
Proving Correctness of a Garbage Collector via Local Reasoning

Lars Birkedal [birkedal@itu.dk], Noah Torp-Smith [noah@itu.dk]

The IT University of Copenhagen

Joint work with John C. Reynolds, Carnegie Mellon University



Motivation



Motivation

- Copying garbage collectors widely used, for example in implementations of functional languages.



Motivation

- Copying garbage collectors widely used, for example in implementations of functional languages.
- A “non-toy” example of the power of Separation Logic. Yang’s proof of the Schoor-Waite algorithm is another such.



Motivation

- Copying garbage collectors widely used, for example in implementations of functional languages.
- A “non-toy” example of the power of Separation Logic. Yang’s proof of the Schoor-Waite algorithm is another such.
- Proof Carrying Code theory assumes an underlying memory allocator, but doesn’t treat it further.



Separation Logic - Intro



Separation Logic - Intro

Storage Model

States $s : Var \rightarrow Vals = \mathbb{Z} \cup Loc$

Heaps $h : Loc \rightarrow Vals$



Separation Logic - Intro

Storage Model

States $s : Var \rightarrow Vals = \mathbb{Z} \cup Loc$

Heaps $h : Loc \rightarrow Vals$

Programming Language

Simple while-language extended with

Allocation, e.g., $x := \mathbf{cons}(e_1, e_2)$

Lookup, e.g., $y := x.1$

Manipulation, e.g., $x.2 := 42$

Disposal, e.g., $\mathbf{dispose}(x)$



Separation Logic - Intro (2)

Problem with *aliasing*: The command $x.1 := 17$ might affect the truth of $y \hookrightarrow (42, 84)$, if x and y are mapped to the same location.



Separation Logic - Intro (2)

Problem with *aliasing*: The command $x.1 := 17$ might affect the truth of $y \hookrightarrow (42, 84)$, if x and y are mapped to the same location.

As a consequence, the traditional *rule of constancy* from Hoare Logic

$$\frac{\{A\} C \{A'\}}{\{A \wedge B\} C \{A' \wedge B\}} \text{Modifies}(C) \cap FV(B) = \emptyset$$

is *not* valid.



Separation Logic - Intro (3)

We extend the assertion language with

		Meaning
Basic Predicates	emp	The heap is empty
	$e_1 \mapsto e_2$	e_1 points to e_2 (tight)
A Connective	$A * B$	A, B hold in <i>disjoint</i> subheaps



Separation Logic - Intro (3)

We extend the assertion language with

		Meaning
Basic Predicates	emp	The heap is empty
	$e_1 \mapsto e_2$	e_1 points to e_2 (tight)
A Connective	$A * B$	A, B hold in <i>disjoint</i> subheaps

We then have the *Frame Rule*:

$$\frac{\{A\} C \{A'\}}{\{A * B\} C \{A' * B\}} \text{Modifies}(C) \cap FV(B) = \emptyset$$

This allows us to do *local reasoning*, as we will see later.



Preliminaries - Prog. Languages

Setup: A *user language* and an *implementation language*, both standard while-languages, but with different memory interactions:

$$\begin{aligned} C_{user} & ::= \dots \mid x := \mathbf{cons}(e_1, e_2) \mid x.i := e \mid x := y.i \\ C_{impl} & ::= \dots \mid [e] := e \mid x := [e] \end{aligned}$$

User language: No pointer arithmetic, “implicit type system”:
 $\mathbf{Vals} = \mathbf{Ints} \uplus \mathbf{Ptr}$, heaps map pointers to pairs of values.

Implementation language: Pointer arithmetic, Heaps map *locations* (a subset of integers) to integers.



Preliminaries (2) - Interface

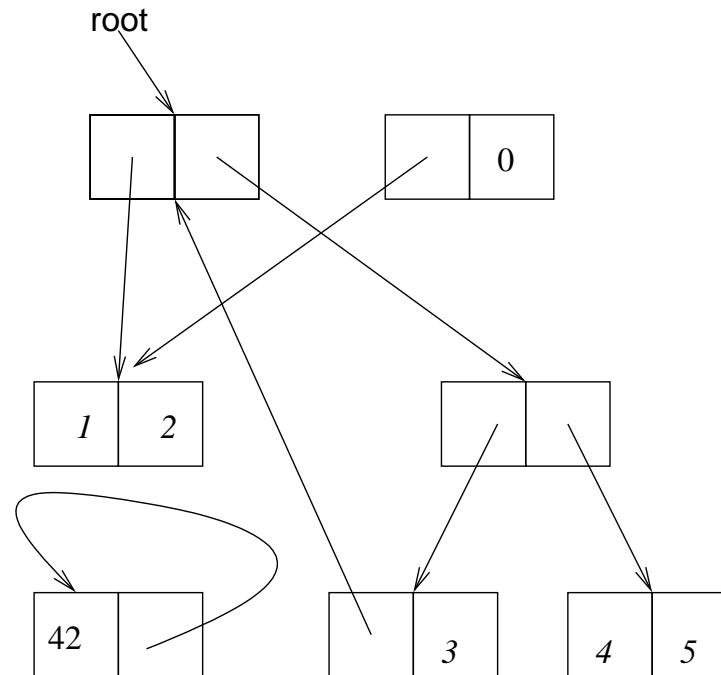
Interface (informal): The command `cons(e_1, e_2)` in the user language results in a call to `alloc` in implementation language.

```
alloc( $l, n_1, n_2$ ) {  
    if (any_space_left) {  
        allocate 2 heap cells;  
        store( $n_1, n_2$ );  
        return address;  
    }  
    else {  
        Garbage collect;  
        alloc( $l, n_1, n_2$ );  
    }  
}
```



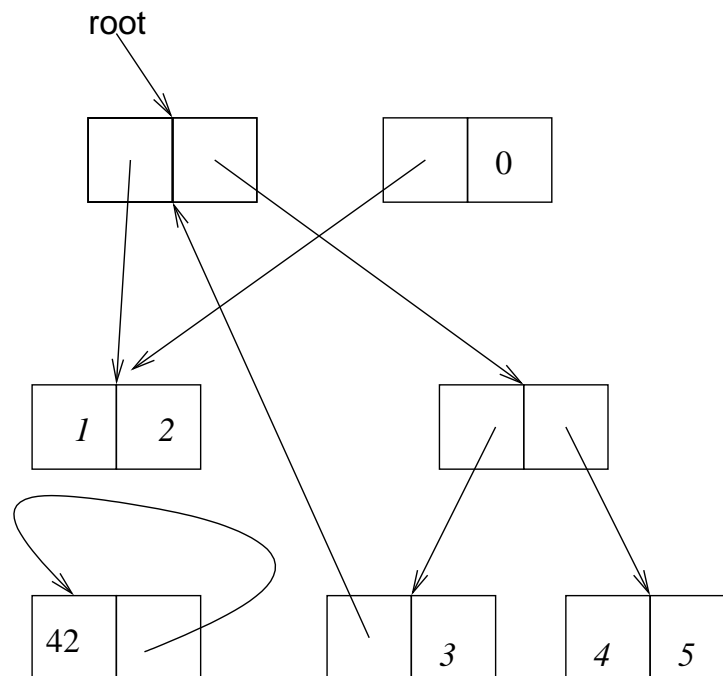
Preliminaries (3) - A Picture

So a picture might look like this:



Preliminaries (3) - A Picture

So a picture might look like this:



Note that some cells cannot be “reached” from the root cell.

These are ignored by copying collectors.



Preliminaries (4) - GC Req's

- A *weak heap isomorphism* $\varphi : (s', h') \cong (s, h)$ is a bijection $\varphi : \text{dom}(h') \cong \text{dom}(h)$ such that for all $p \in \text{dom}(h')$,

$$h(\varphi(p)) = \varphi^*(h'(p)),$$

where φ^* is the extension of φ to all integers (pointers and nonpointers) with the identity on nonpointers. If also $\varphi(s'(\text{root})) = s(\text{root})$, we call φ a *heap isomorphism*.



Preliminaries (4) - GC Req's

- A *weak heap isomorphism* $\varphi : (s', h') \cong (s, h)$ is a bijection $\varphi : \text{dom}(h') \cong \text{dom}(h)$ such that for all $p \in \text{dom}(h')$,

$$h(\varphi(p)) = \varphi^*(h'(p)),$$

where φ^* is the extension of φ to all integers (pointers and nonpointers) with the identity on nonpointers. If also $\varphi(s'(\text{root})) = s(\text{root})$, we call φ a *heap isomorphism*.

- (s, h) is a *garbage collected version* of (s', h') , if there is a heap isomorphism $\varphi : \text{prune}(s, h) \cong \text{prune}(s', h')$. We do not have to *remove* anything.



Preliminaries (4) - GC Req's

- A *weak heap isomorphism* $\varphi : (s', h') \cong (s, h)$ is a bijection $\varphi : \text{dom}(h') \cong \text{dom}(h)$ such that for all $p \in \text{dom}(h')$,

$$h(\varphi(p)) = \varphi^*(h'(p)),$$

where φ^* is the extension of φ to all integers (pointers and nonpointers) with the identity on nonpointers. If also $\varphi(s'(\text{root})) = s(\text{root})$, we call φ a *heap isomorphism*.

- (s, h) is a *garbage collected version* of (s', h') , if there is a heap isomorphism $\varphi : \text{prune}(s, h) \cong \text{prune}(s', h')$. We do not have to *remove* anything.
- So if GC is our garbage collector, and if $GC, s, h \rightsquigarrow s', h'$ the requirement is that (s', h') is a garbage collected version of (s, h) .



Cheney's Algorithm (1970)



Cheney's Algorithm (1970)

Assumes 2 contiguous “semi-spaces” of equal size,

OLD = [startOld, endOld[and NEW = [startNew, endNew[,

$s(\text{root}) \in \text{OLD}$. ALIVE = $\{p \mid p \text{ is reachable}\}$. Copies ALIVE to NEW in a “structure preserving way” and resumes allocation there.



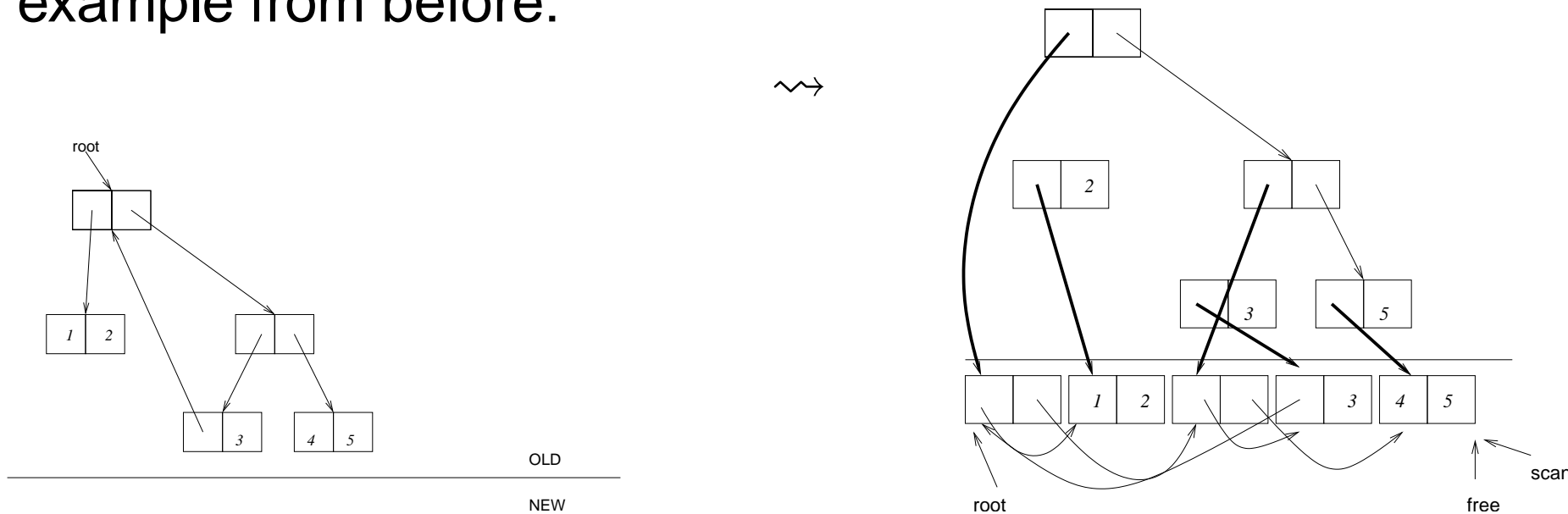
Cheney's Algorithm (1970)

Assumes 2 contiguous “semi-spaces” of equal size,

OLD = [startOld, endOld[and NEW = [startNew, endNew[,

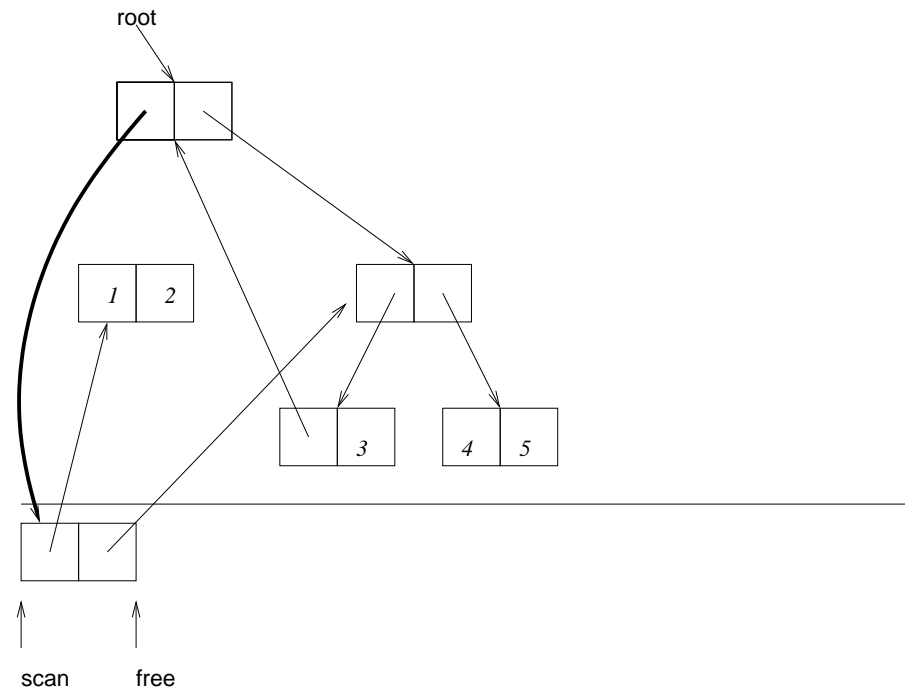
$s(\text{root}) \in \text{OLD}$. ALIVE = $\{p \mid p \text{ is reachable}\}$. Copies ALIVE to NEW in a “structure preserving way” and resumes allocation there.

The example from before:



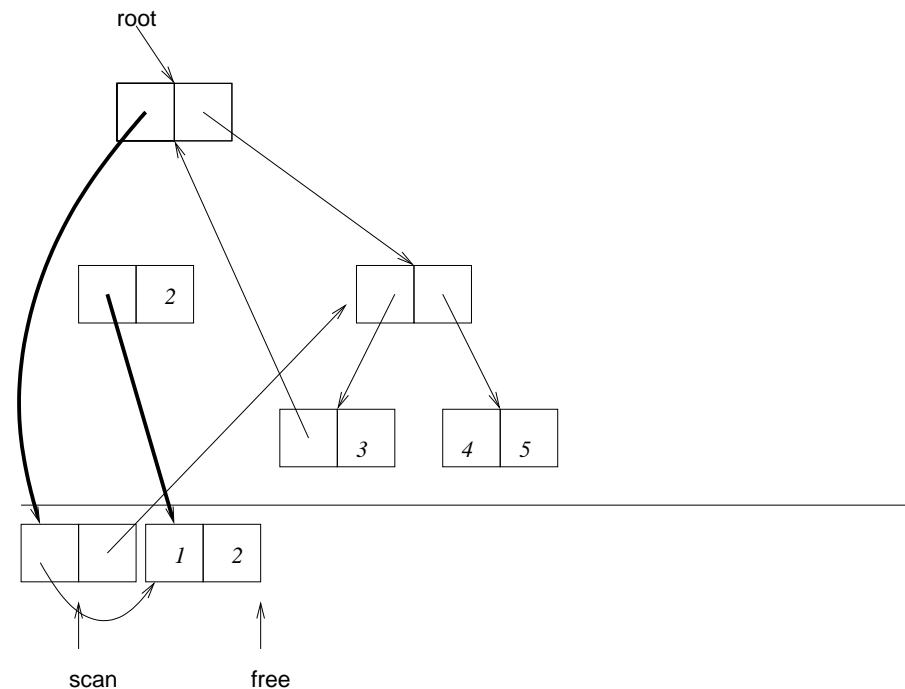
An Example

Initializing code: Copy the root cell and update the first component to point to the copy:



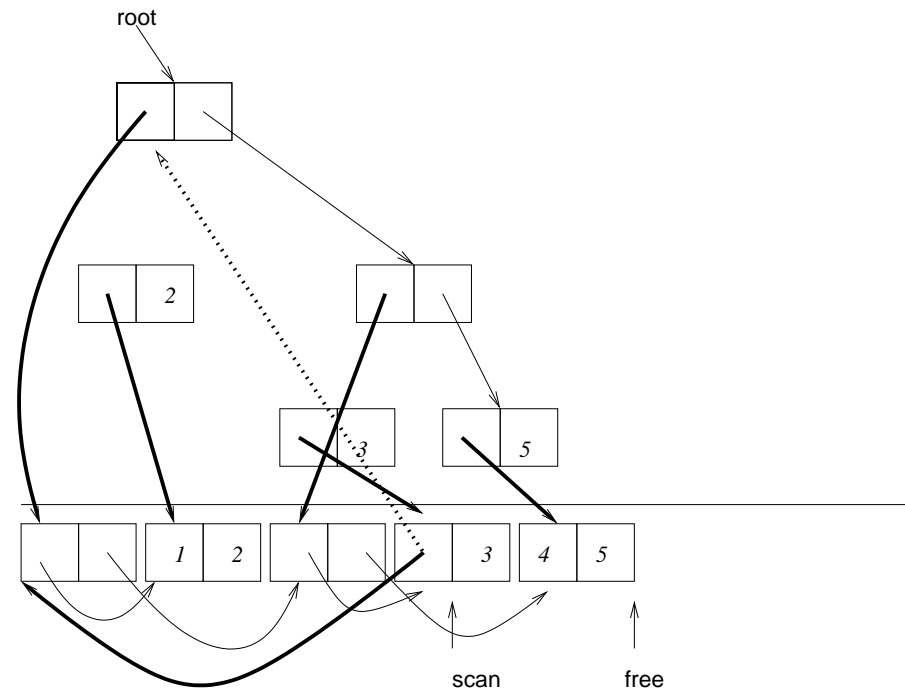
An Example (2)

Scanning a pointer-component (1): If the first component of the cell it points to is not a pointer into NEW, we just copy the cell and update its first component. Then we update the component we are scanning:



An Example (3)

Scanning a pointer-component (2): If the first component of the cell it points to *is* a pointer into NEW, we do *not* make another copy; we just update the component, we are scanning appropriately:



After this, nothing more interesting happens.



Extension of Separation Logic

For the proof, we will divide the pointers into groups, so we extend the term language with *finite sets of pointers*:



Extension of Separation Logic

For the proof, we will divide the pointers into groups, so we extend the term language with *finite sets of pointers*:

$$m ::= \dots \mid m^{\text{fs}} \oplus e \mid m^{\text{fs}} \ominus e \mid \text{Itv}(e, e) \mid \dots$$



Extension of Separation Logic

For the proof, we will divide the pointers into groups, so we extend the term language with *finite sets of pointers*:

$$m ::= \dots \mid m^{\text{fs}} \oplus e \mid m^{\text{fs}} \ominus e \mid \text{Itv}(e, e) \mid \dots$$

We will also need *finite relations*:

$$f ::= \dots \mid f \circ g \mid f \odot g$$



Extension of Separation Logic

For the proof, we will divide the pointers into groups, so we extend the term language with *finite sets of pointers*:

$$m ::= \dots \mid m^{\text{fs}} \oplus e \mid m^{\text{fs}} \ominus e \mid \text{Itv}(e, e) \mid \dots$$

We will also need *finite relations*:

$$f ::= \dots \mid f \circ g \mid f \odot g$$

Semantics for \odot : extension with identity on non-pointers:

$$\begin{aligned} \llbracket f \odot h \rrbracket = & \{ (p, n) \mid ((p, n) \in \llbracket h \rrbracket s \wedge n \notin \text{Ptr}) \vee \\ & (\exists p' \in \text{Ptr}. (p, p') \in \llbracket h \rrbracket s \wedge (p', n) \in \llbracket f \rrbracket s) \} \end{aligned}$$



Extension of Separation Logic (2)

We will also have new assertion forms. We mention some:



Extension of Separation Logic (2)

We will also have new assertion forms. We mention some:

- $p \in m, m_1 = m_2, (x_1, x_2) \in f, \text{iso}(f, m_1, m_2), \text{Tfun}(f, m)$.
Semantics is straightforward.



Extension of Separation Logic (2)

We will also have new assertion forms. We mention some:

- $p \in m, m_1 = m_2, (x_1, x_2) \in f, \text{iso}(f, m_1, m_2), \text{Tfun}(f, m)$.
Semantics is straightforward.
- *Iterated Separating Conjunction* over a finite set:

$$\forall_* p \in m. A(p)$$



Extension of Separation Logic (2)

We will also have new assertion forms. We mention some:

- $p \in m, m_1 = m_2, (x_1, x_2) \in f, \text{iso}(f, m_1, m_2), \text{Tfun}(f, m)$.
Semantics is straightforward.
- *Iterated Separating Conjunction* over a finite set:

$$\forall_* p \in m. A(p)$$

Semantics:

$s, h \Vdash \forall_* p \in m. A(p)$ iff

$\llbracket m \rrbracket s = \emptyset$ implies $s, h \Vdash \text{emp}$, and

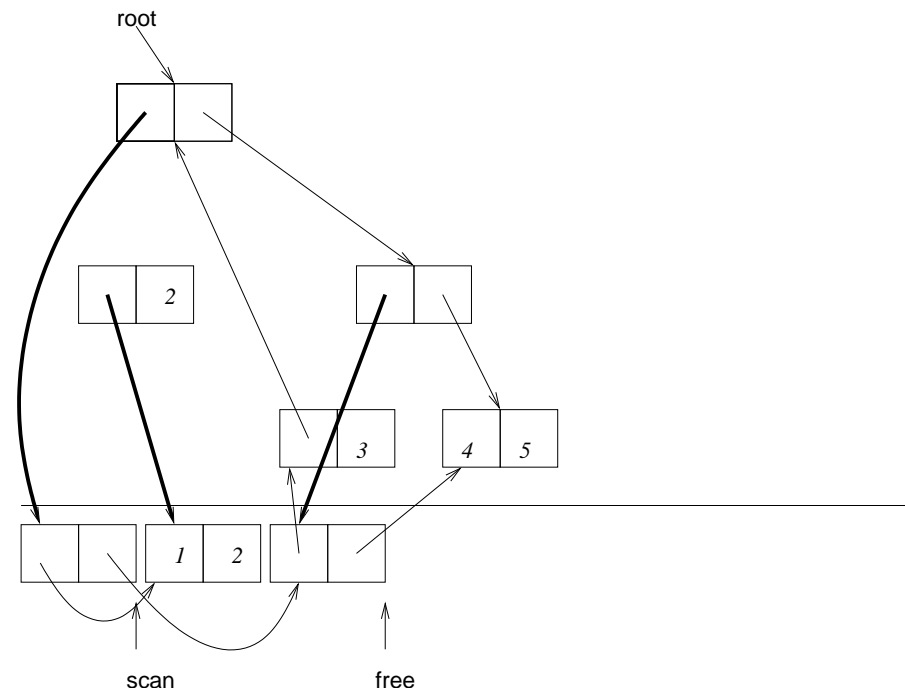
$\llbracket m \rrbracket s = \{p_1, \dots, p_k\}$ implies

$s, h \Vdash A(p_1) * \dots * A(p_k)$



Informal Analysis

At some stage of our example, we had the following situation:



The Proof

We will have



The Proof

We will have

- The sets from our partition



The Proof

We will have

- The sets from our partition
- Relations head and tail that record the initial heap



The Proof

We will have

- The sets from our partition
- Relations head and tail that record the initial heap
- φ , a bijection,

$$\varphi : \text{FORW} \rightarrow \text{BUSY} = \text{FIN} \cup \text{UNFIN} = [\text{startNew}, \text{free}[$$



The Proof

We will have

- The sets from our partition
- Relations head and tail that record the initial heap
- φ , a bijection,

$$\varphi : \text{FORW} \rightarrow \text{BUSY} = \text{FIN} \cup \text{UNFIN} = [\text{startNew}, \text{free}[$$

These are all added to the program as *auxiliary variables*, and will be part of the proof.



The Proof (2) - Set Analysis

Analysis of each set:



The Proof (2) - Set Analysis

Analysis of each set:

- UNFORW is not yet modified, so we can use head, tail.

$$A_{Uf} \equiv \forall_* p \in \text{UNFORW}. ((\exists q. (p, q) \in \text{head} \wedge p \mapsto q) * (\exists q'. (p, q') \in \text{tail} \wedge p + 4 \mapsto q')))$$



The Proof (2) - Set Analysis

Analysis of each set:

- UNFORW is not yet modified, so we can use head, tail.

$$A_{Uf} \equiv \forall_* p \in \text{UNFORW}. ((\exists q.(p, q) \in \text{head} \wedge p \mapsto q) * (\exists q'.(p, q') \in \text{tail} \wedge p + 4 \mapsto q'))$$

- FORW: First component points to cell determined by φ :

$$A_{Fw} \equiv \forall_* p \in \text{FORW}. (\exists q.(p, q) \in \varphi \wedge p \mapsto q, -)$$



The Proof (2) - Set Analysis

Analysis of each set:

- UNFORW is not yet modified, so we can use head, tail.

$$A_{Uf} \equiv \forall_* p \in \text{UNFORW}. ((\exists q.(p, q) \in \text{head} \wedge p \mapsto q) * (\exists q'.(p, q') \in \text{tail} \wedge p + 4 \mapsto q'))$$

- FORW: First component points to cell determined by φ :

$$A_{Fw} \equiv \forall_* p \in \text{FORW}. (\exists q.(p, q) \in \varphi \wedge p \mapsto q, -)$$

- FREE. Pointers here are in the domain of the heap:

$$A_{Fr} \equiv \forall_* p \in \text{FREE}. p \mapsto -, -$$

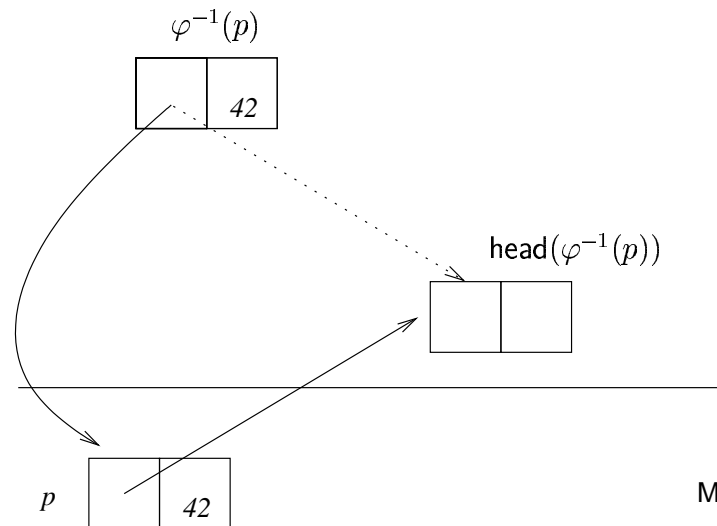


The Proof (3) - UNFIN

Analysis of each set, ct'd:

- UNFIN: Each cell is a copy of the cell in FORW that points to it:

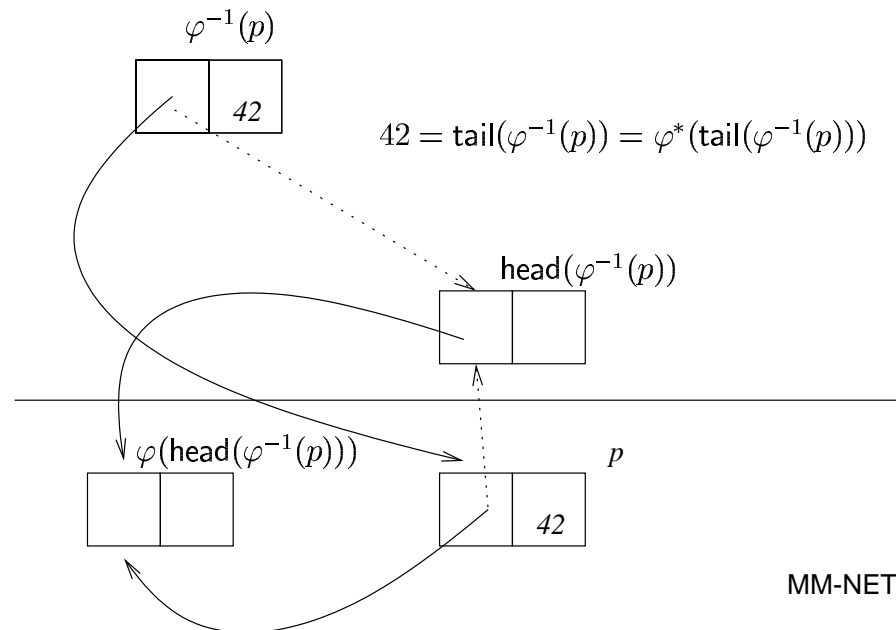
$$A_{Un} \equiv \forall_* p \in UNFIN. ((\exists q. (p, q) \in \text{head} \circ \varphi^T \wedge p \mapsto q) * (\exists q'. (p, q') \in \text{tail} \circ \varphi^T \wedge p + 4 \mapsto q'))$$



The Proof (4) - FIN

- FIN: scanned version of cells in UNFIN. Scanning means updating component to φ -value (but only if the component is a pointer). This is captured by \odot :

$$A_{Fn} \equiv \forall_* p \in \text{FIN}. ((\exists q. (p, q) \in \varphi \odot (\text{head} \circ \varphi^T) \wedge p \mapsto q) * (\exists q'. (p, q') \in \varphi \odot (\text{tail} \circ \varphi^T) \wedge p + 4 \mapsto q'))$$



The Proof (5) - Invariant

The Invariant:

$I \equiv$

$iso(\varphi, \text{FORW}, \text{BUSY}) \wedge \text{isUnion}(\text{FORW}, \text{UNFORW}, \text{ALIVE}) \wedge$
 $\#ALIVE \leq \#NEW \wedge \text{root} \in \text{FORW} \wedge \text{scan} \leq \text{free} \wedge$
 $\text{Disjoint}(\text{ALIVE}, \text{NEW}) \wedge \text{Ptr}(\text{free}) \wedge \text{Ptr}(\text{scan}) \wedge \text{Ptr}(\text{offset}) \wedge$
 $\text{Ptr}(\text{maxFree}) \wedge \text{Reachable}(\text{ALIVE}, \text{root}) \wedge$
 $\text{PtrRg}(\text{head}, \text{ALIVE}) \wedge \text{PtrRg}(\text{tail}, \text{ALIVE}) \wedge$
 $\text{Tfun}(\text{head}, \text{ALIVE}) \wedge \text{Tfun}(\text{tail}, \text{ALIVE}) \wedge$
 $(A_{Uf} * A_{Fw} * A_{Fn} * A_{Un} * A_{Fn})$



The Proof (6) - Local Reasoning

The most interesting part of the proof is when we copy a cell. We prove a local specification and use the Frame Rule. The local specification only mentions the “footprint” of the program fragment (x is cell pointed to by scan):



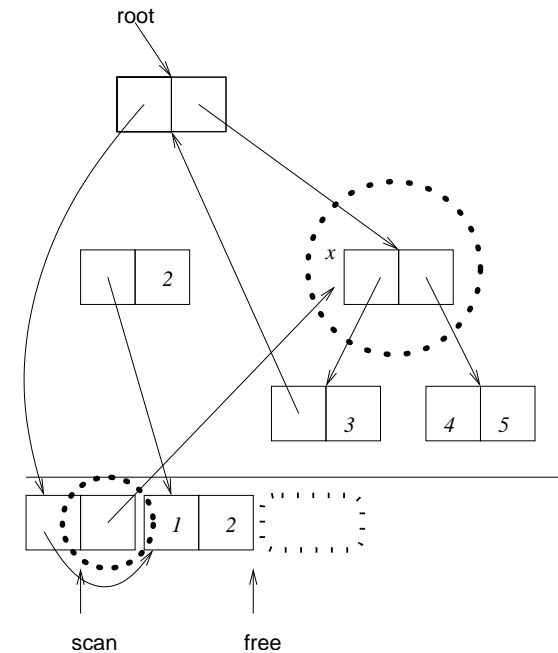
The Proof (6) - Local Reasoning

The most interesting part of the proof is when we copy a cell. We prove a local specification and use the Frame Rule. The local specification only mentions the “footprint” of the program fragment (x is cell pointed to by scan):

$$\{(\exists q. (x, q) \in \text{head} \wedge x \mapsto q)^* \\ (\exists q'. (x, q') \in \text{tail} \wedge x + 4 \mapsto q')^* \\ (\text{scan} \mapsto -) * (\text{free} \mapsto -, -)\}$$

CopyCell

$$\{((x \mapsto \text{free}, -) * (\text{scan} \mapsto \text{free})^* \\ (\text{free} \mapsto t_1, t_2)) \wedge \\ (x, t_1) \in \text{head} \wedge (x, t_2) \in \text{tail}\}$$



The Proof (7) - Remarks

Remarks about the Proof:



The Proof (7) - Remarks

Remarks about the Proof:

- The proof of the specification is entirely formal: uses only proof-rules, not “semantical arguments”.



The Proof (7) - Remarks

Remarks about the Proof:

- The proof of the specification is entirely formal: uses only proof-rules, not “semantical arguments”.
- The proof that the invariant is strong enough to conclude that there is a heap isomorphism, is *almost* logical: We prove formally that,

$$I \wedge \text{scan} = \text{free} \rightarrow (p \in \text{ALIVE} \wedge (p, q) \in \varphi \rightarrow (q \hookrightarrow r \leftrightarrow (p, r) \in \varphi \odot \text{head}))$$

Recall equation for heap isos:

$$h'(\varphi(p)) = \varphi^*(h(p))$$



Conclusion and Future Work



Conclusion and Future Work

- Formal proof of an algorithm that is used in practice.



Conclusion and Future Work

- Formal proof of an algorithm that is used in practice.
- Method of finite sets and relations is believed to be widely applicable, so further study is needed.



Conclusion and Future Work

- Formal proof of an algorithm that is used in practice.
- Method of finite sets and relations is believed to be widely applicable, so further study is needed.
- A more precise formulation of *interface issues* is needed.



Conclusion and Future Work

- Formal proof of an algorithm that is used in practice.
- Method of finite sets and relations is believed to be widely applicable, so further study is needed.
- A more precise formulation of *interface issues* is needed.
- A technical report will be available soon.

