Verification of the Heap Manager of an Operating System using Separation Logic

Nicolas Marti
University of Tokyo

Reynald Affeldt
AIST

Akinori Yonezawa
University of Tokyo/AIST

Motivation: Formal Verification of Operating Systems

Correctness of operating systems is the basis of computer security

- E.g., Memory Isolation for multi-users systems

Our test-bed: Topsy [Ruf, ANTA 2003]

- Embedded OS for autonomous network devices
- Simple and small, yet contains most general-purpose OS features

Today's Presentation

Formal verification of the memory allocation mechanism of Topsy

Main aspects of our approach:

- Source code verification
- In the Coq proof assistant [INRIA, 1984-2005]
- Using separation logic [Reynolds, O'Hearn, 1999-2001]

Main contributions:

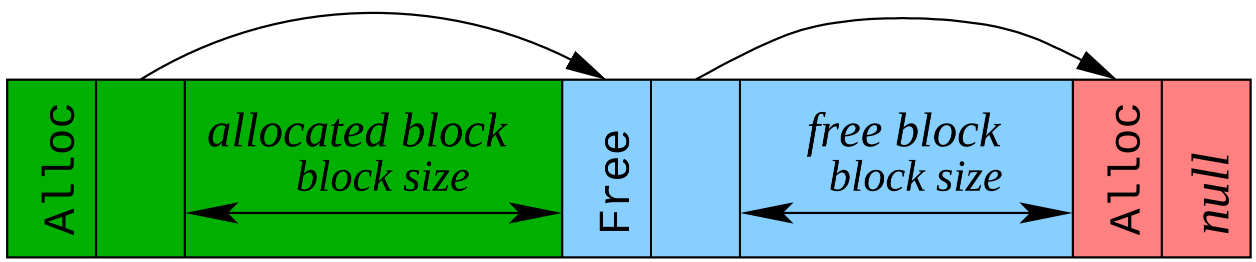
- Certification of a reusable memory allocation library
- A reusable Coq implementation of separation logic
  (available online)

Outline

1. Heap Manager Overview
2. The Heap-List Data Structure
3. Formal Verification
4. Implementation in Coq
5. Related Work

Heap Manager: The underlying Data Structure

A linked list, hereafter Heap-List:



- Covers a fixed area of contiguous memory
  - No lost space
- Composed of variable-size memory blocks
  - Two-fields header: (status, address of the next block)

Heap Manager: Interface

Basically three functions:

- Initialization:

  `Error hmInit(Address addr)`

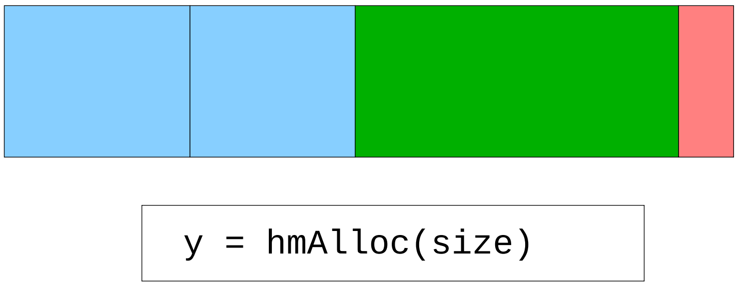
- Allocation:

  `Error hmAlloc(Address* addressPtr, unsigned long int size)`

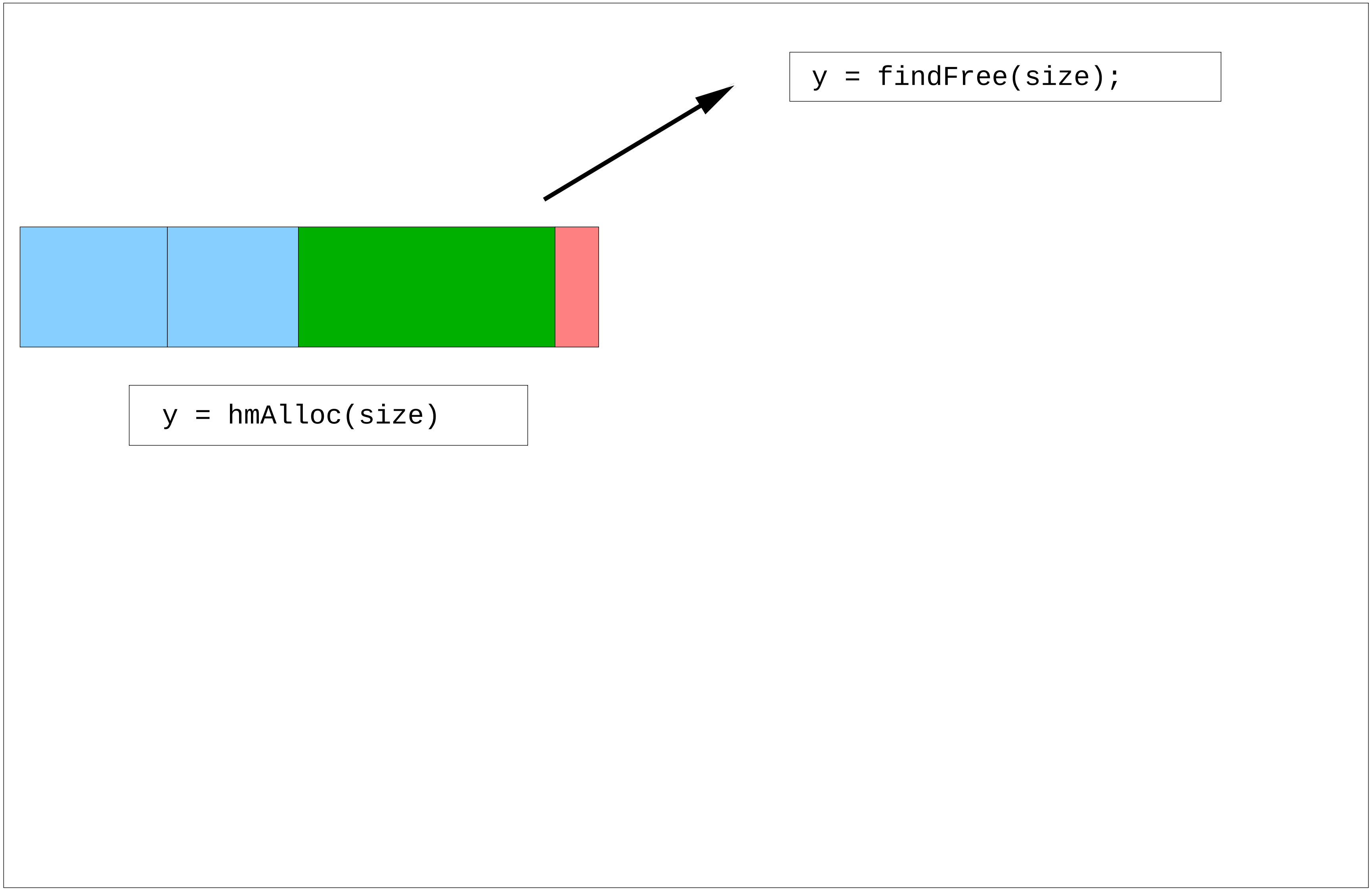
- Deallocation:

  `Error hmFree(Address address)`

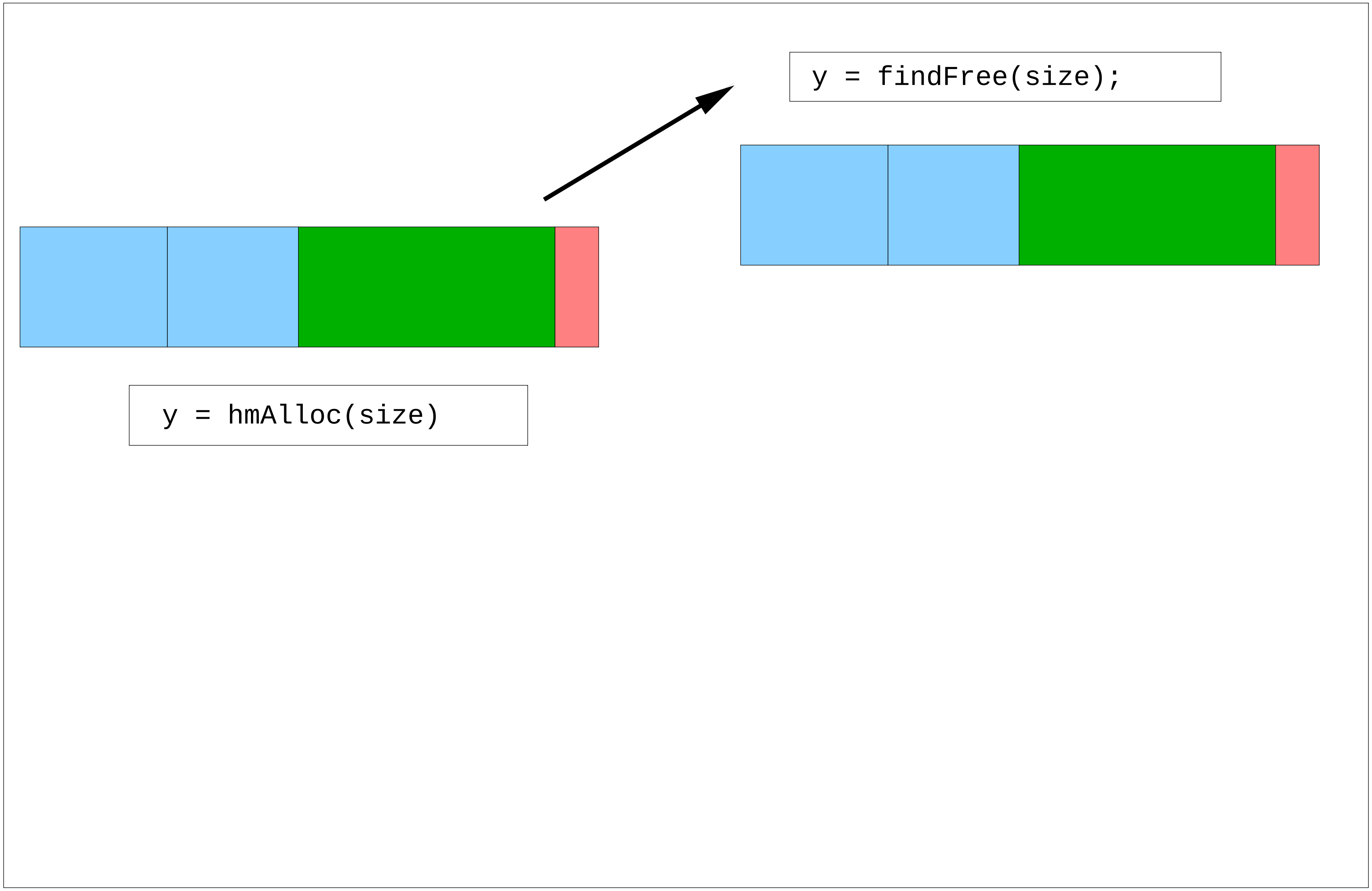
Implementation of the Allocation Function (Overview)



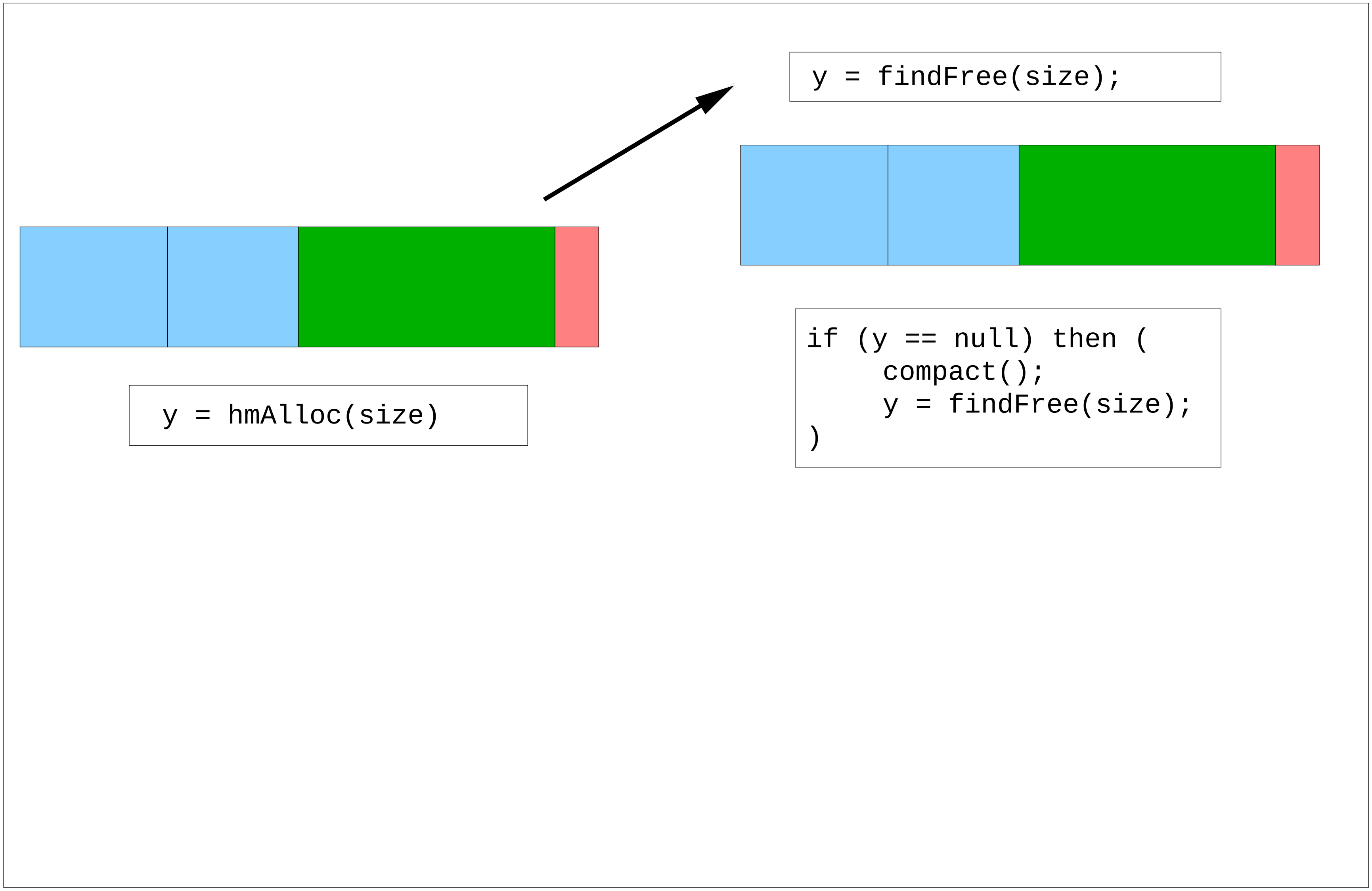
Implementation of the Allocation Function (Overview)



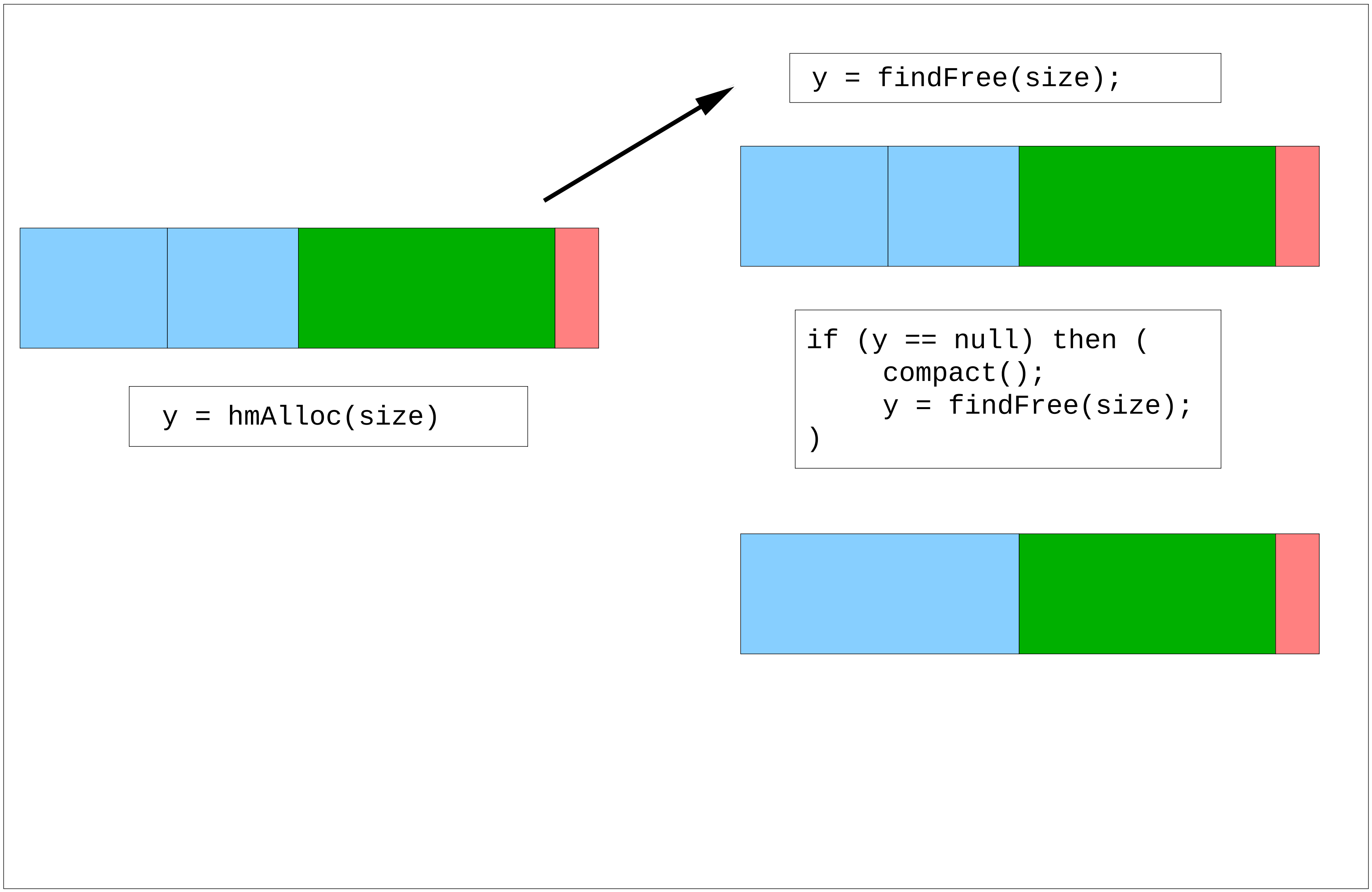
Implementation of the Allocation Function (Overview)



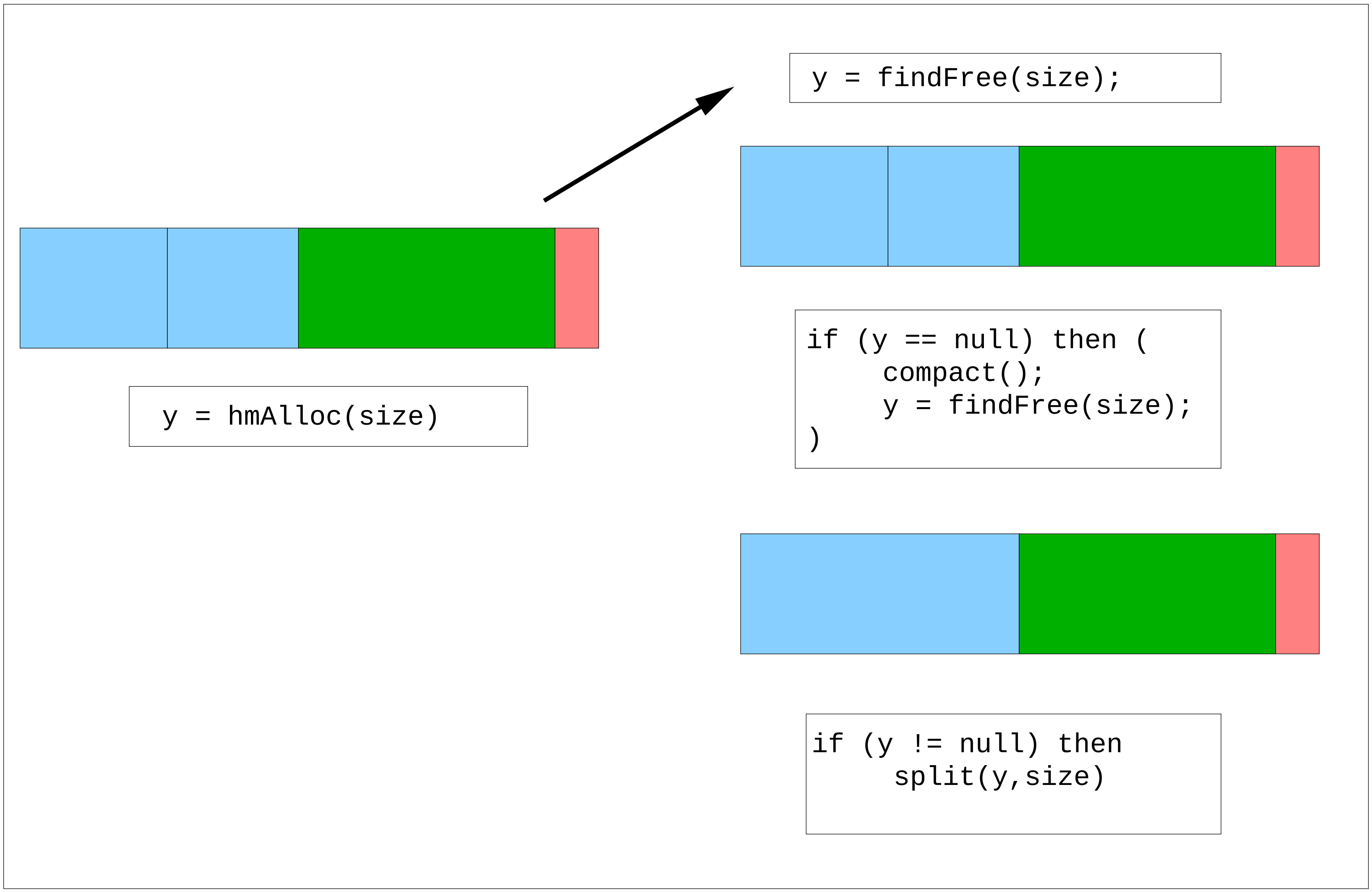
Implementation of the Allocation Function (Overview)



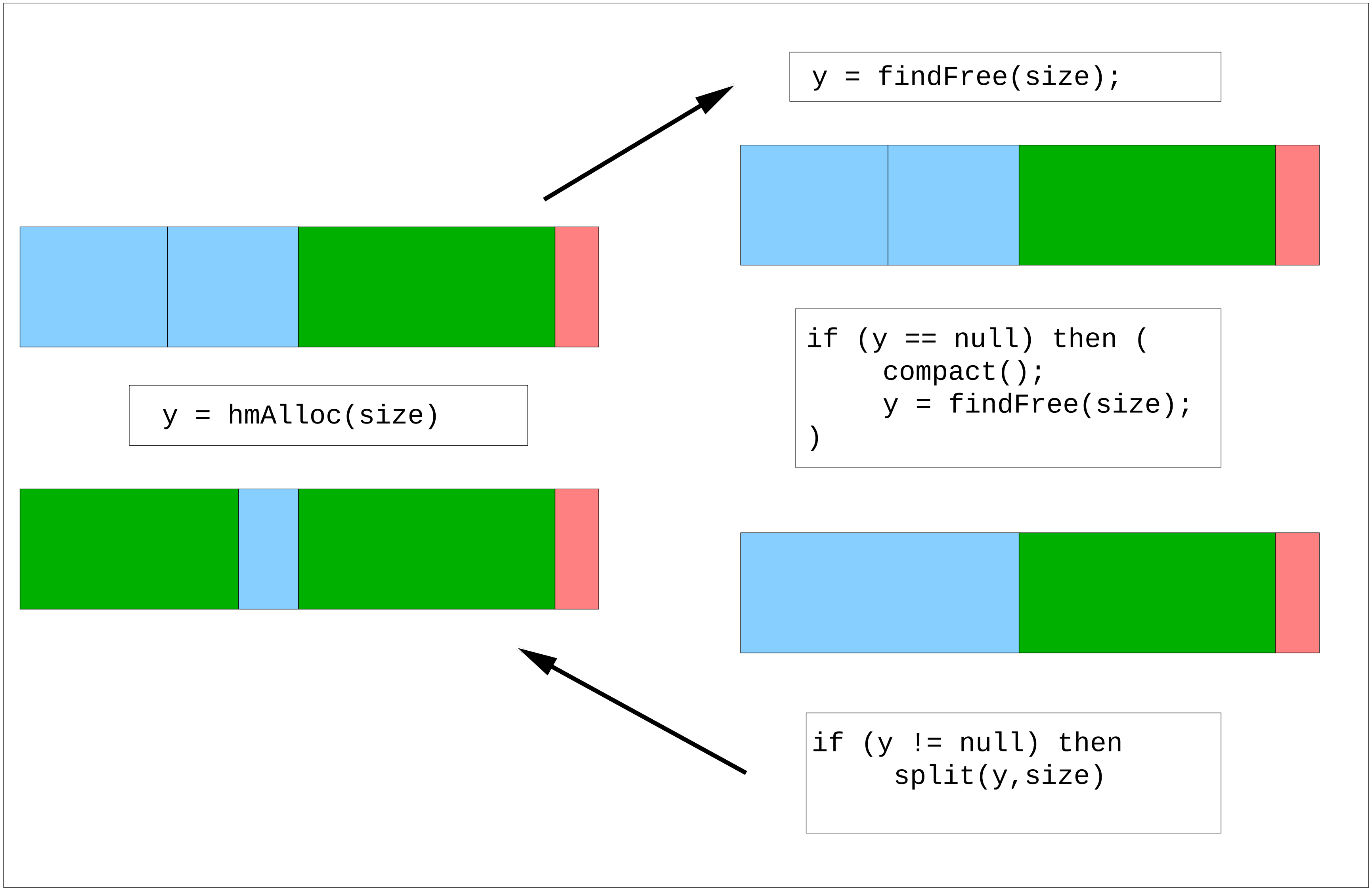
Implementation of the Allocation Function (Overview)



Implementation of the Allocation Function (Overview)

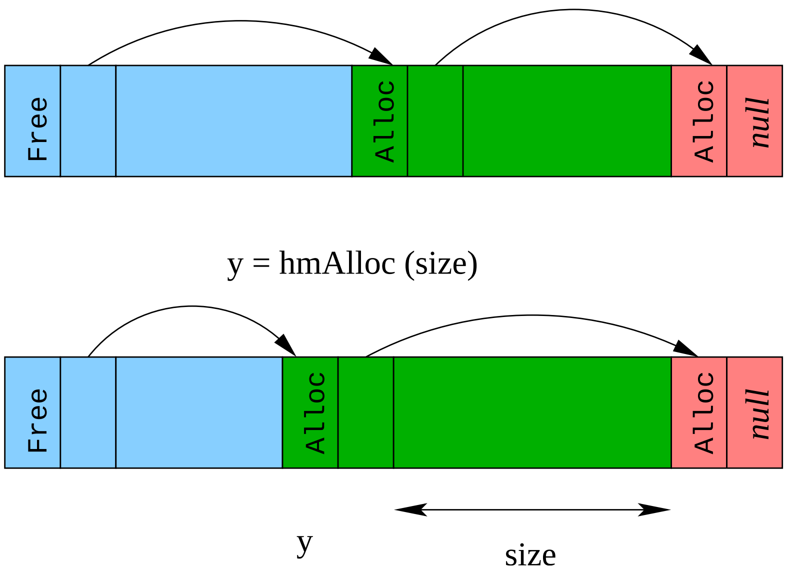


Implementation of the Allocation Function (Overview)

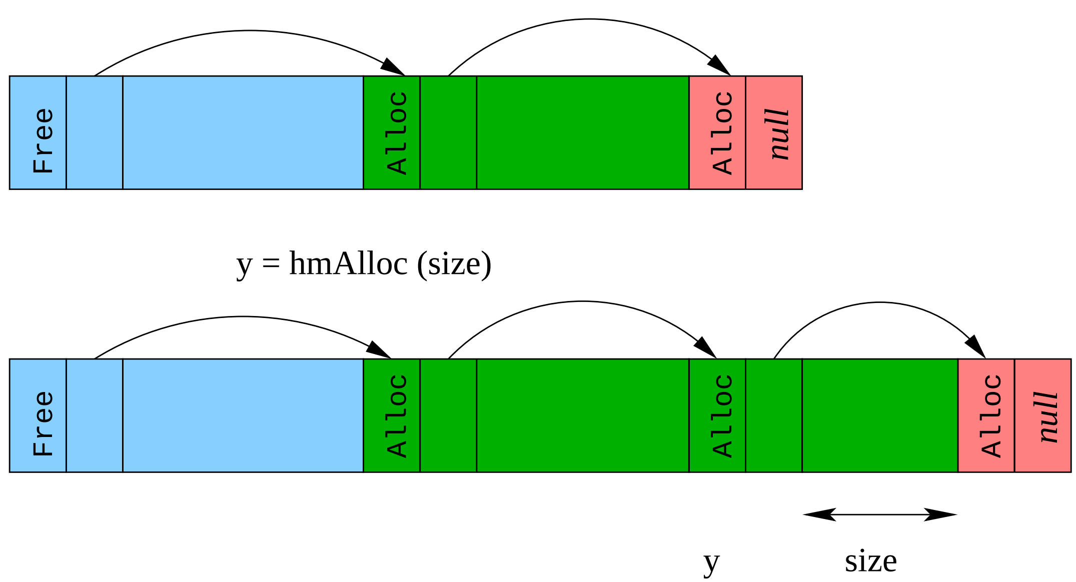


What Kind of Bugs Can We Fear?

Overwriting of allocated blocks



Heap overrun



⇒ Separation logic is a convenient way to specify such cases

Outline

1. Heap Manager Overview
2. The Heap-List Data Structure
3. Formal Verification
4. Implementation in Coq
5. Related Work

Formal Specification: Heap-List

Inductive definition (three constructors):

$$\text{Heap-list } (l : \mathit{list\ of}\ (loc \times nat \times status))\ (x : loc)\ (y : loc) \stackrel{def}{=}$$

$$l = nil \wedge (x \mapsto \texttt{Alloc}, null) \wedge y = 0 \vee$$

$$l = nil \wedge x = y \geq 0 \wedge \epsilon \vee$$

$$\exists size. \exists status. \exists l'.$$

$$(status = \texttt{Alloc} \vee status = \texttt{Free}) \wedge$$

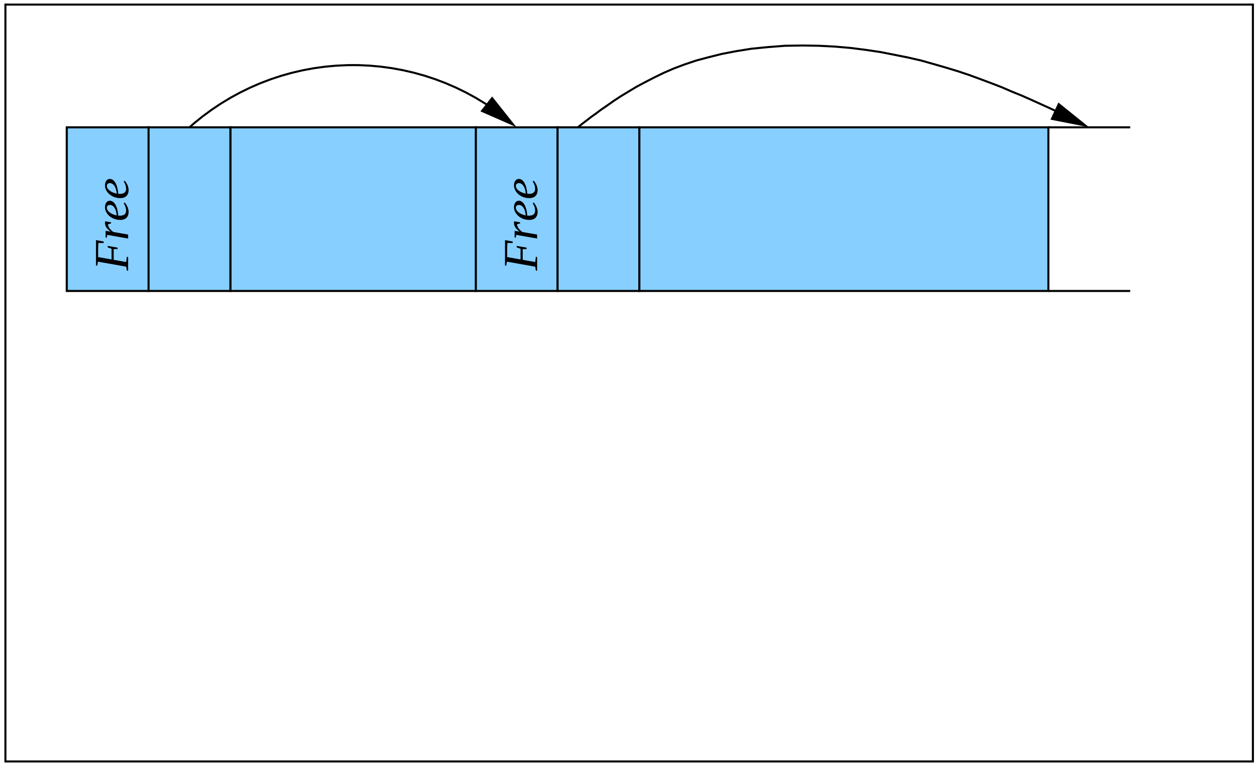
$$l = (x, size, status) :: l' \wedge x > 0 \wedge$$

$$(x \mapsto status, x+2+size) *$$

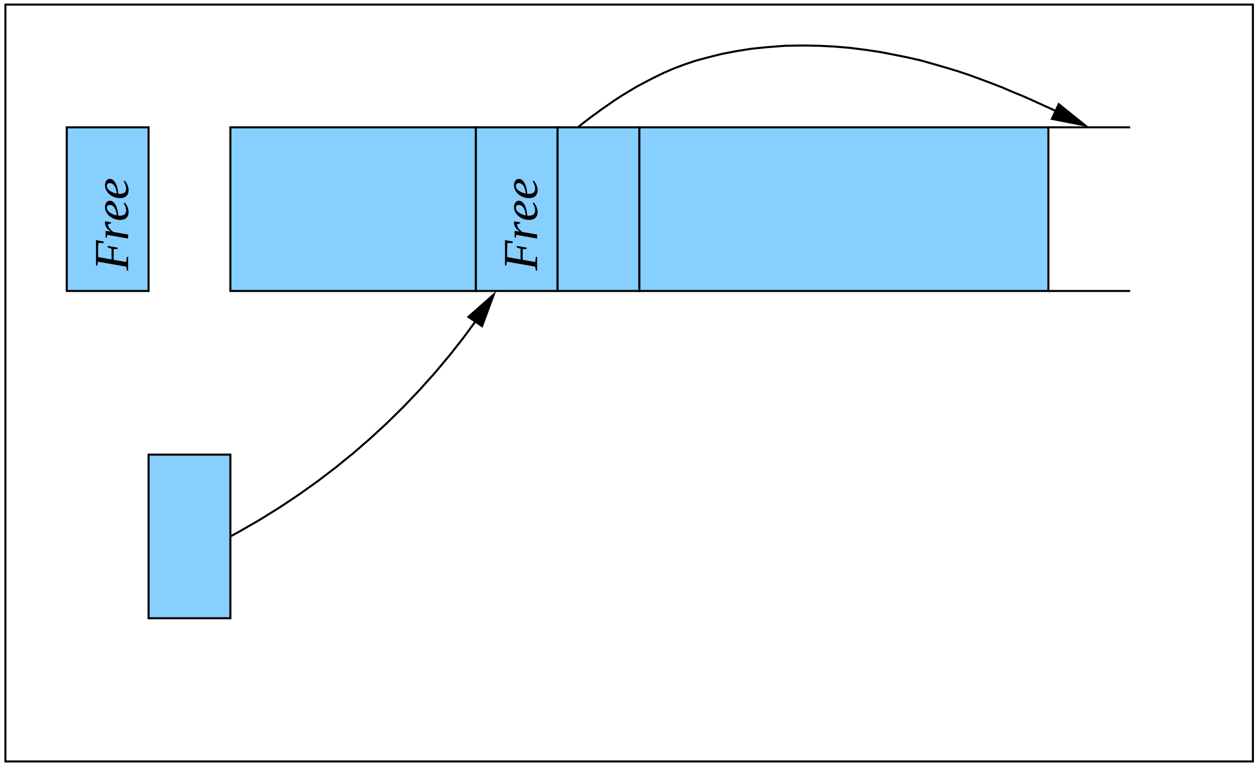
$$\text{Array } (x+2)\ size * \text{Heap-list } l'\ (x+2+size)\ y$$

N.B.: A Heap-List is valid when $y = 0$

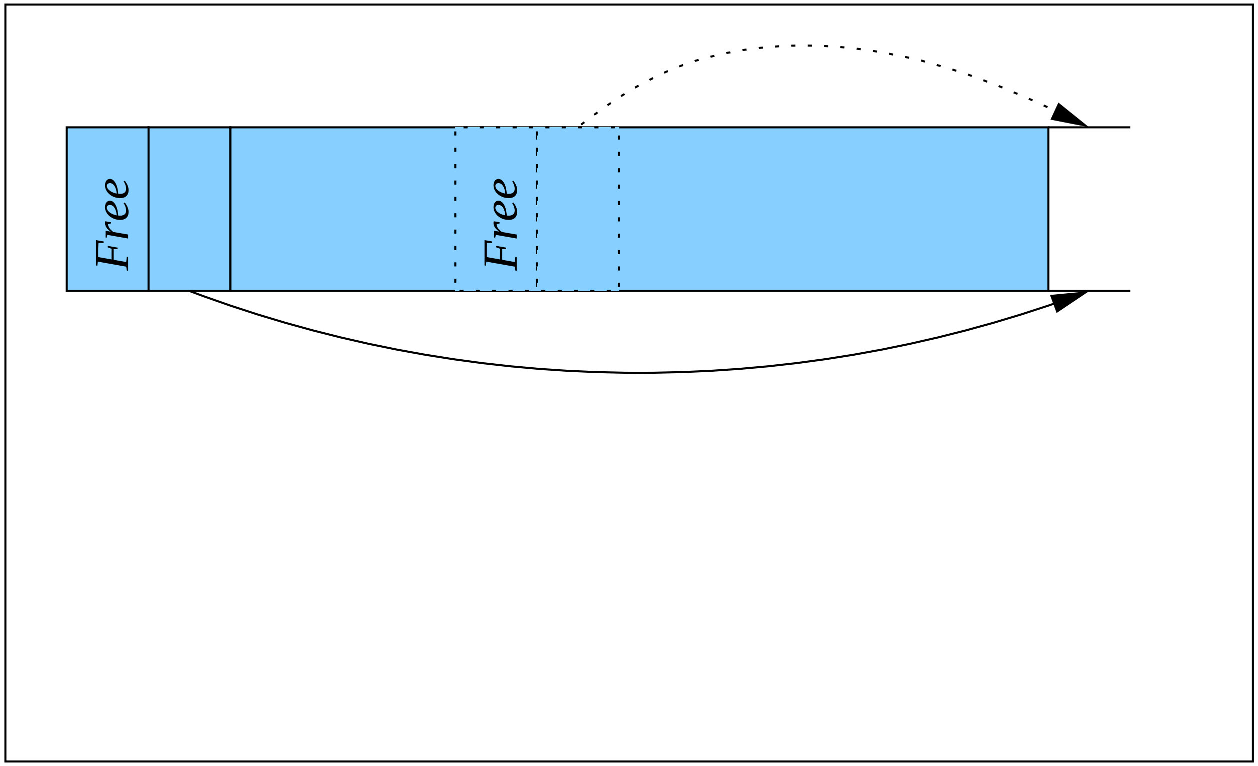
An Important Block Manipulation: Compaction



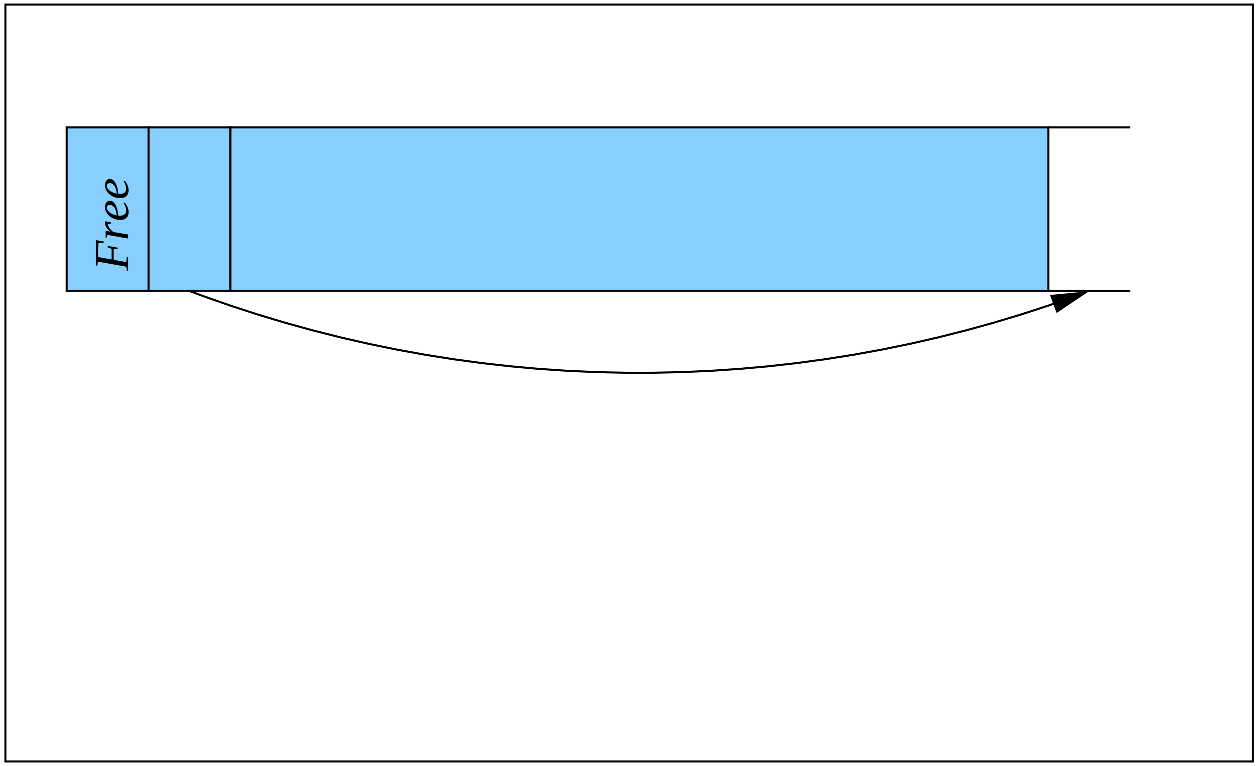
An Important Block Manipulation: Compaction



An Important Block Manipulation: Compaction



An Important Block Manipulation: Compaction



A Lemma that captures Block Compaction

COMPACTION:

Before Heap-list $(l_1 +\!\!+ ((x, size_1, \texttt{Free}) :: (y, size_2, \texttt{Free}) :: nil) +\!\!+ l_2)\ x_0\ 0 \rightarrow$

Destructive $(x+1 \mapsto y) *$

update $((x+1 \mapsto address\_nextblock) -\!\!*$

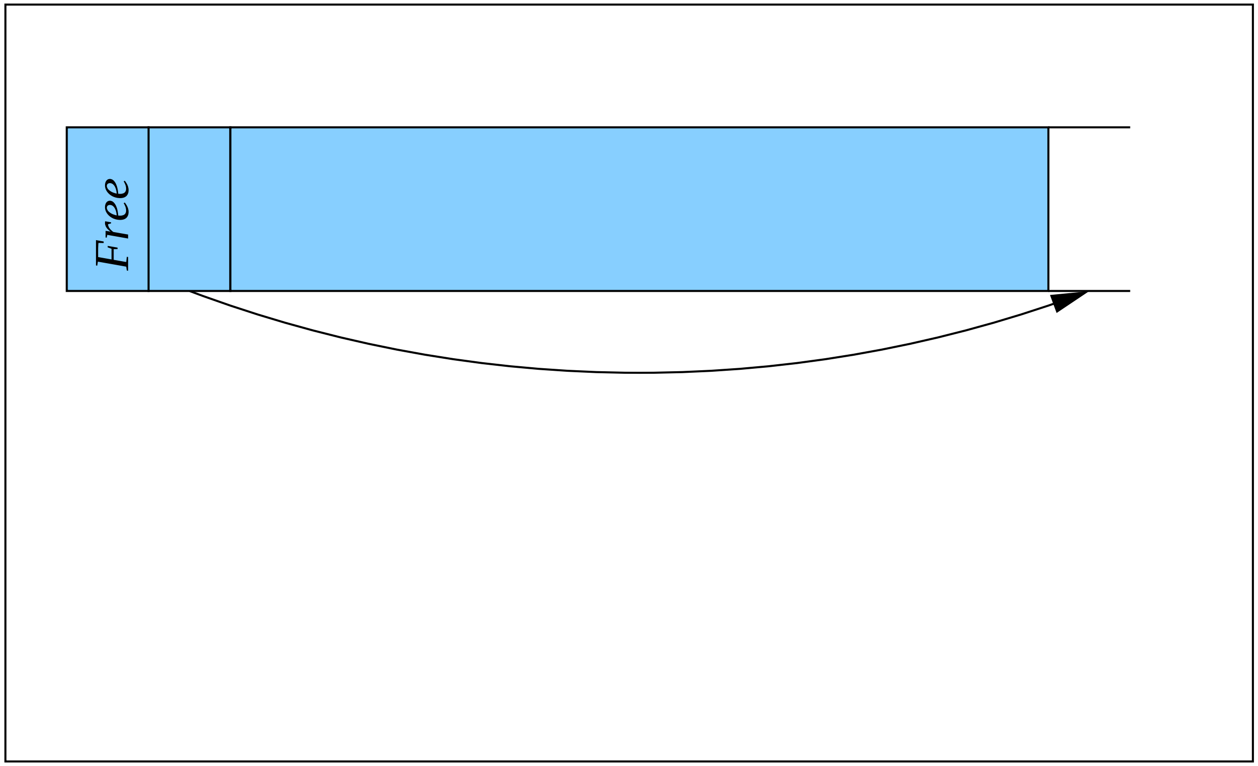
After Heap-list $(l_1 +\!\!+ ((x, size_1 + 2 + size_2, \texttt{Free}) :: nil) +\!\!+ l_2)\ x_0\ 0)$

where

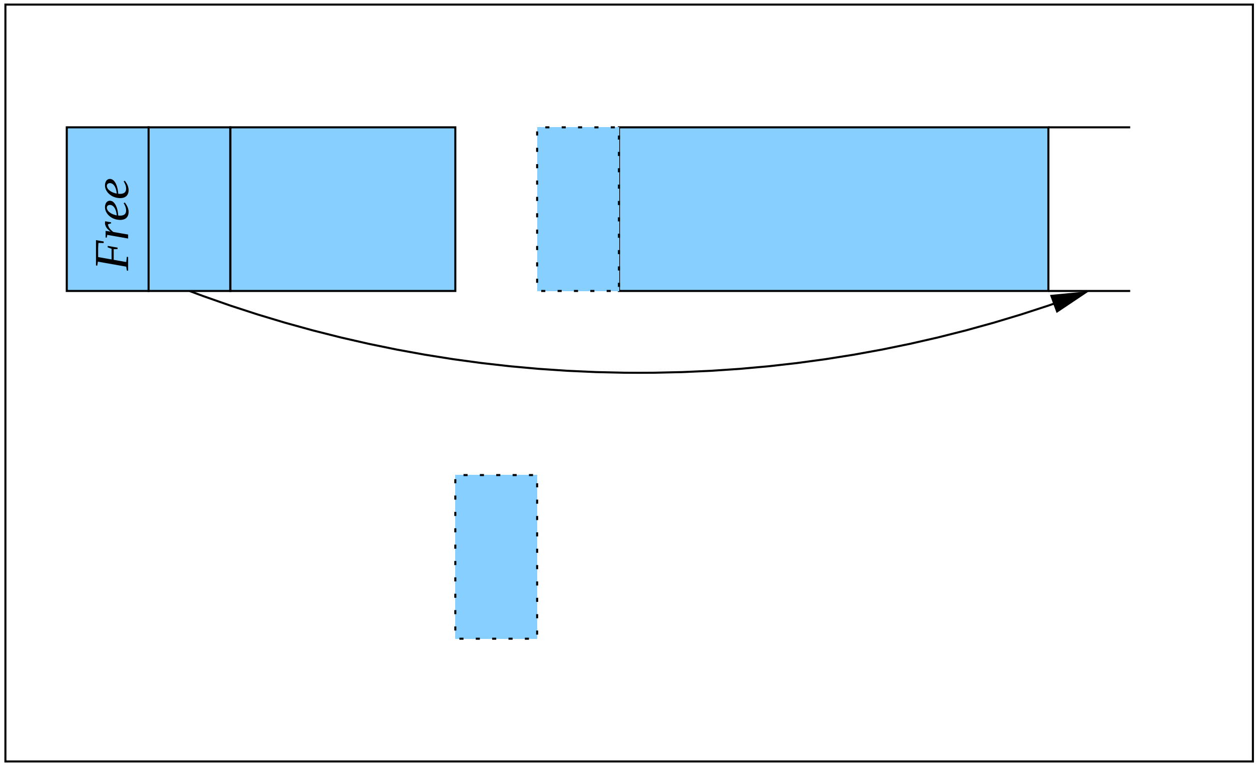
$y = x + 2 + size_1$

$address\_nextblock = x + size_1 + 4 + size_2$

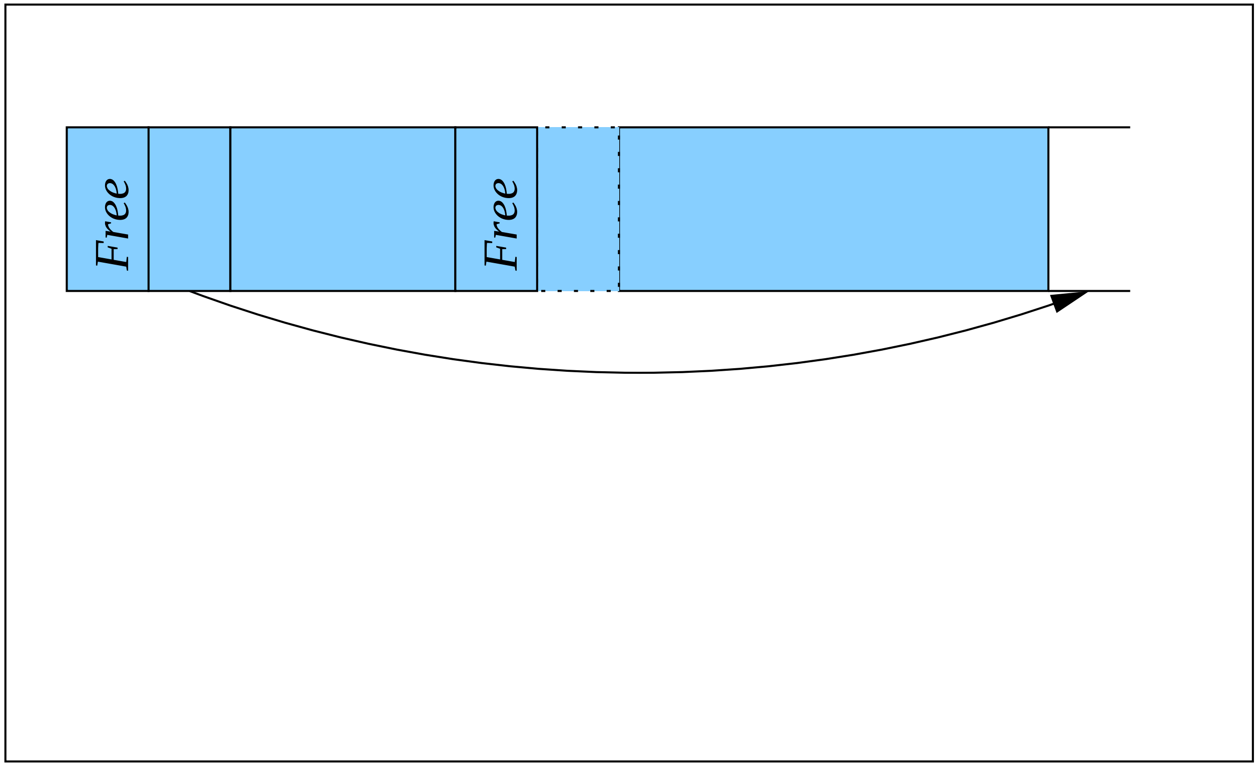
An Important Block Manipulation: Splitting



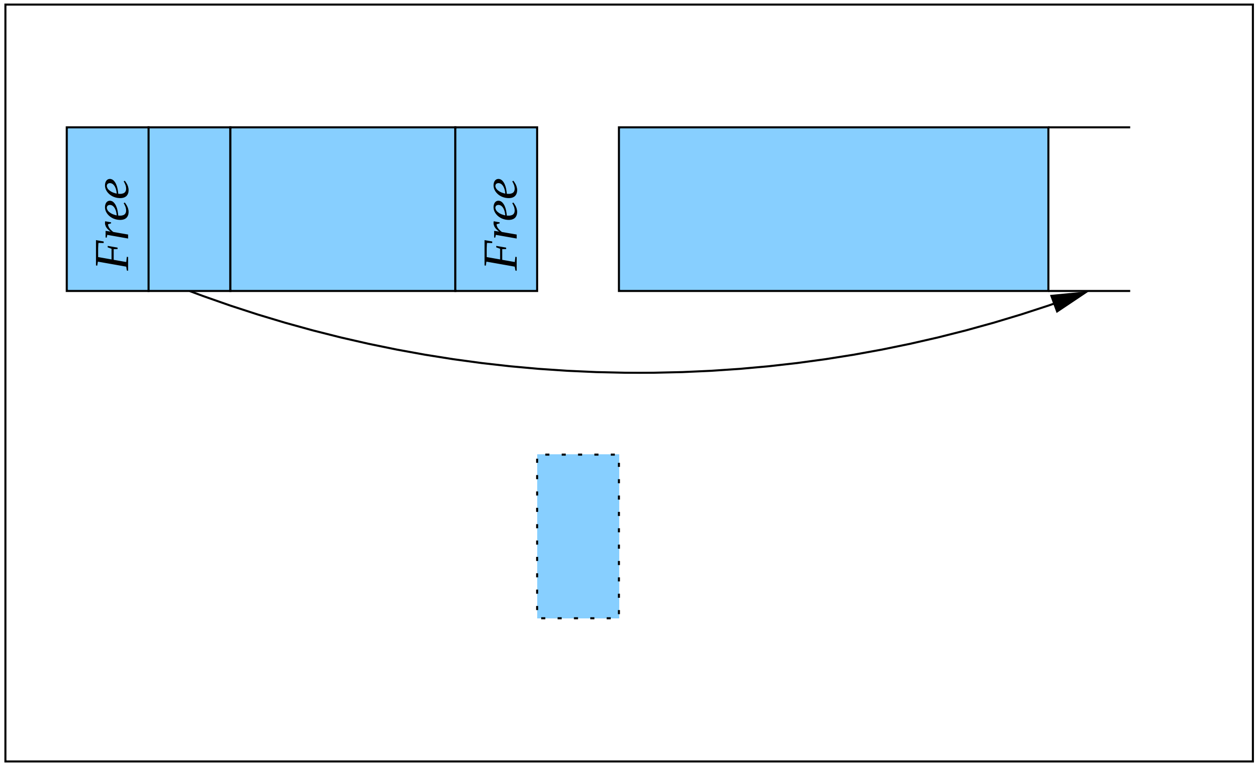
An Important Block Manipulation: Splitting



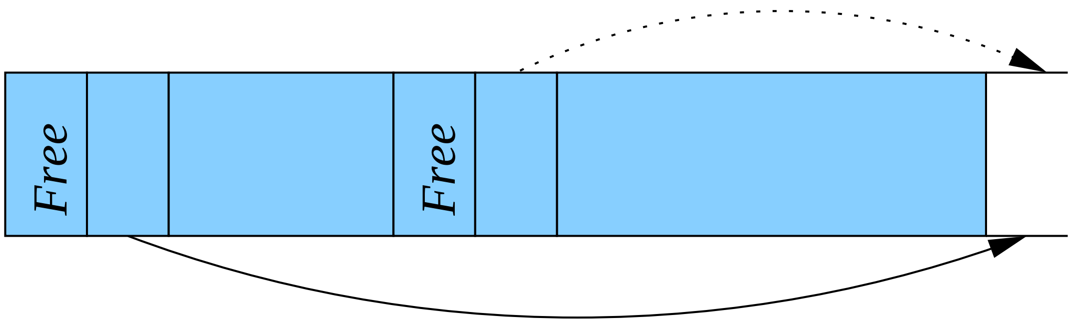
An Important Block Manipulation: Splitting



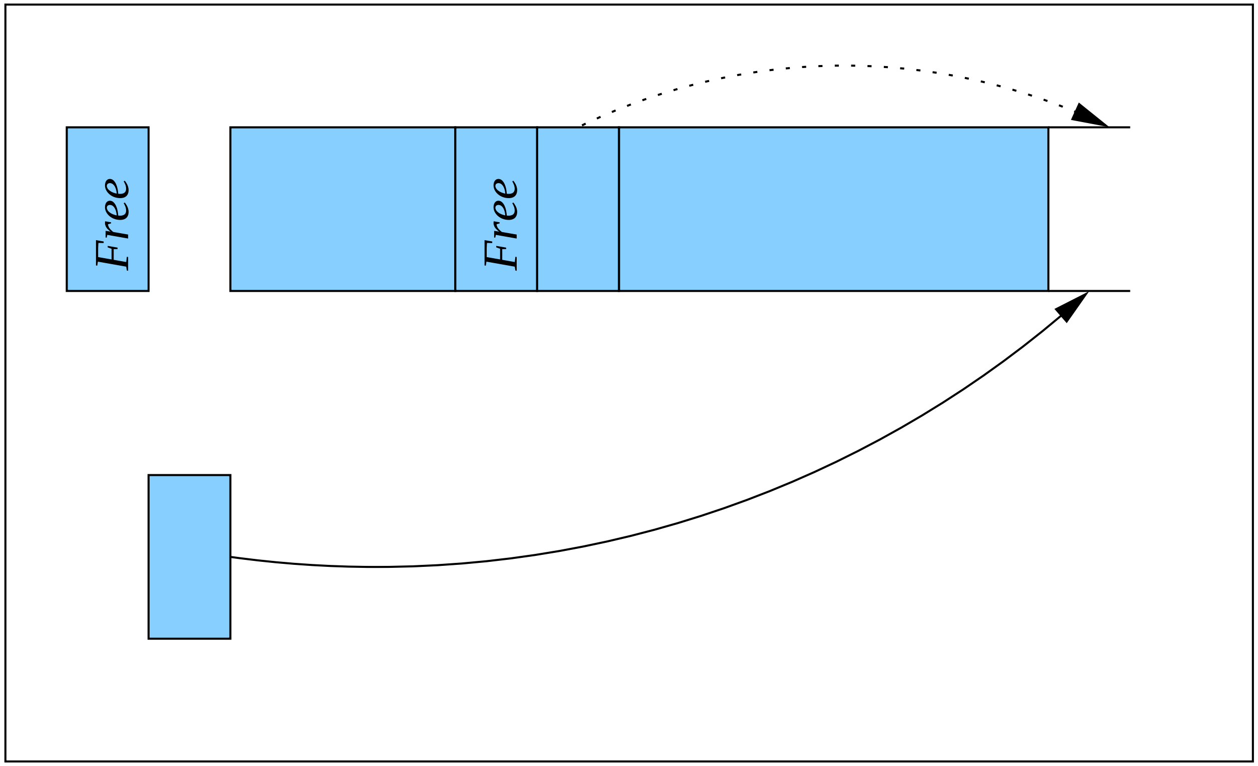
An Important Block Manipulation: Splitting



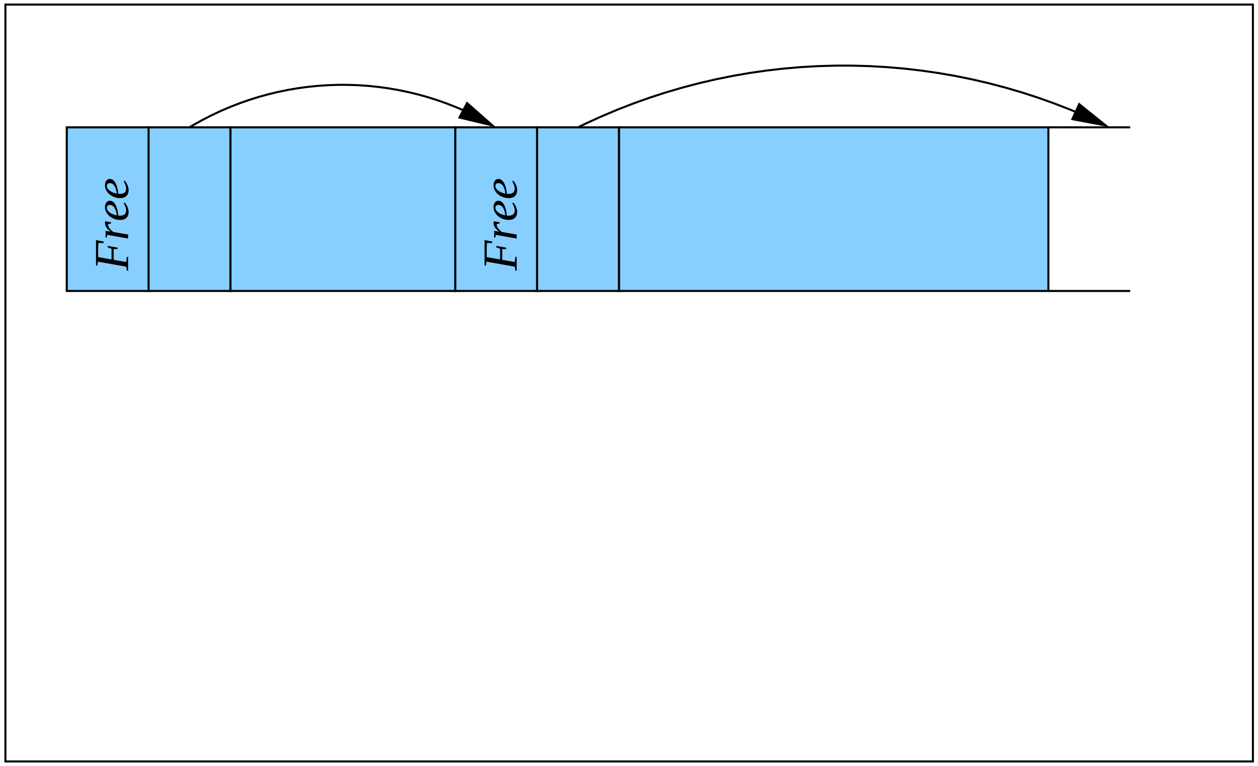
An Important Block Manipulation: Splitting



An Important Block Manipulation: Splitting



An Important Block Manipulation: Splitting



Outline

1. Heap Manager Overview
2. The Heap-List Data Structure
3. Formal Verification (excerpt)
4. Implementation in Coq
5. Related Work

Formal Specification: `hmAlloc`

$$\left\{ \begin{array}{c} \exists l.\textsf{Heap-list}\ l\ \texttt{hmStart}\ 0 \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \\ \textsf{Init-array}\ (x{+}2)\ (list_x) \end{array} \right\}$$

`hmAlloc result size entry cptr fnd stts nptr sz`

$$\left\{ \begin{array}{c} \exists l.\textsf{Heap-list}\ l\ \texttt{hmStart}\ 0 \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \\ \textsf{Init-array}\ (x{+}2)\ (list_x) \wedge \\ \left(\begin{array}{c} \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Alloc}) \in l \wedge \\ \texttt{entry} = y \wedge \texttt{result} = \texttt{entry}+2 \wedge x \neq y \\ \vee \\ \texttt{result} = 0 \end{array} \right) \end{array} \right\}$$

Formal Verification (1/2): List Traversal and Compaction

$$\left\{ \begin{array}{c} \exists l.\textsf{Heap-list}\ l\ \texttt{hmStart}\ 0 \wedge (x, size_x, \texttt{Alloc}) \in l \wedge \\ \textsf{Init-array}\ (x{+}2)\ (list_x) \end{array} \right\}$$

```
y = findFree(size);
if (y == null) then (
   compact();
   y = findFree(size);
)
```

$$\left\{ \begin{array}{c} \exists l.\textsf{Heap-list}\ l\ \wedge\ (x, size_x, \texttt{Alloc}) \in l \wedge \textsf{Init-array}\ (x{+}2)\ (list_x) \wedge \\ \left(\begin{array}{c} \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Free}) \in l\ \wedge\ x \neq y \\ \vee \\ y{=}0 \end{array} \right) \end{array} \right\}$$

Formal Verification (2/2): Splitting

$$\left\{ \begin{array}{c} \exists l.\text{Heap-list}\ l\ \wedge\ (x, size_x, \texttt{Alloc}) \in l \wedge \text{Init-array}\ (x{+}2)\ (list_x) \wedge \\ \left(\begin{array}{c} \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Free}) \in l\ \wedge\ x \neq y \\ \vee \\ y{=}0 \end{array} \right) \end{array} \right\}$$

$$\texttt{if (}y\texttt{ != null) then}$$

$$\texttt{split(}y\texttt{, size);}$$

$$\left\{ \begin{array}{c} \exists l.\text{Heap-list}\ l\ \wedge\ (x, size_x, \texttt{Alloc}) \in l \wedge \text{Init-array}\ (x{+}2)\ (list_x) \wedge \\ \left(\begin{array}{c} \exists y.\exists size_y.size_y \geq \texttt{size} \wedge (y, size_y, \texttt{Alloc}) \in l\ \wedge\ x \neq y \\ \vee \\ y{=}0 \end{array} \right) \end{array} \right\}$$

Formal Verification: Results

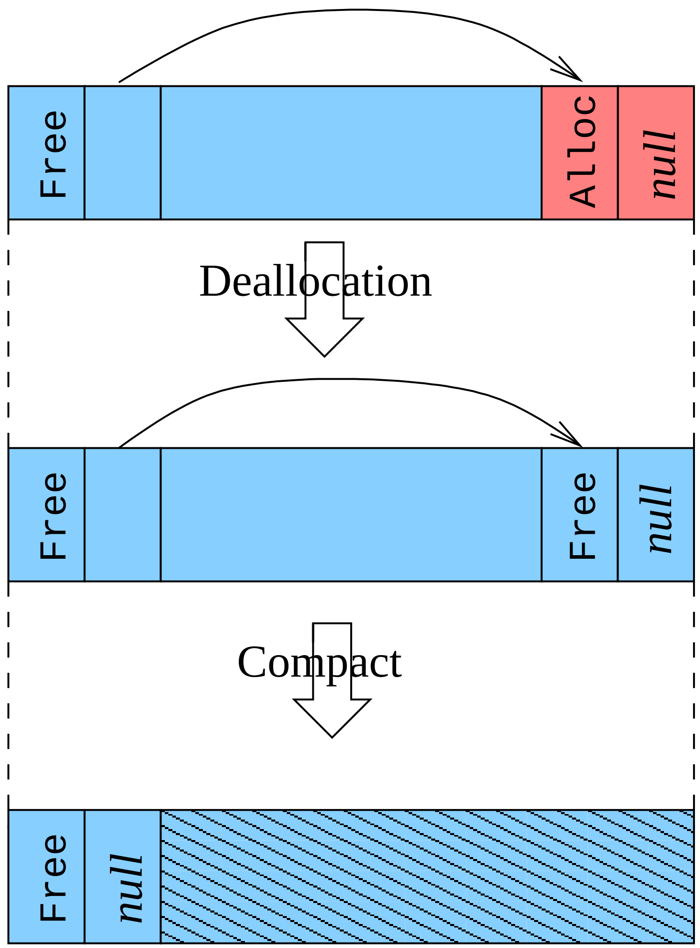
We certified the heap manager:

- Source code can be reused
- The Hoare triples can be reused

We did find bugs:

- Initialization wrote the ending header outside of the heap (corrected in recent versions)
- Allocation of empty blocks succeeded
- Deallocation: A much more subtle bug. . .

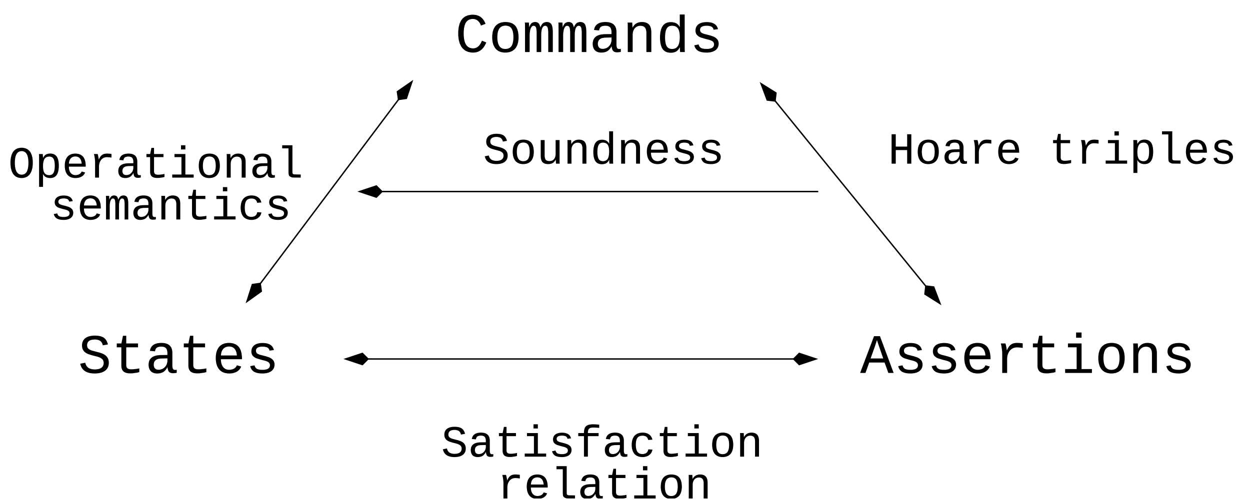
Deallocation Function Bug



Outline

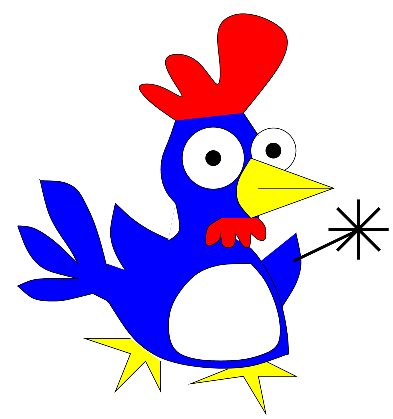
1. Heap Manager Overview
2. The Heap-List Data Structure
3. Formal Verification
4. Implementation in Coq
5. Related Work

Implementation in Coq: Core



Additional features:

- Data structures (arrays, lists,. . .)
- Lemmas (frame rule, monotony,. . .)
- Weakest precondition generator (proved sound)
- Tactics (equality/disjointness of heaps,. . .)
- A cute mascot



Outline

1. Heap Manager Overview
2. The Heap-List Data Structure
3. Formal Verification
4. Implementation in Coq
5. Related Work

Comparison With Related Work

Certification of dynamic storage allocation in an assembly language [Yu et al., ESOP 2003]:

- In constrat, our experiment deals with existing C code
- Topsy heap manager is self-content

Implementation of separation logic in the Isabelle proof assistant [Weber, CSL 2004]:

- Our library is larger (use-cases)
- Technical difference: use of abstract data type / partial functions

Future work

Extension of the framework:

- With a decision procedure for loop-free programs (in progress)
- Interface with Smallfoot [Berdine et al., APLAS 2005]?
- To assembly language (in progress)
- Pursue proof of memory isolation for Topsy
  (pending problem: how to handle concurrency)

Thank you

A Possible Improvement of the Formal Specification

$$\left\{\begin{array}{c} \exists l.\mathsf{Heap\text{-}list}\ l\ \mathtt{hmStart}\ 0 \wedge (x, size_x, \mathtt{Alloc}) \in l\ \wedge \\ \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x)\wedge \\ \mathsf{ContiguousFree}\ l\ z\ \mathtt{size} \end{array}\right\}$$

`(hmAlloc result size entry cptr fnd stts nptr sz)`

$$\left\{\begin{array}{c} \exists l.\mathsf{Heap\text{-}list}\ l\ \mathtt{hmStart}\ 0 \wedge (x, size_x, \mathtt{Alloc}) \in l\ \wedge \\ \mathsf{Init\text{-}array}\ (x{+}2)\ (list_x)\ \wedge \\ \left(\begin{array}{c} \exists y.\exists size_y.size_y \geq \mathtt{size} \wedge (y, size_y, \mathtt{Alloc}) \in l\ \wedge \\ \mathtt{entry} = y \wedge \mathtt{result} = \mathtt{entry}{+}2 \wedge x \neq y \end{array}\right) \end{array}\right\}$$

$$\begin{array}{rl} \mathsf{ContiguousFree}\ l\ z\ \mathtt{size} \stackrel{def}{=} & \exists l'.l' \subseteq l\ \wedge \\ & l' = (z, sz, \mathtt{Free}) :: (z{+}sz, sz_1, \mathtt{Free}) :: \cdots \wedge \sum_i sz_i \geq \mathtt{size} \end{array}$$

findFree always succeeds because compact maintains:

$$\texttt{cptr} \neq null \wedge \texttt{cptr} \leq z \rightarrow (\textsf{ContiguousFree}\ l\ z\ \texttt{size})$$

$$\wedge$$

$$z \leq \texttt{cptr} < z{+}size \rightarrow (z, \texttt{cptr}{-}z, \texttt{Free}) \in l \wedge (\textsf{ContiguousFree}\ l\ \texttt{cptr}\ (\texttt{size}{-}(\texttt{cptr}{-}z)))$$

$$\wedge$$

$$\texttt{cptr}{=}null \vee \texttt{cptr} \geq z{+}\texttt{size} \rightarrow (z, \texttt{size}, \texttt{Free}) \in l$$