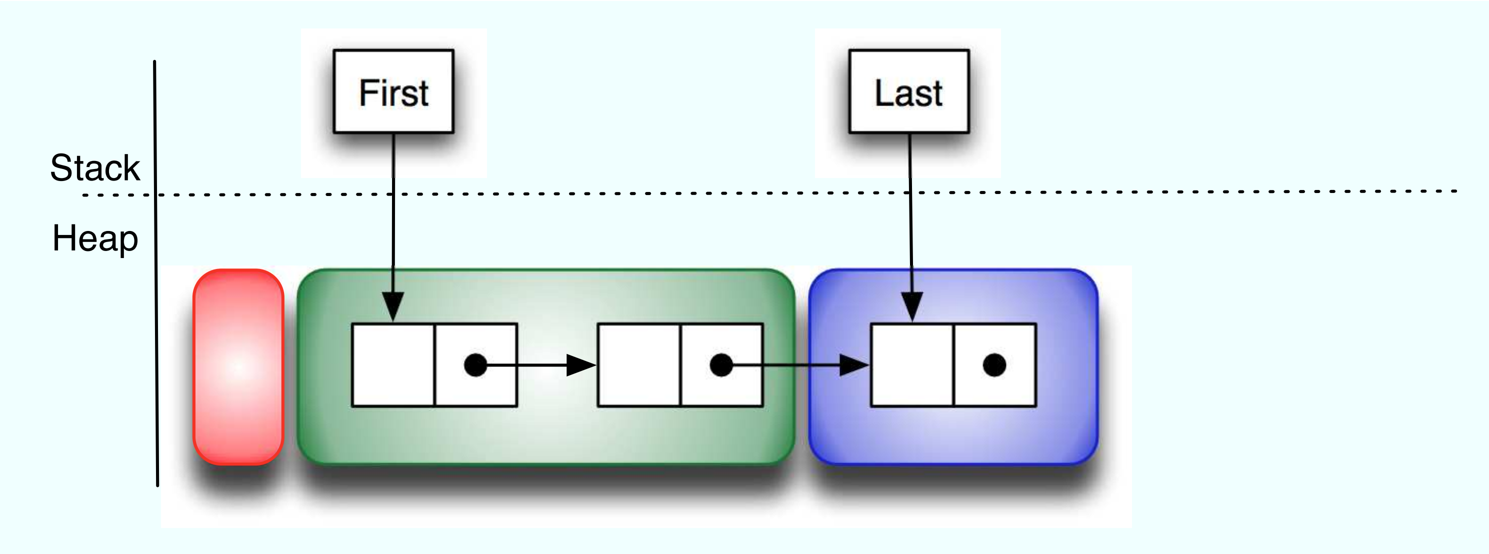
unlock ();

$\{emp\}$

where $P = 10 \mapsto \_$

Note: we have not had to worry at all about interference.

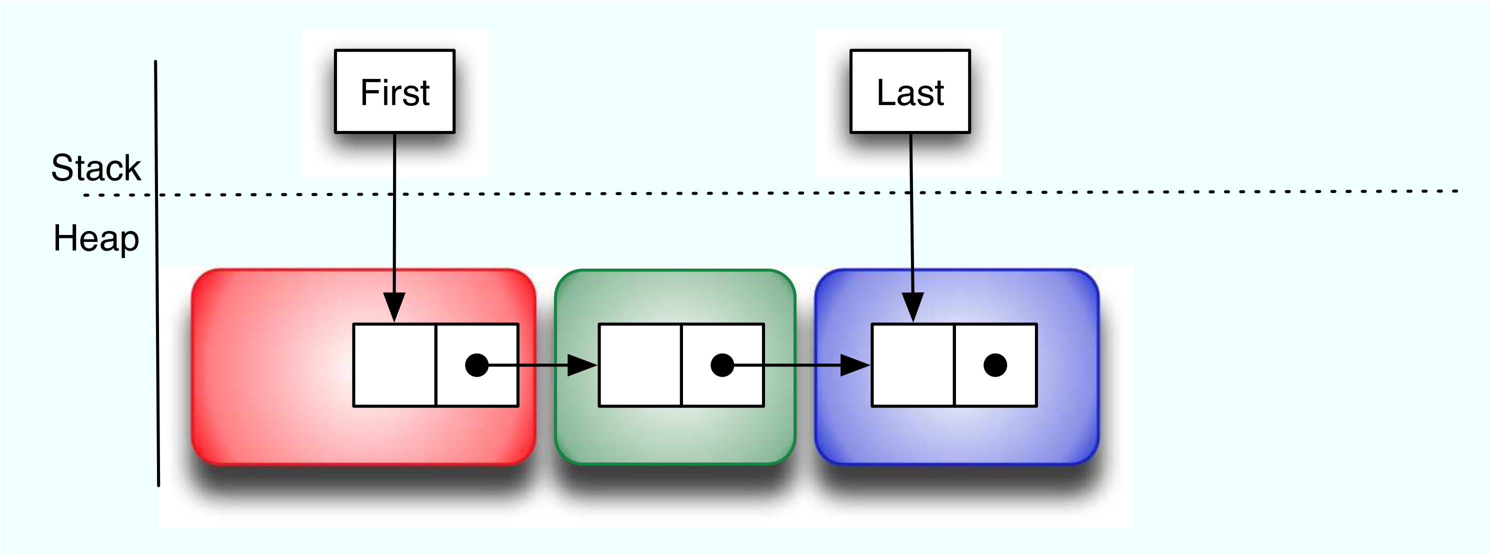
Example: Queue



semaphore number= 0;

P (number);	x = cons(*,*)
t = first	[last+1] = x;
first = [first+1]	last = [last+1]
dispose(t);	V(number);

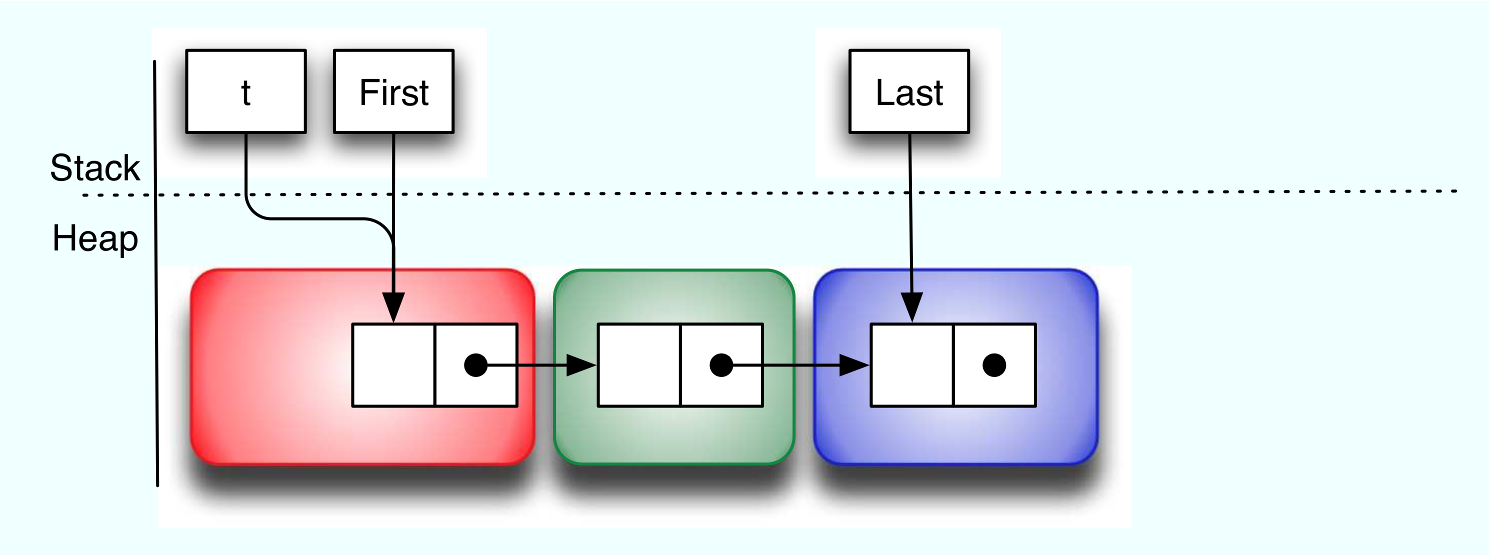
Example: Queue



semaphore number= 0;

P (number);	x = cons(*,*)
t = first	[last+1] = x;
first = [first+1]	last = [last+1]
dispose(t);	V(number);

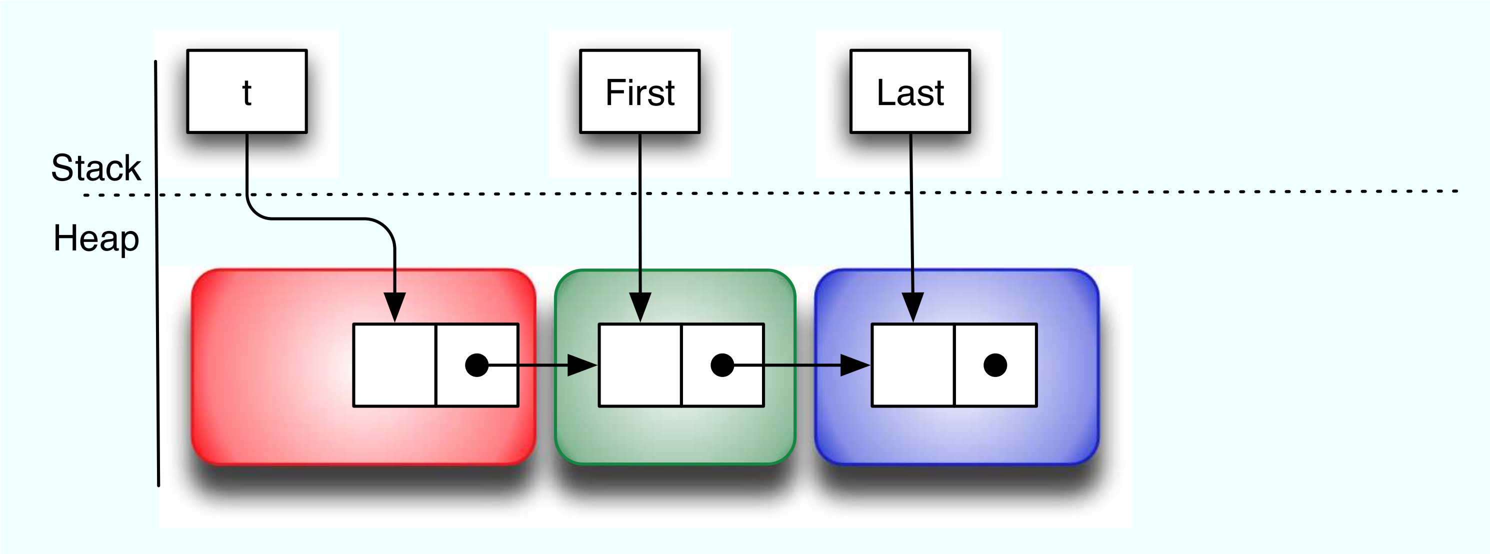
Example: Queue



semaphore number= 0;

P (number);	x = cons(*,*)
t = first	[last+1] = x;
first = [first+1]	last = [last+1]
dispose(t);	V(number);

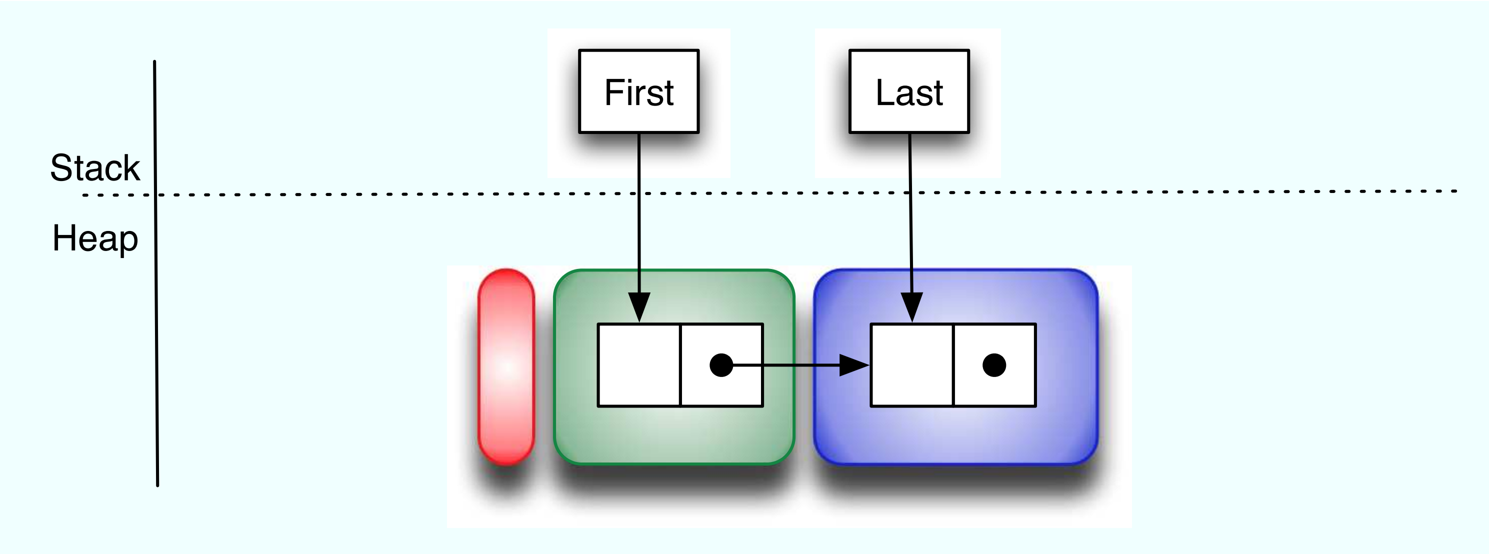
Example: Queue



semaphore number= 0;

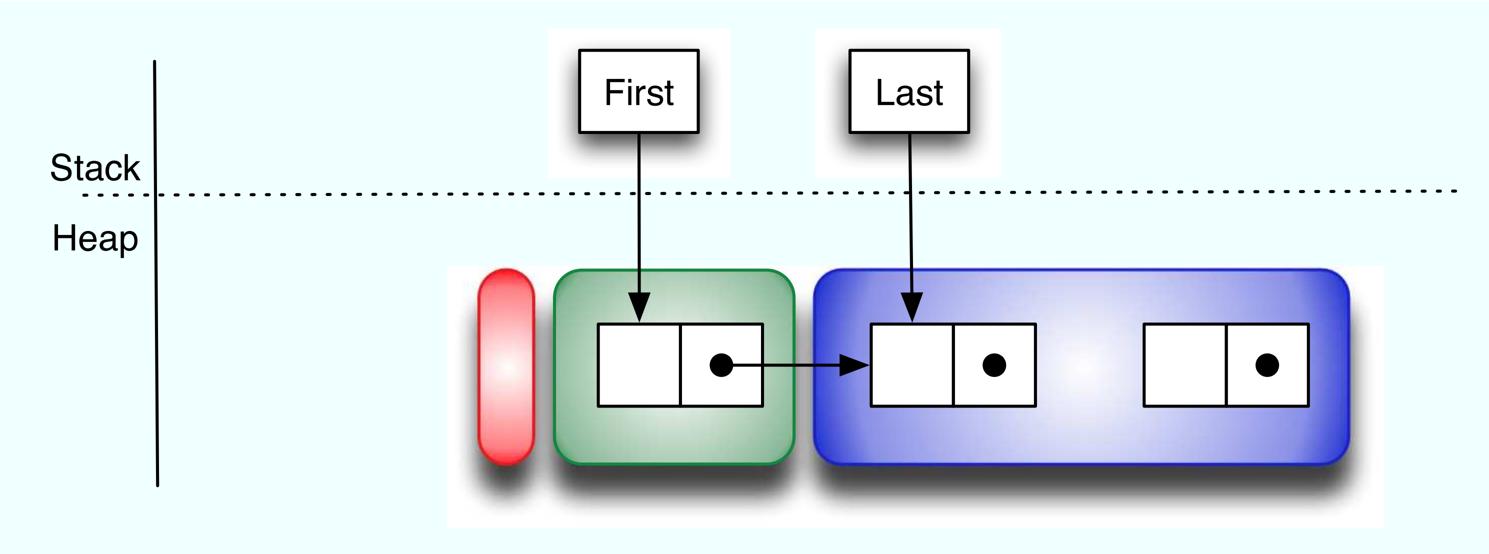
P (number);	x = cons(*,*)
t = first	[last+1] = x;
first = [first+1]	last = [last+1]
dispose(t);	V(number);

Example: Queue



semaphore number= 0;

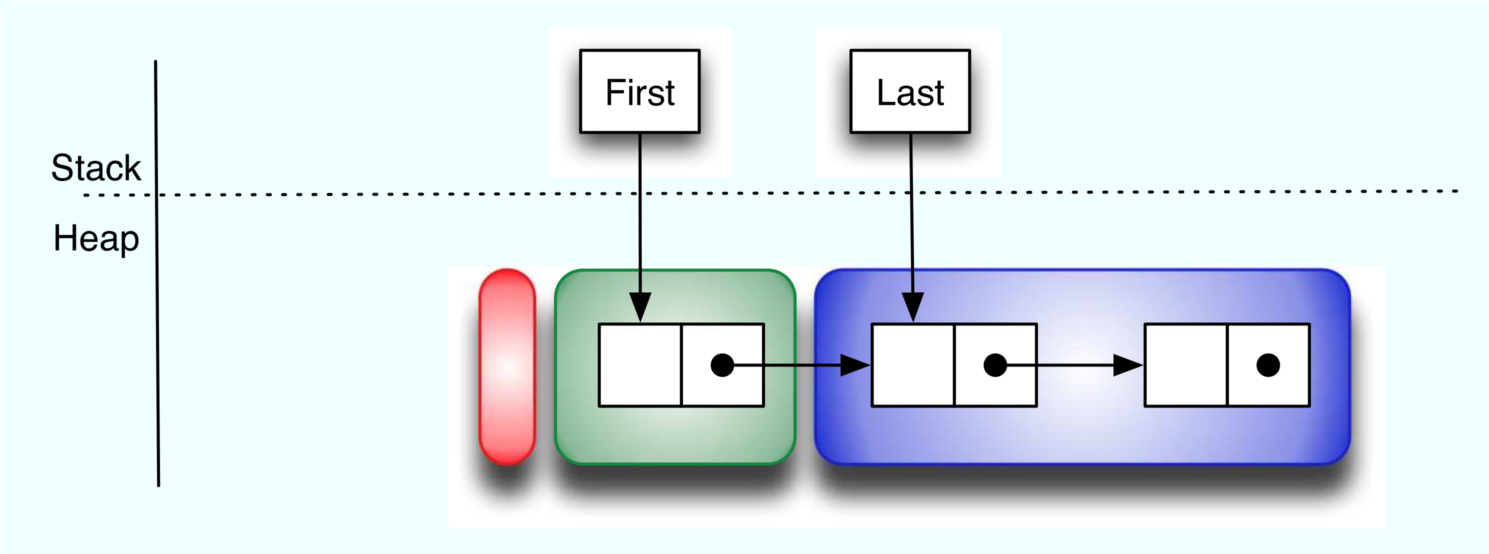
P (number);	x = cons(*,*)
t = first	[last+1] = x;
first = [first+1]	last = [last+1]
dispose(t);	V(number);



semaphore number= 0;

P (number);	x = cons(*,*)
t = first	[last+1] = x;
first = [first+1]	last = [last+1]
dispose(t);	V(number);

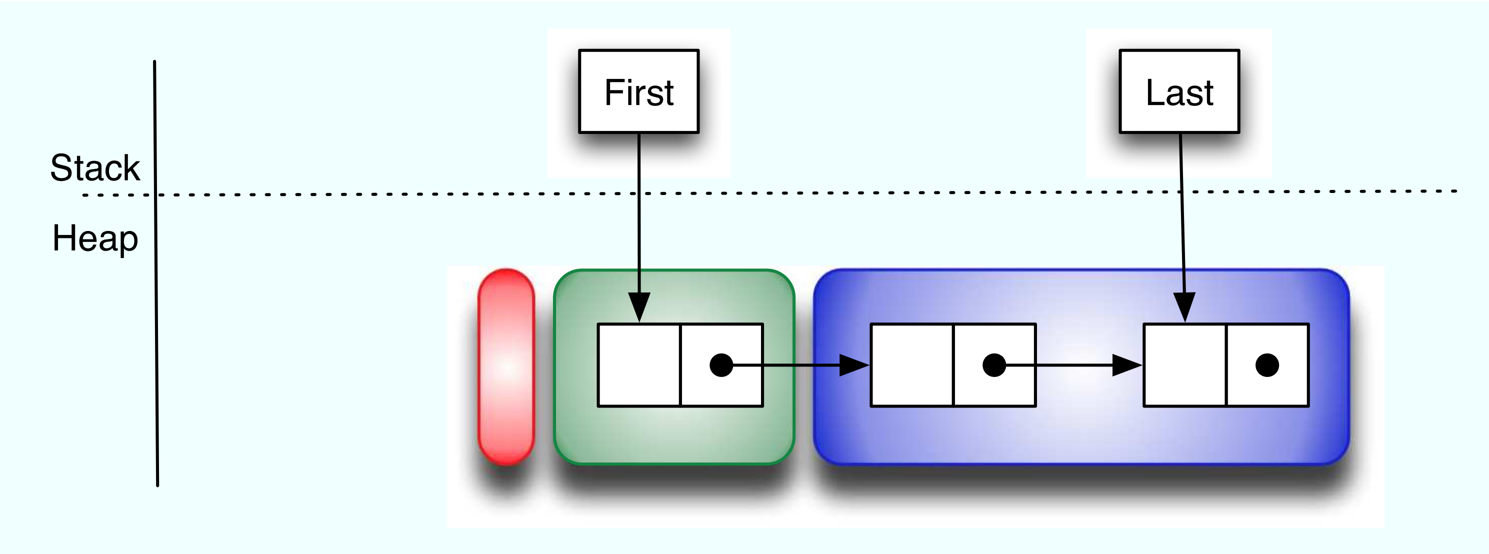
Example: Queue



semaphore number= 0;

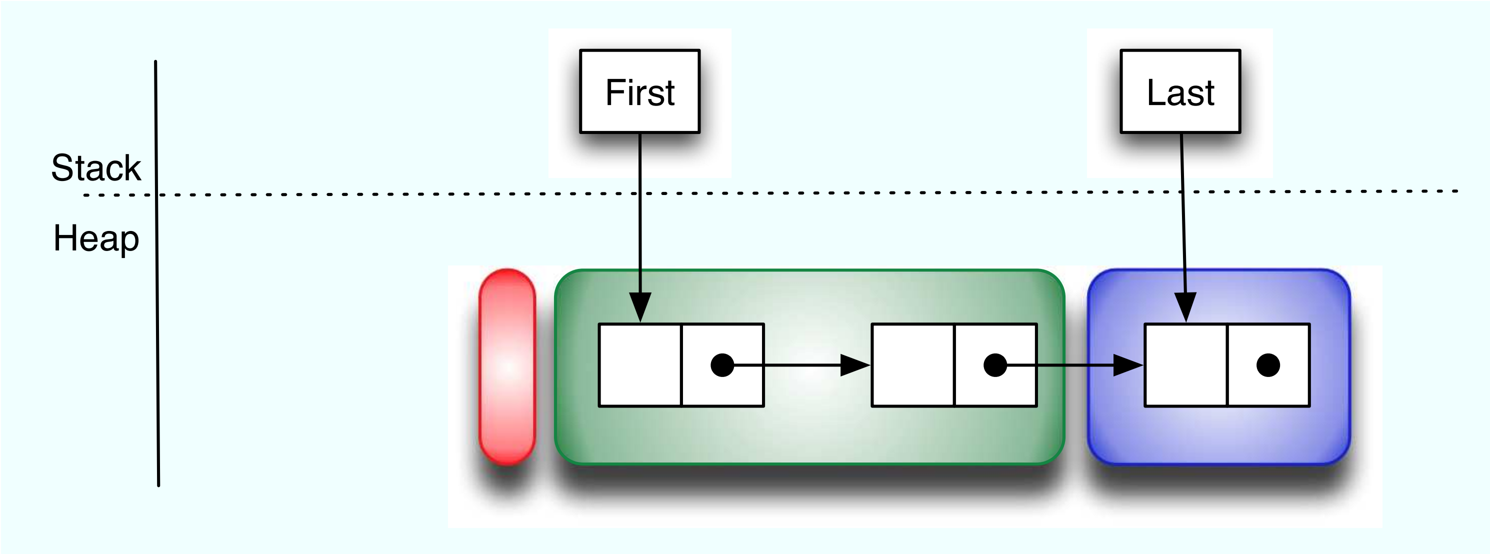
P (number);	x = cons(*,*)
t = first	[last+1] = x;
first = [first+1]	last = [last+1]
dispose(t);	V(number);

Example: Queue



	semaphore number= 0;	
P (number);	‖	x = cons(*,*)
t = first	‖	[last+1] = x;
first = [first+1]	‖	last = [last+1]
dispose(t);	‖	V(number);

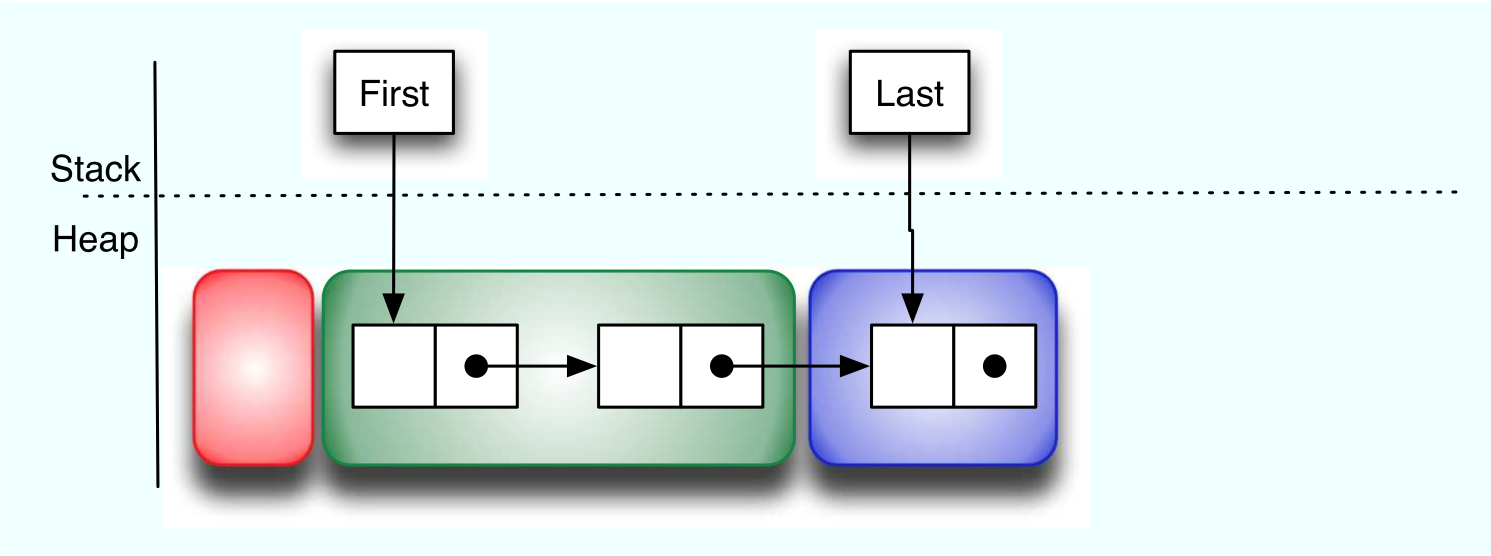
Example: Queue



semaphore number= 0;

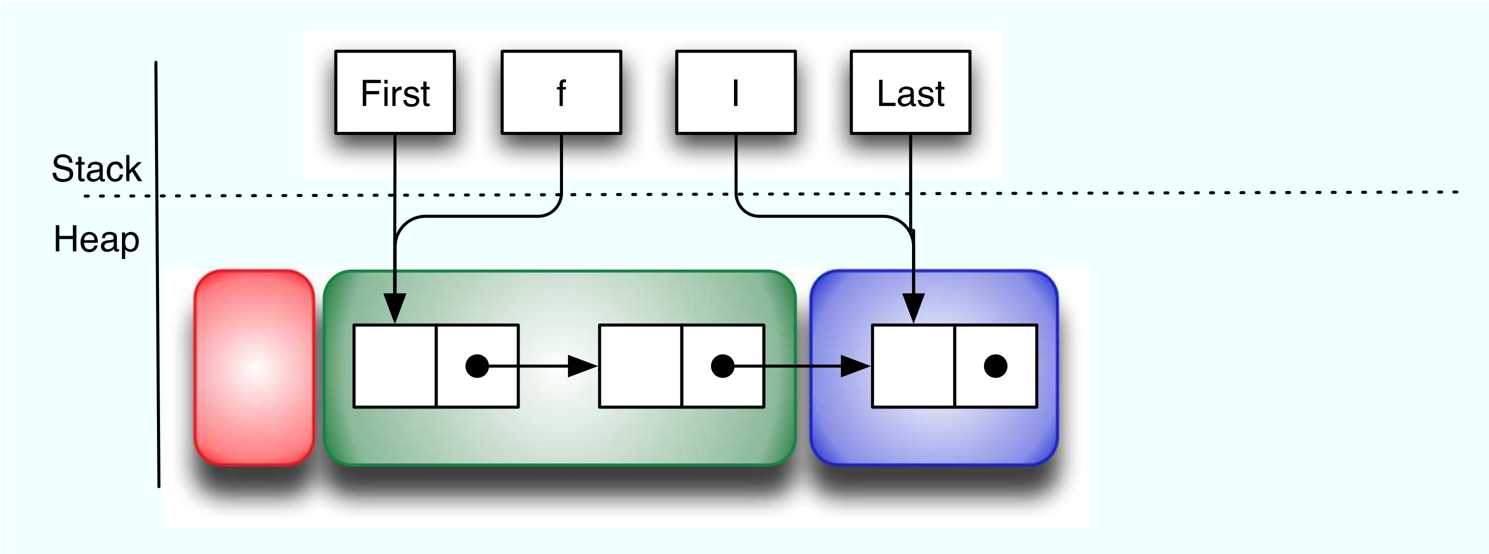
P (number);	x = cons(*,*)
t = first	[last+1] = x;
first = [first+1]	last = [last+1]
dispose(t);	V(number);

What is the green box?



P(n) = with n when n > 0 do f = [first+1]; n--
V(n) = with n when do l = last; n++

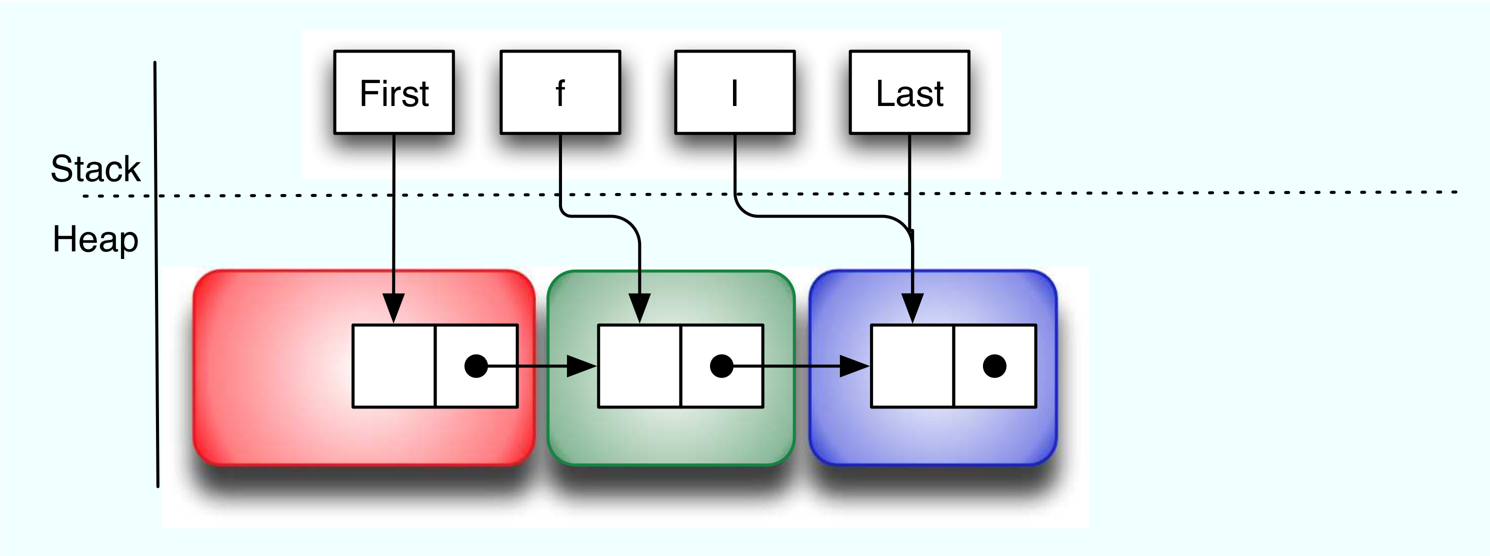
What is the green box?



P(n) = with n when n > 0 do f = [first+1]; n--

V(n) = with n when do l = last; n++

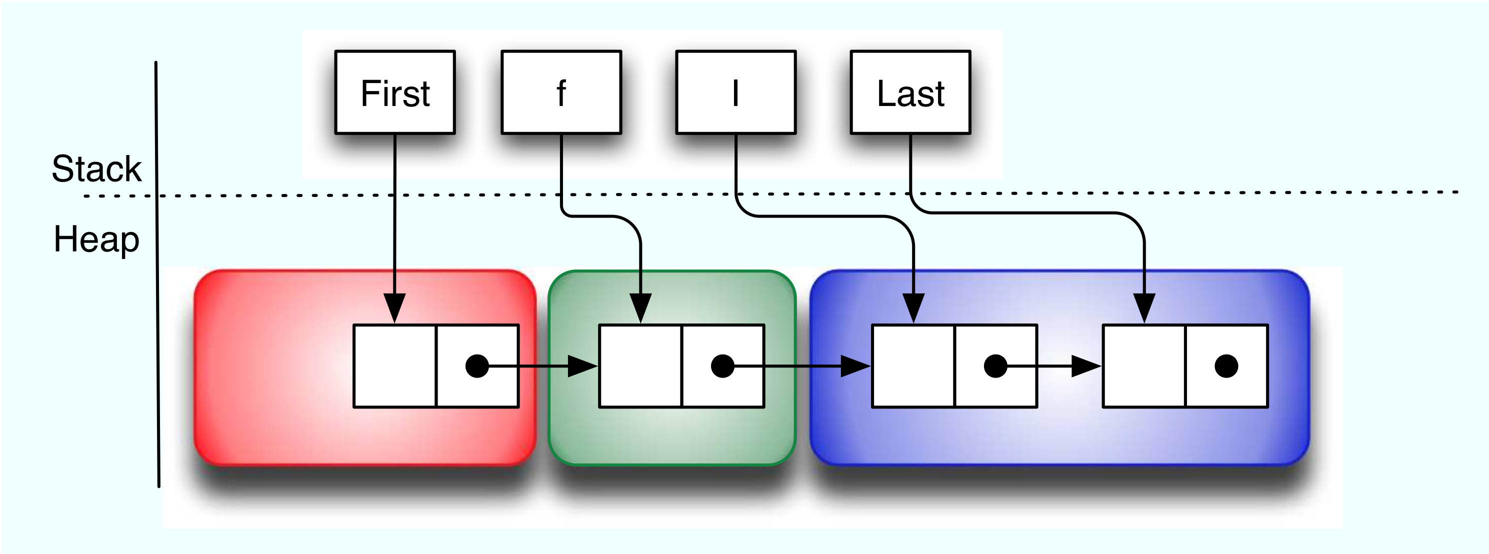
What is the green box?



P(n) = with n when n > 0 do f = [first+1]; n--

V(n) = with n when do l = last; n++

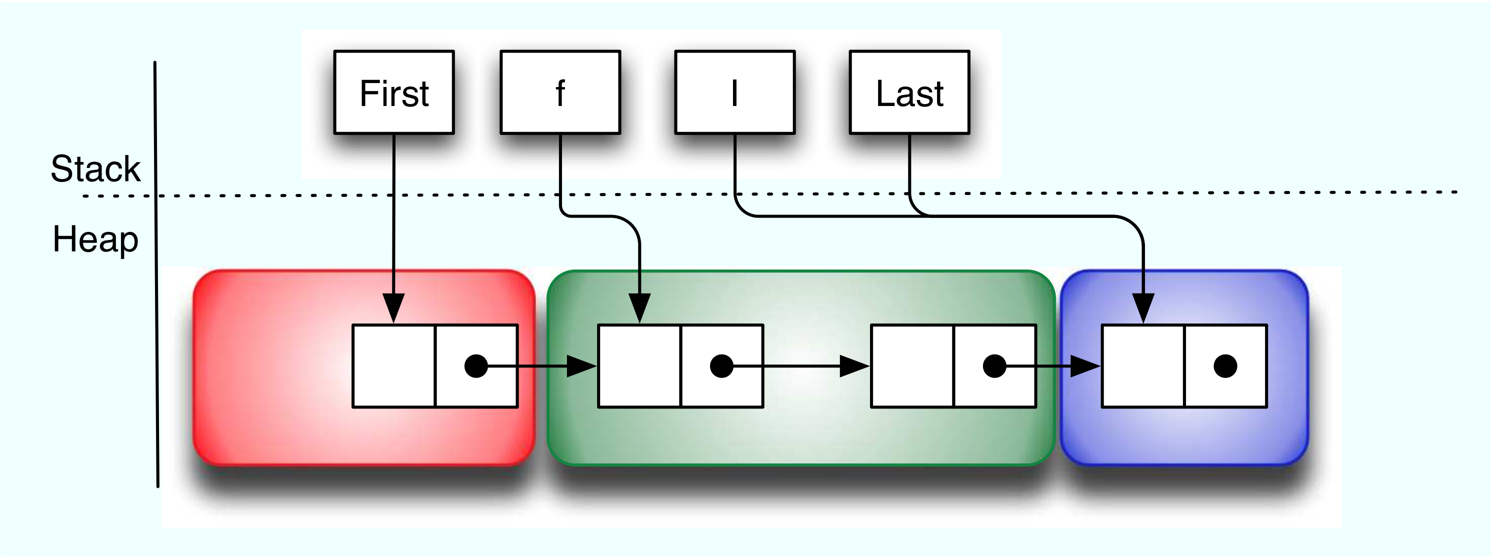
What is the green box?



P(n) = with n when n > 0 do f = [first+1]; n--

V(n) = with n when do l = last; n++

What is the green box?



P(n) = with n when n > 0 do f = [first+1]; n--
V(n) = with n when do l = last; n++

$\left\{f = first \wedge emp\right\}$	$\left\{l = last \wedge last \mapsto \_,\_\right\}$
P (number);	x = cons(*,*)
$\left\{first \mapsto \_, f\right\}$	$\left\{l = last \wedge last \mapsto \_,\_ * x \mapsto \_,\_\right\}$
t = first	[last+1] = x;
$\left\{first \mapsto \_, f \wedge t = first\right\}$	$\left\{l = last \wedge last \mapsto \_, x * x \mapsto \_,\_\right\}$
first = [first+1]	last = [last+1]
$\left\{t \mapsto \_, f \wedge f = first\right\}$	$\left\{l \mapsto \_, last * last \mapsto \_,\_\right\}$
dispose(t);	V(number);
$\left\{f = first\right\}$	$\left\{last \mapsto \_,\_ \wedge l = last\right\}$

Invariant = ls(f number l)

where $ls\ x\ n\ z = \ (x = z \wedge n = 0 \wedge emp)$

$$\vee\ (x \neq z \wedge n > 0 \wedge \exists y.\ x \mapsto \_, y * ls\ y\ (n-1)\ z)$$

Conclusions

- Can handle modularity
  - static: single instance of hidden datastructure
  - dynamic: abstract datatypes and classes
- Can handle concurrency
  - No interference flooding

What's next?

- Tool support – Berdine, Calcagno, O'Hearn
- Inference – Distefano, Berdine, Cook, O'Hearn
- Code Pointers – Thielecke
- Racy Concurrency – Parkinson, Bornat
- Java – Parkinson
- Dynamic Allocation of Semaphores – ?

References

The references for this part of the course can all be found on:
`http://www.dcs.qmw.ac.uk/~ohearn/localreasoning.html`