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. 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. 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.

Preliminaries (1)

Setup: A *user language* and an *implemetation language*, both standard $\mathbf{while}$-languages, but with different memory interactions:

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

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

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

Preliminaries (2)

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

```
alloc(l,n1,n2) {
    if (any_space_left) {
        allocate 2 heap cells;
        store(n1,n2);
        return address;
    }
    else {
        Garbage collect;
        alloc(l,n1,n2);
    }
}
```

Preliminaries (3)

- All values allocated by the user language at runtime (through $\mathbf{cons}$-operations) are pairs of values, so the garbage collector only needs to deal with pairs of locations. A *pointer* is a first component of such a pair.

Preliminaries (3)

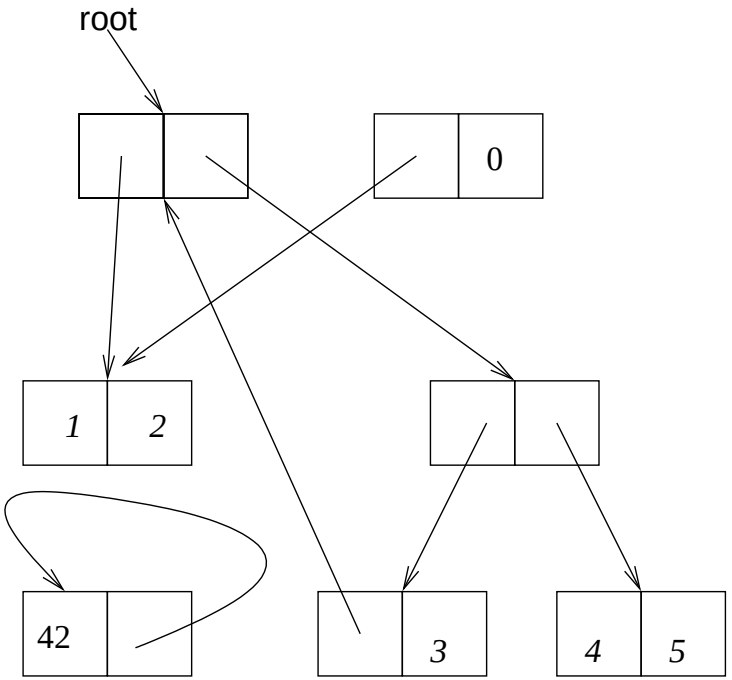
- All values allocated by the user language at runtime (through $\mathbf{cons}$-operations) are pairs of values, so the garbage collector only needs to deal with pairs of locations. A *pointer* is a first component of such a pair.
- Pointers are divisible by 8.

Preliminaries (3)

- All values allocated by the user language at runtime (through $\mathbf{cons}$-operations) are pairs of values, so the garbage collector only needs to deal with pairs of locations. A *pointer* is a first component of such a pair.
- Pointers are divisible by 8.
- A simplifying assumption: Only one pointer in user language, call it root (more “root pointers” do not add anything interesting to the proof).

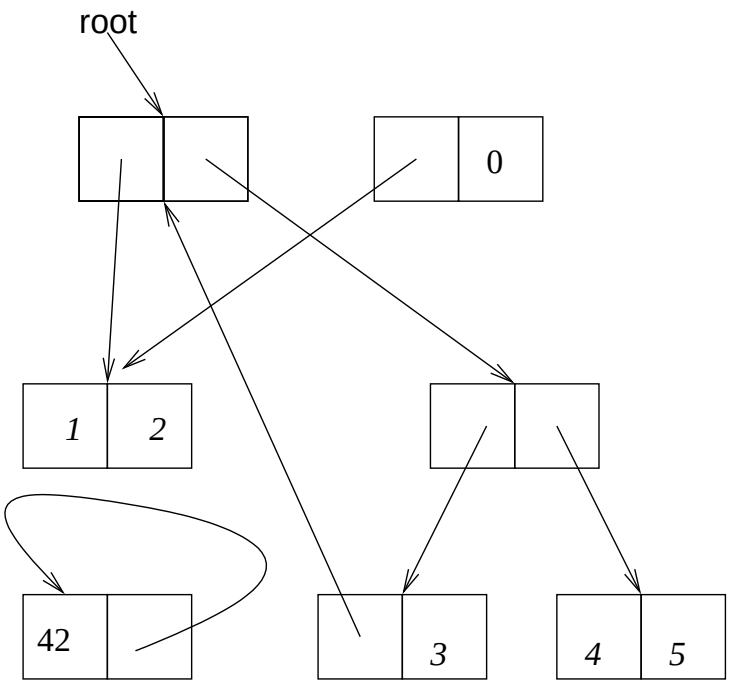
Preliminaries (4)

So a picture might look like this:



Preliminaries (4)

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 (5)

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

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

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

Preliminaries (5)

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

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

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

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

Preliminaries (5)

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

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

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

- (s, h) is a *garbage collected version* of (s', h'), if there is a heap isomorphism $\varphi : prune(s, h) \cong prune(s', h')$. 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 contigous "semi-spaces" of equal size,

$$\mathsf{OLD} = [\mathsf{startOld}, \mathsf{endOld}[\quad \text{and} \quad \mathsf{NEW} = [\mathsf{startNew}, \mathsf{endNew}[,$$

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

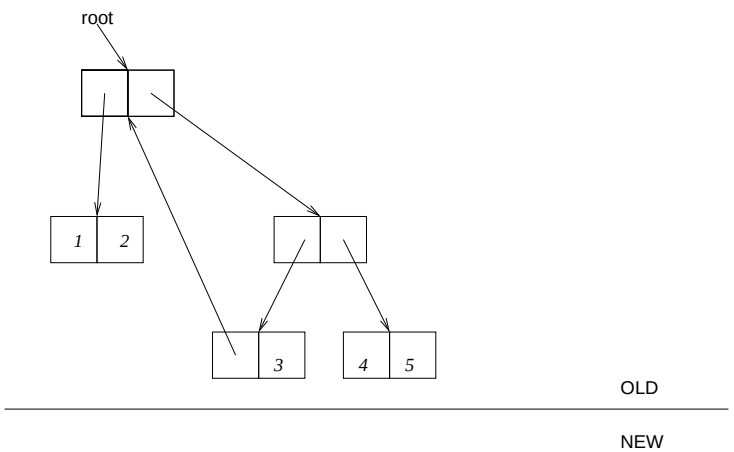
Cheney's Algorithm (1970)

Assumes 2 contigous "semi-spaces" of equal size,

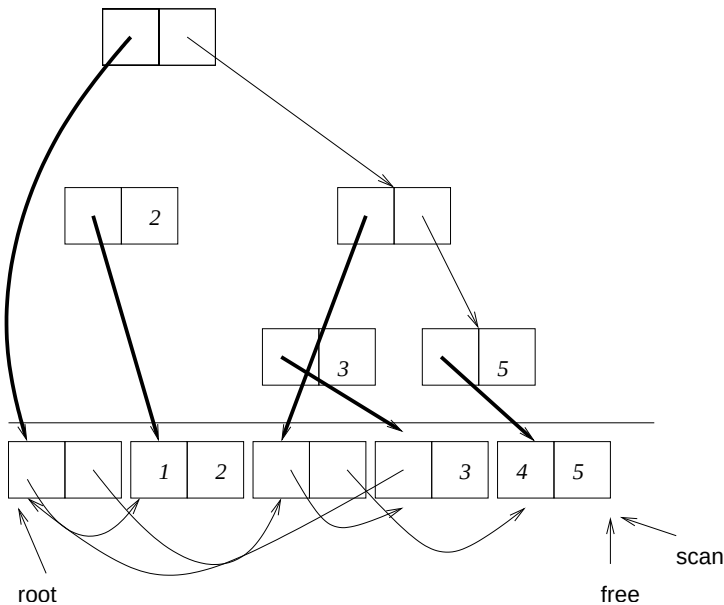
$$\mathsf{OLD} = [\mathsf{startOld}, \mathsf{endOld}[\quad \text{and} \quad \mathsf{NEW} = [\mathsf{startNew}, \mathsf{endNew}[,$$

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

The example from before:

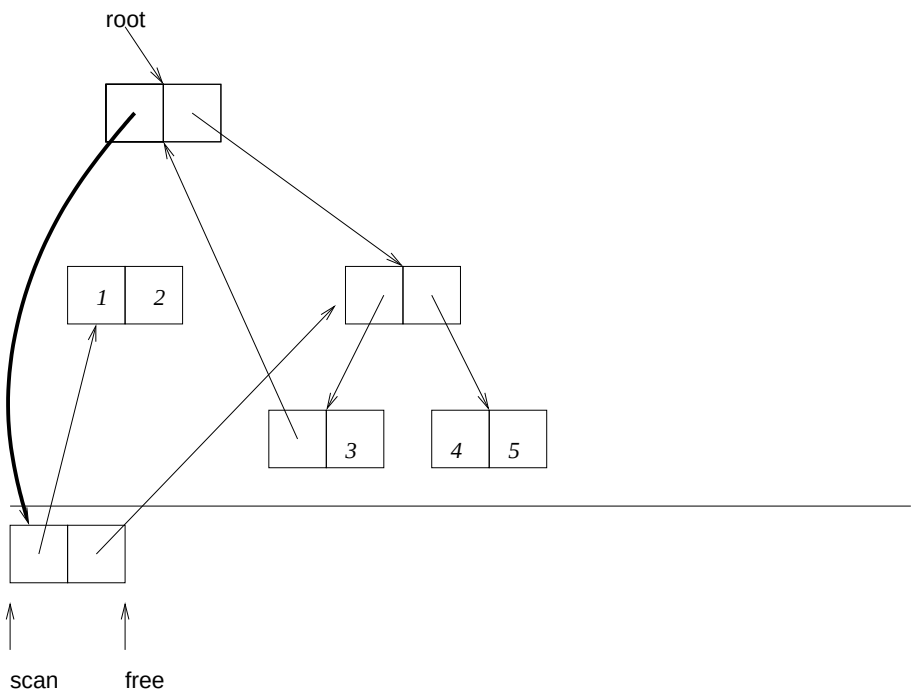


$\rightsquigarrow$



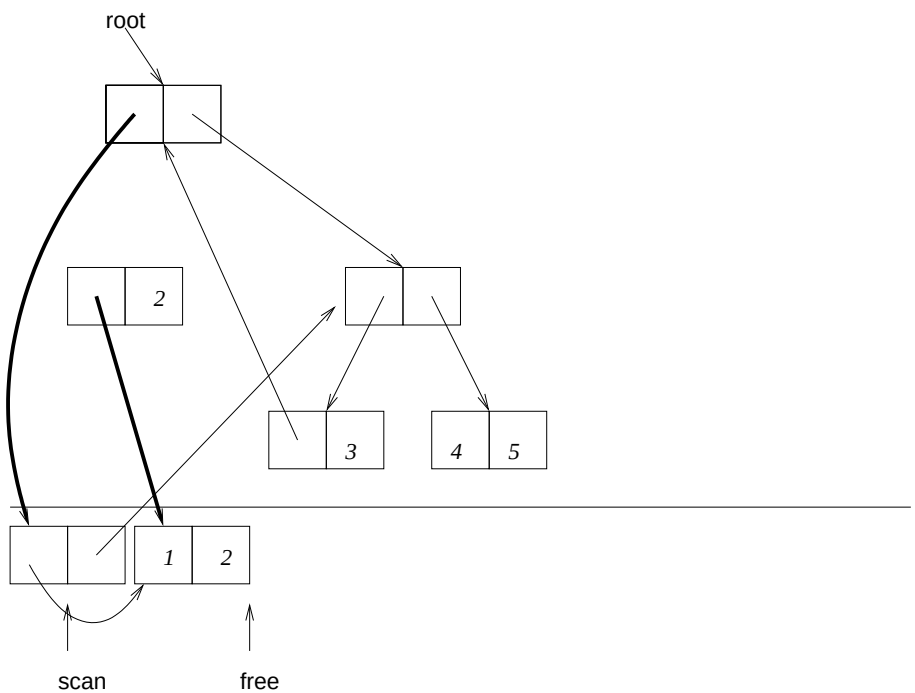
An Example

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



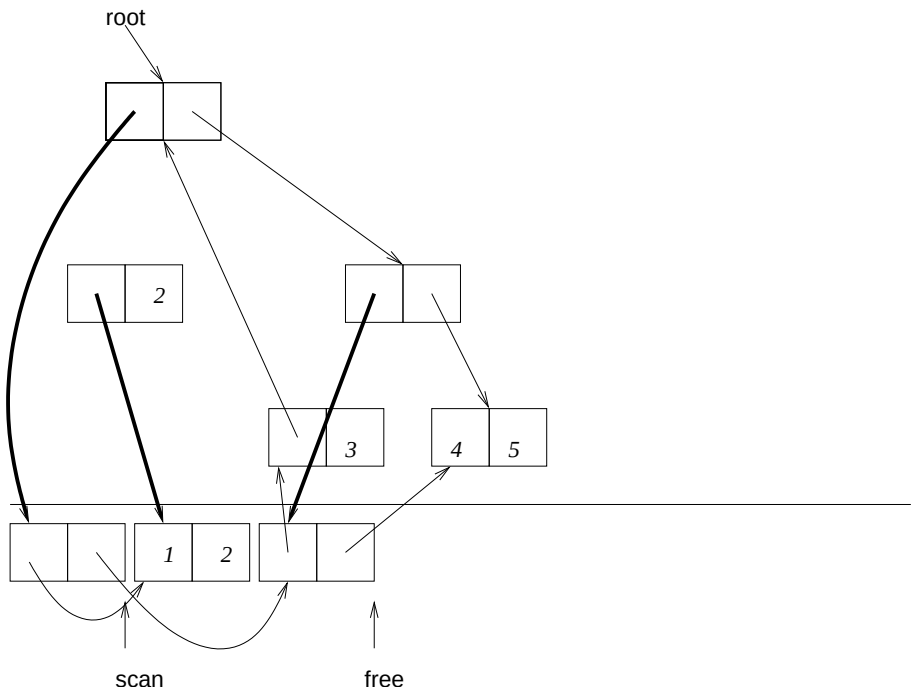
An Example (2)

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:



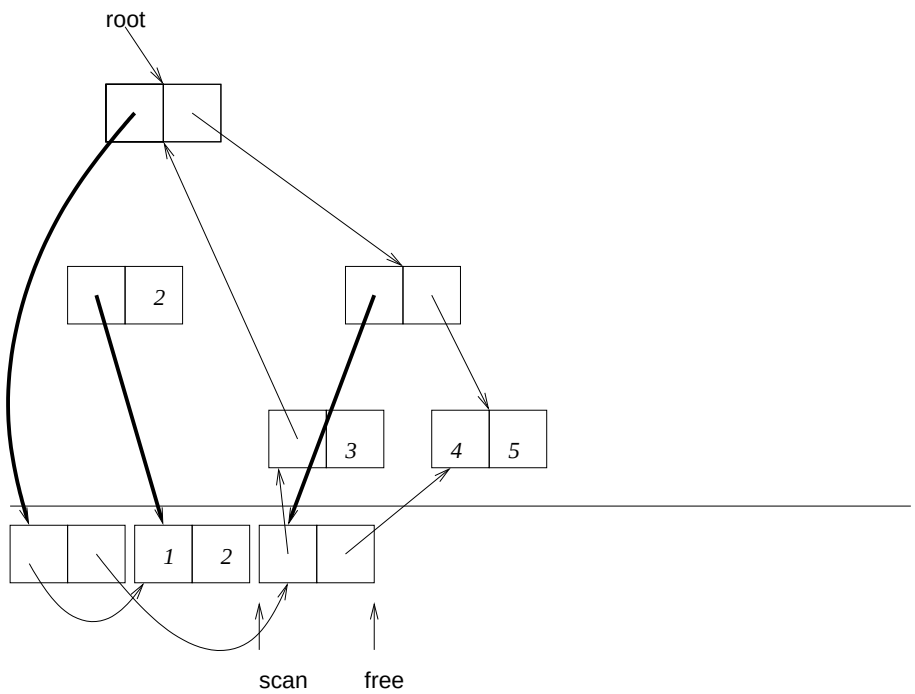
An Example (3)

... and again:



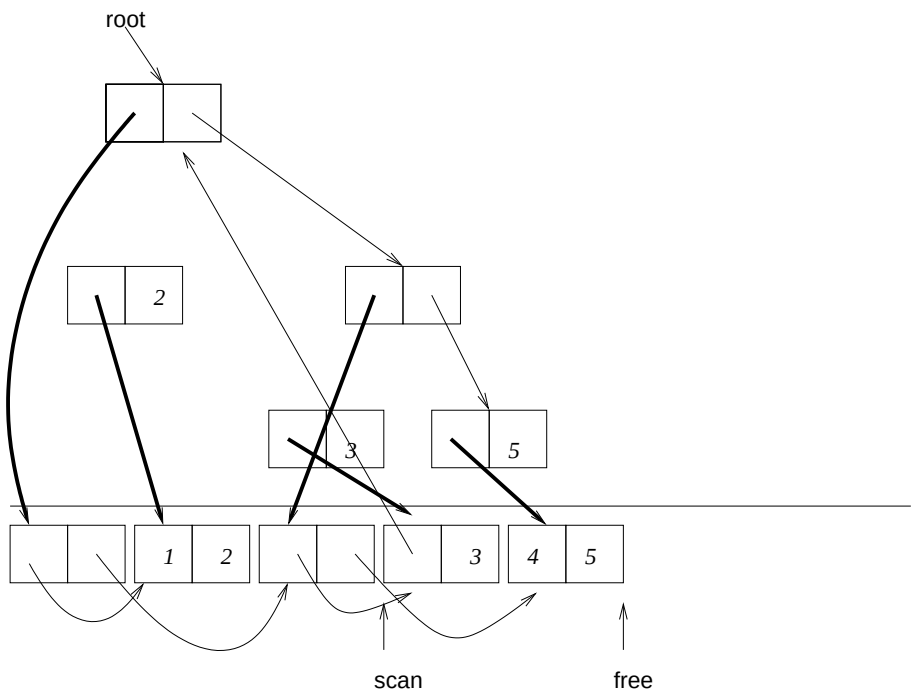
An Example (4)

Scanning a non-pointer component: Nothing happens.



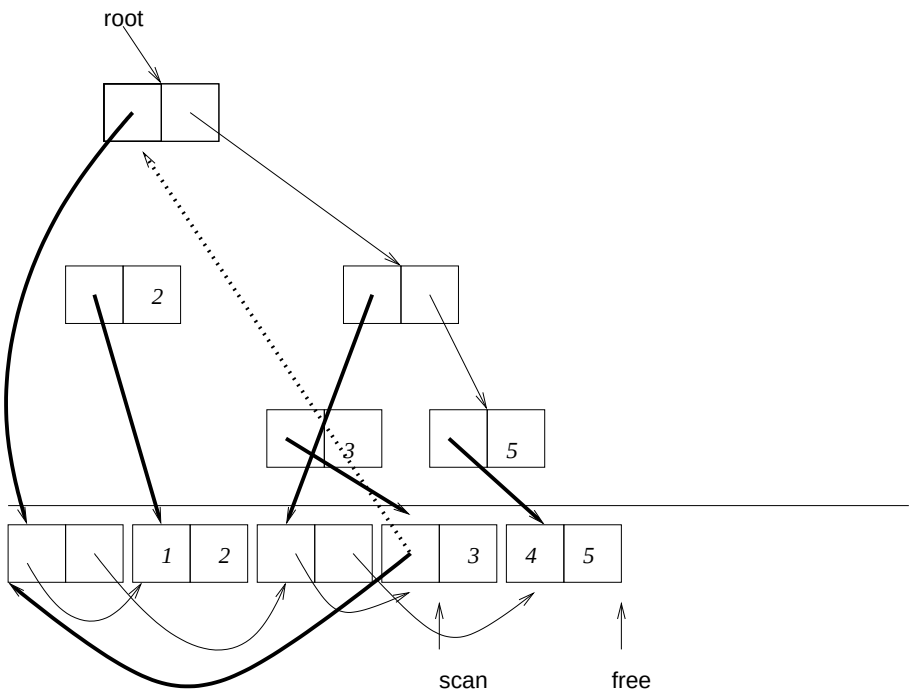
An Example (5)

Scanning two pointer components as before:



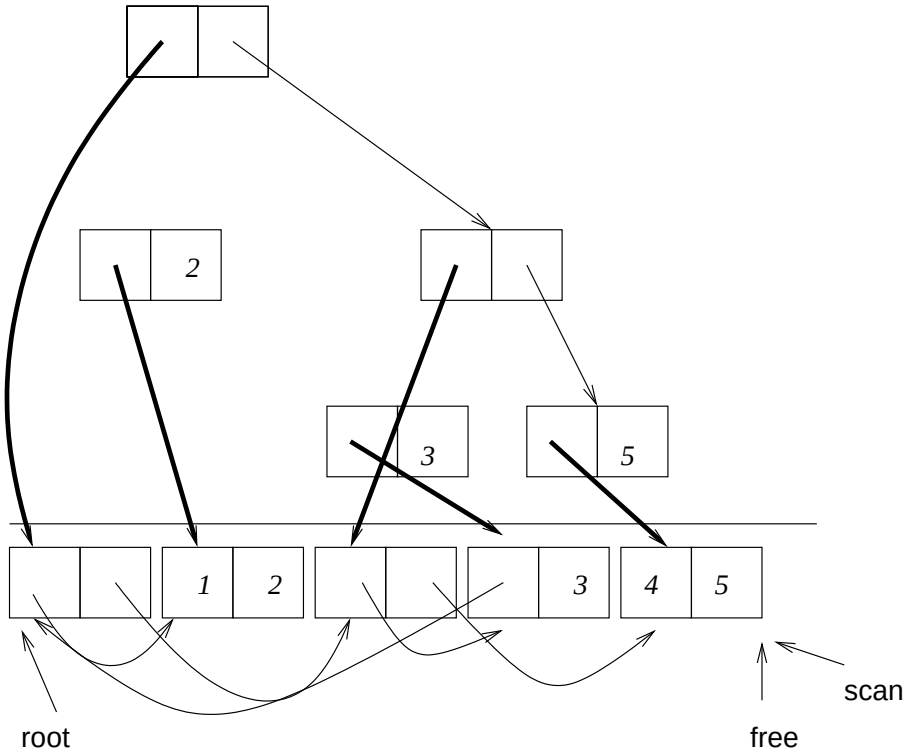
An Example (6)

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:



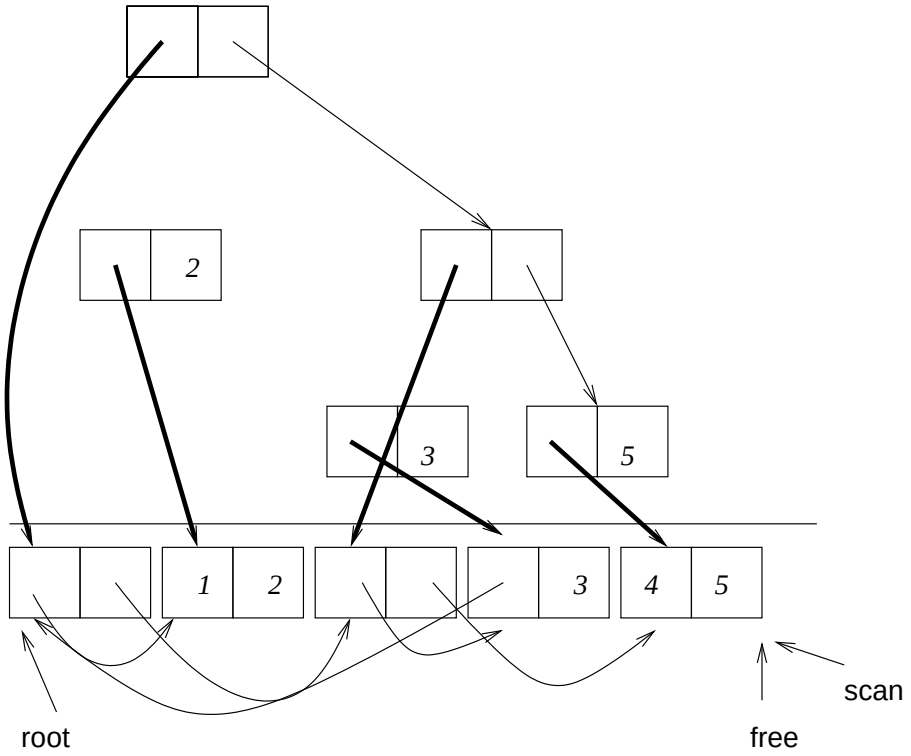
An Example (7)

After this, nothing more interesting happens, and we update `root`:



An Example (7)

After this, nothing more interesting happens, and we update `root`:



...and we're done!

Extension of Separation Logic

To formalize our partition, we extend the term language with *finite sets of pointers*:

Extension of Separation Logic

To formalize our partition, we extend the term language with *finite sets of pointers*:

$$m \quad ::= \quad \cdots \mid m^{\mathsf{fs}} \oplus e \mid m^{\mathsf{fs}} \ominus e \mid Itv(e, e) \mid \cdots$$

Extension of Separation Logic

To formalize our partition, we extend the term language with *finite sets of pointers*:

$$m \quad ::= \quad \cdots \mid m^{\mathsf{fs}} \oplus e \mid m^{\mathsf{fs}} \ominus e \mid Itv(e, e) \mid \cdots$$

We will also need *finite relations*:

$$f \quad ::= \quad \cdots \mid f \circ g \mid f \circledcirc g$$

Extension of Separation Logic

To formalize our partition, we extend the term language with *finite sets of pointers*:

$$m \quad ::= \quad \cdots \mid m^{\mathsf{fs}} \oplus e \mid m^{\mathsf{fs}} \ominus e \mid Itv(e,e) \mid \cdots$$

We will also need *finite relations*:

$$f \quad ::= \quad \cdots \mid f \circ g \mid f \circledcirc g$$

Semantics for $\circledcirc$: extension with identity on non-pointers:

$$\begin{aligned} [\![f \circledcirc h]\!] \quad = \quad & \{(p,n) \mid ((p,n) \in [\![h]\!]s \wedge n \notin Ptr) \vee \\ & (\exists p' \in Ptr.(p,p') \in [\![h]\!]s \wedge (p',n) \in [\![f]\!]s)\} \end{aligned}$$

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:

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

Extension of Separation Logic (2)

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

- $p \in m$, $m_1 = m_2$, $(x_1, x_2) \in f$, $\mathsf{iso}(f, m_1, m_2)$, $\mathsf{Tfun}(f, m)$. 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:

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

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

Semantics:

$$\begin{array}{l} s, h \Vdash \forall_* p \in m.\ A(p) \text{ iff} \\ [\![m]\!]s = \varnothing \text{ implies } s, h \Vdash \mathsf{emp}, \text{ and} \\ [\![m]\!]s = \{p_1, \ldots, p_k\} \text{ implies} \\ s, h \Vdash A(p_1) * \cdots * A(p_k) \end{array}$$

Interlude

Recall from Owicki, Gries:

> Let C be a command, and let AV be a set of variables that appear in C only in assignments $x := E$, where $x \in AV$. Then AV is an *auxiliary variable set* for C.

Interlude

Recall from Owicki, Gries:

> Let C be a command, and let AV be a set of variables that appear in C only in assignments $x := E$, where $x \in AV$. Then AV is an *auxiliary variable set* for C.
>
> Let AV be an auxiliary variable set for C', and P and Q assertions not containing free variables from AV. Let C be the command obtained from C' by deleting all assignments to the variables in AV. Then
>
> $$\frac{\{P\}\ C'\ \{Q\}}{\{P\}\ C\ \{Q\}}$$

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):

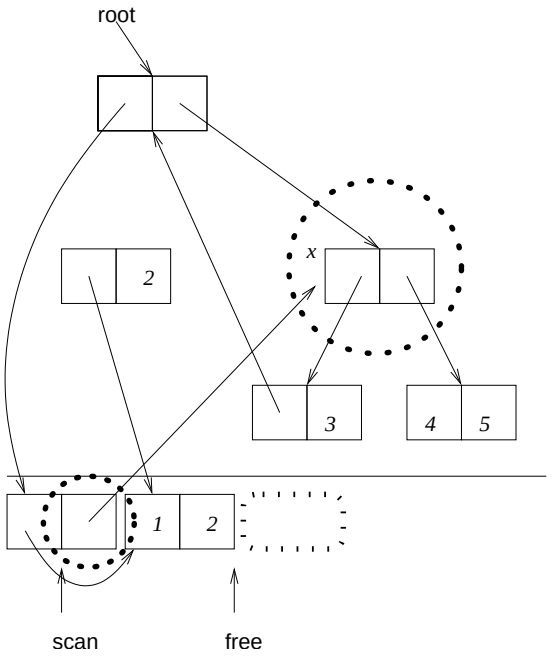
Local Reasoning

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):

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

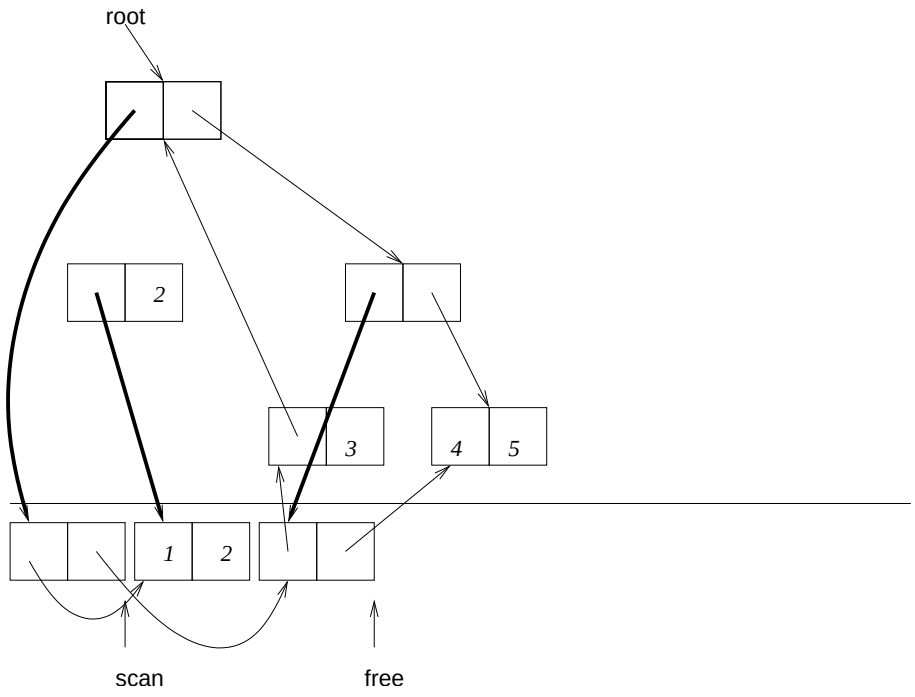
CopyCell

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



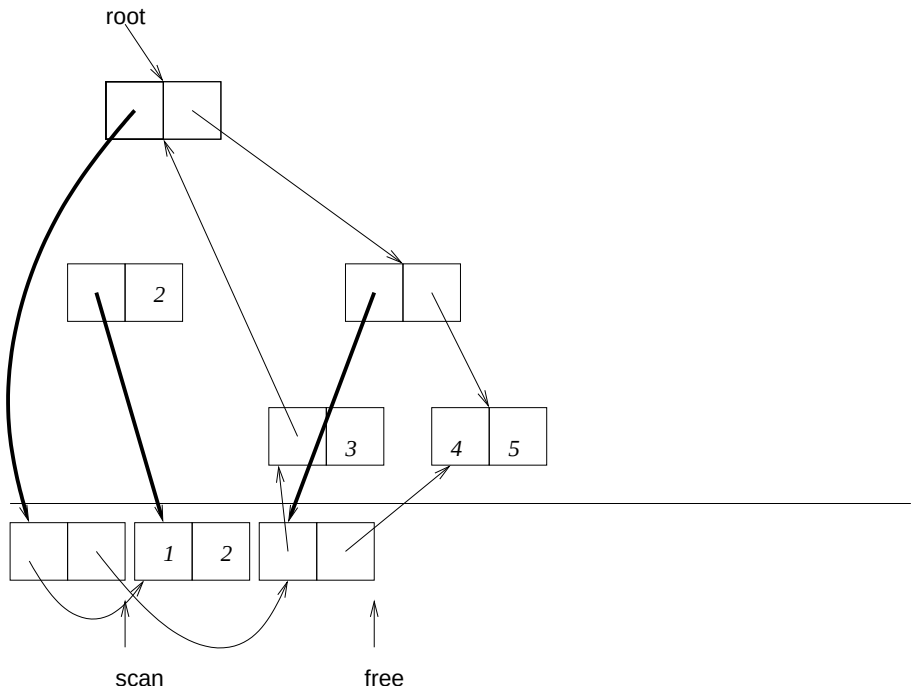
Informal Analysis

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



Informal Analysis

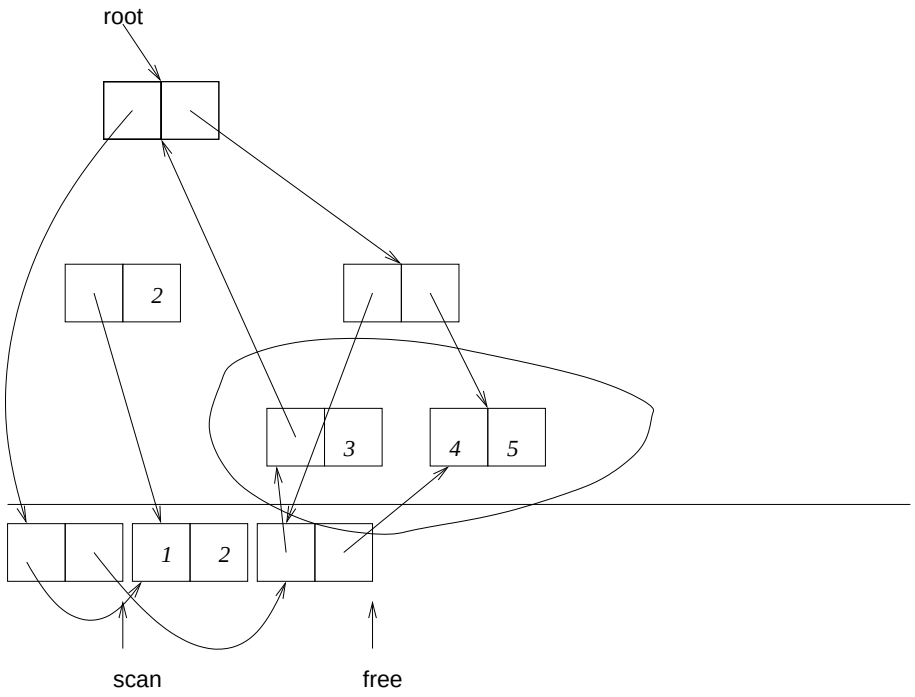
At some stage of our example, we had the following situation:



We will partition the cells into different “kinds”.

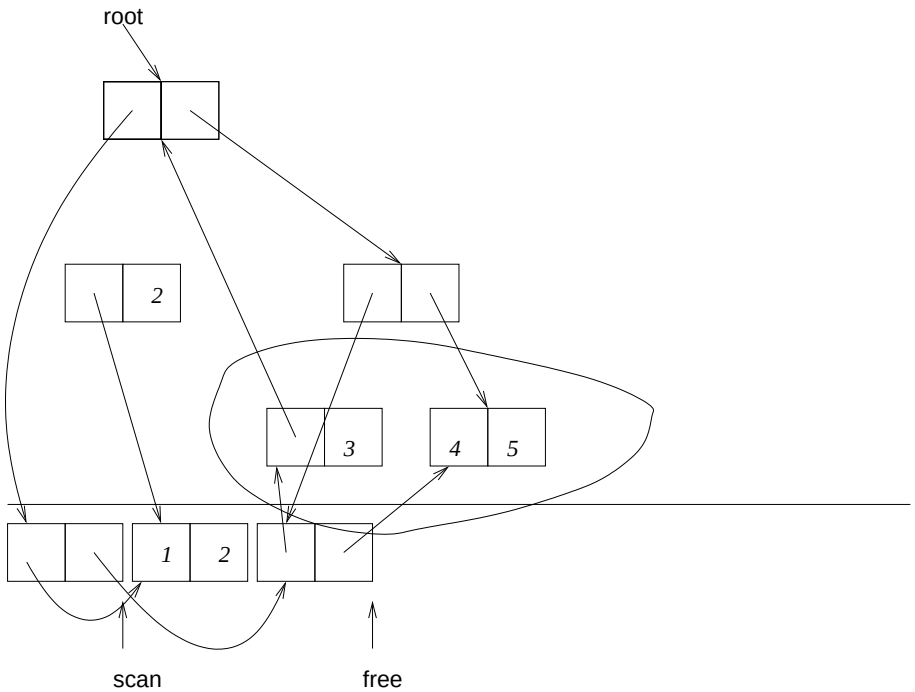
Informal Analysis (2)

Sets of Pointers: Some cells in OLD have not yet been copied yet.



Informal Analysis (2)

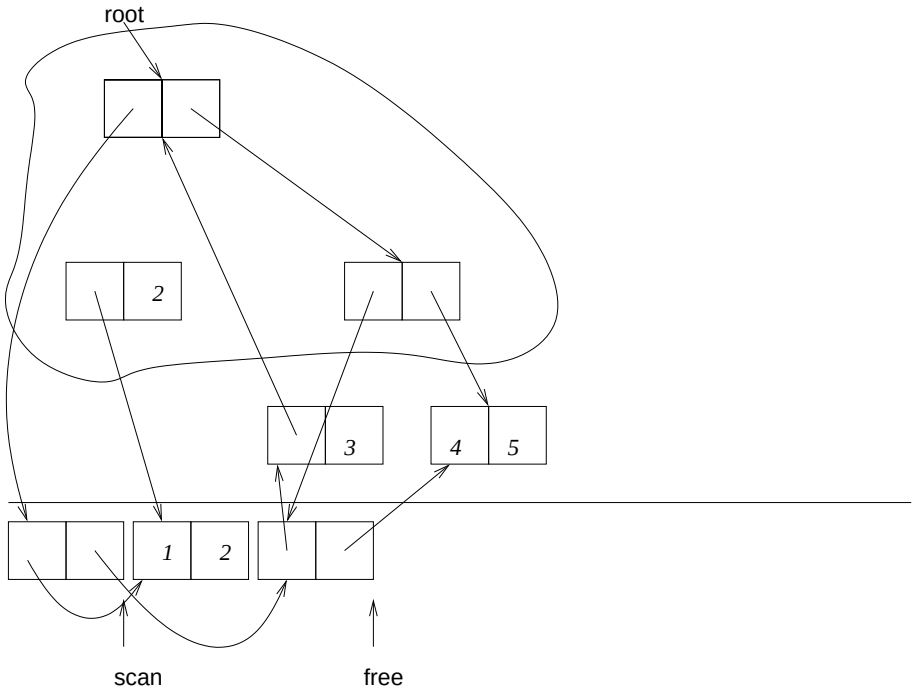
Sets of Pointers: Some cells in OLD have not yet been copied yet.



We call the set of these pointers UNFORW.

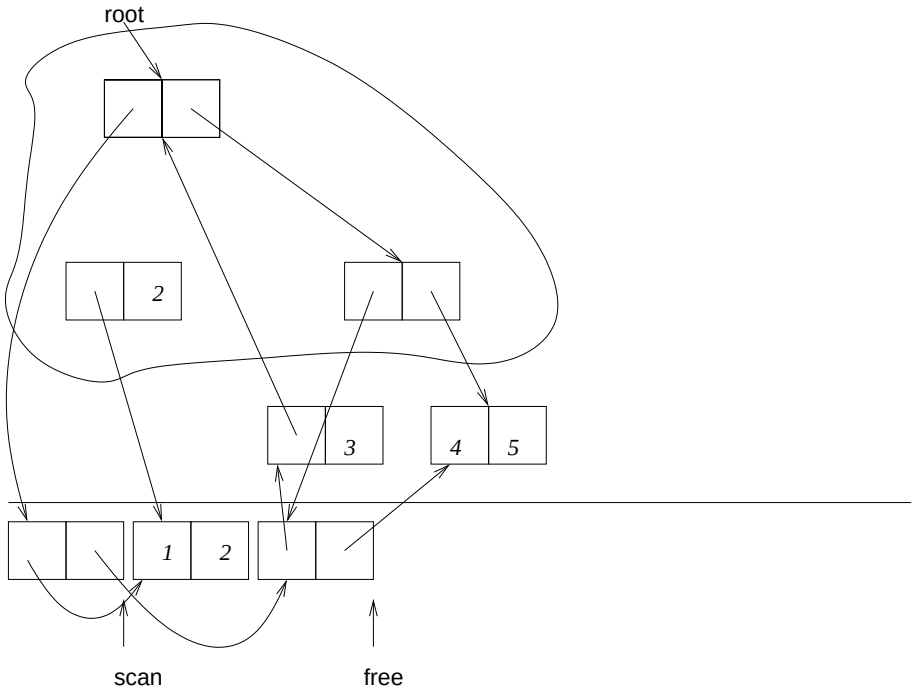
Informal Analysis (3)

Sets of Pointers: Some cells in OLD have been copied, they're marked with a "forward pointer" in the first component.



Informal Analysis (3)

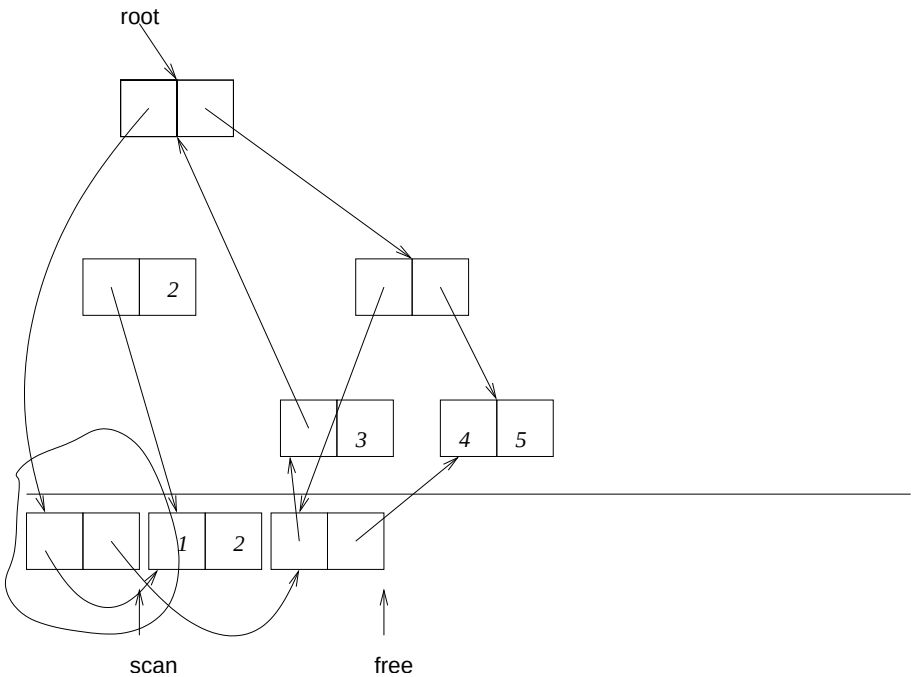
Sets of Pointers: Some cells in OLD have been copied, they're marked with a "forward pointer" in the first component.



We call the set of these pointers FORW.

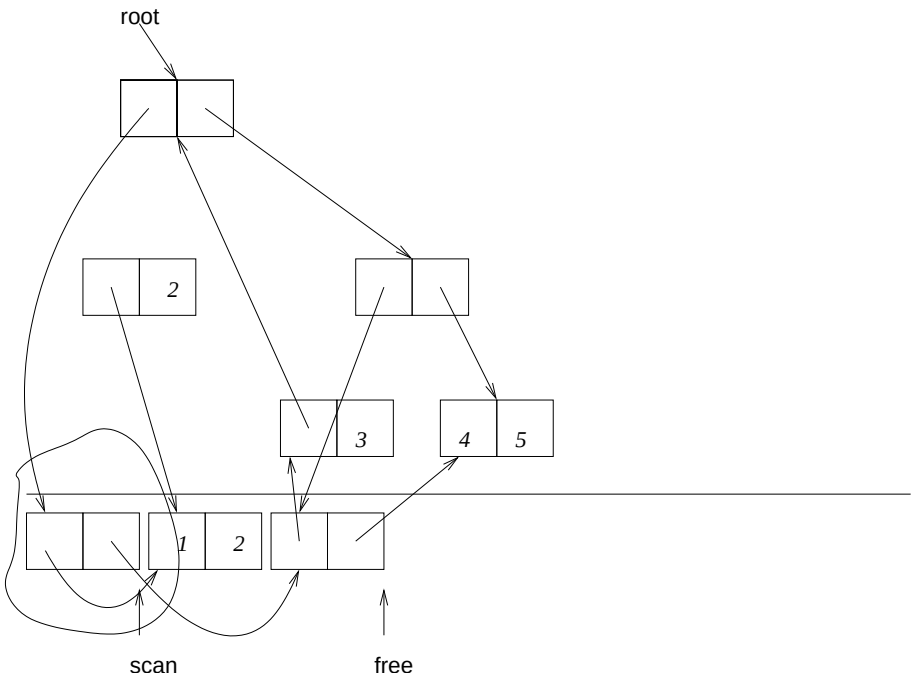
Informal Analysis (4)

Sets of Pointers: Some cells in NEW have been copied and scanned, they will not be modified (or read) further.



Informal Analysis (4)

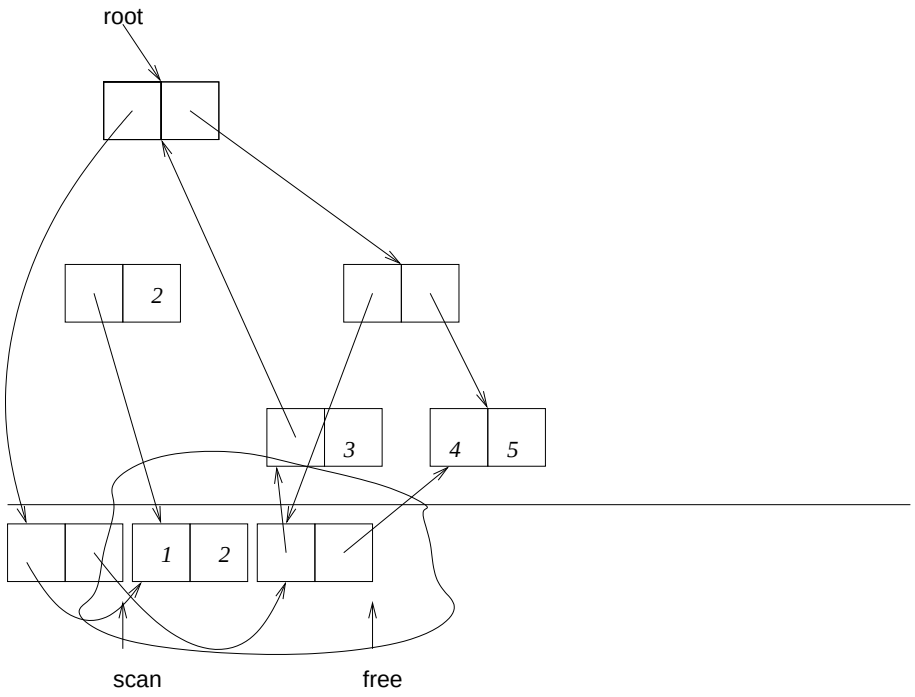
Sets of Pointers: Some cells in NEW have been copied and scanned, they will not be modified (or read) further.



We call the set of these pointers FIN. $\mathsf{FIN} = [\mathsf{startNew}, \mathsf{scan}[$.

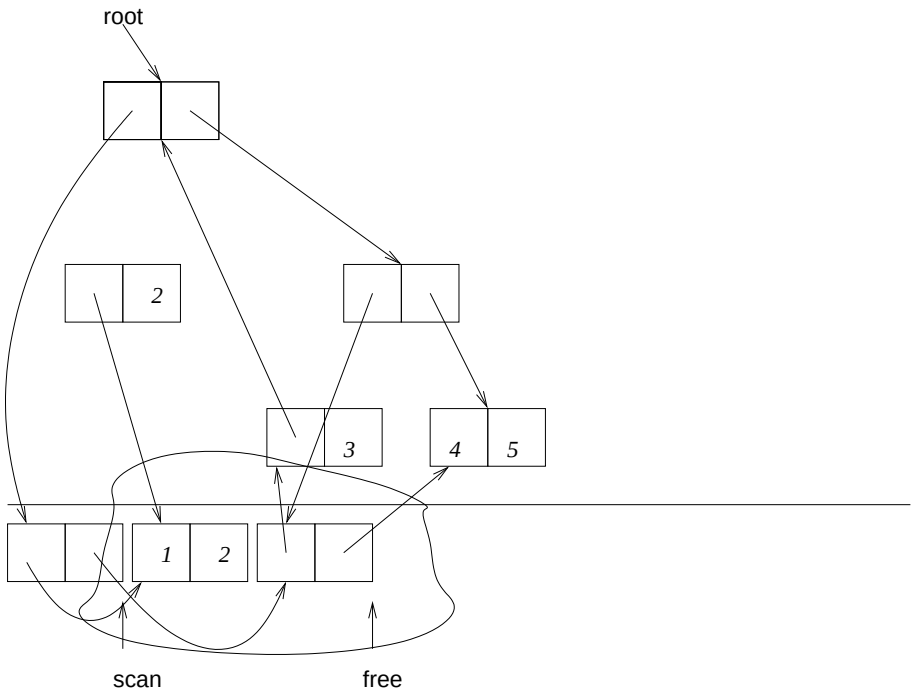
Informal Analysis (5)

Sets of Pointers: Some cells in NEW have been copied but not scanned.



Informal Analysis (5)

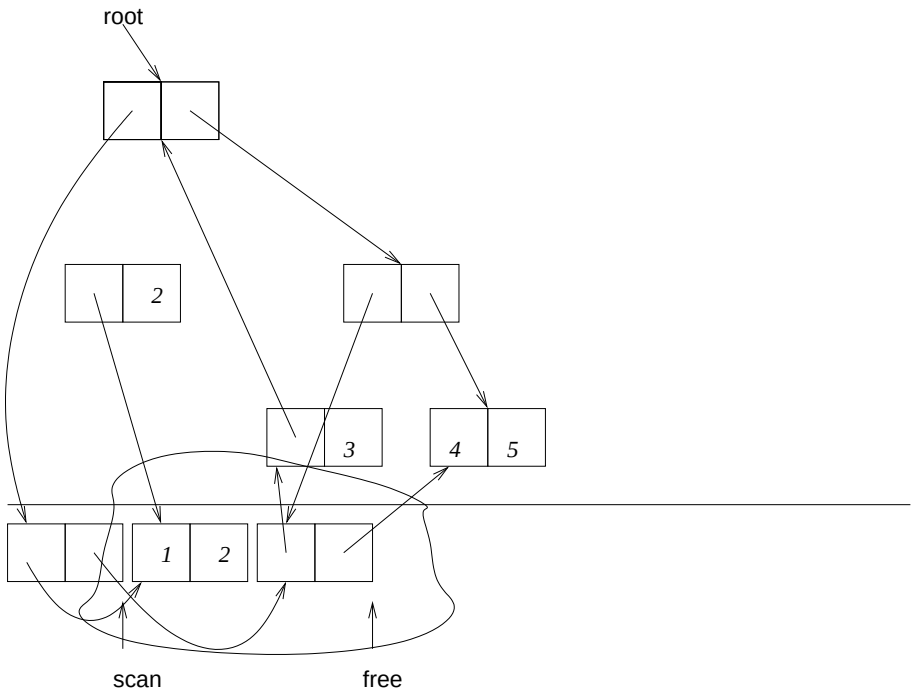
Sets of Pointers: Some cells in NEW have been copied but not scanned.



We call the set of these pointers UNFIN. $\mathrm{UNFIN} = [\mathrm{scan}, \mathrm{free}[$.

Informal Analysis (5)

Sets of Pointers: Some cells in NEW have been copied but not scanned.



We call the set of these pointers UNFIN. $\mathsf{UNFIN} = [\mathsf{scan}, \mathsf{free}[$.

Finally, $\mathsf{FREE} = [\mathsf{free}, \mathsf{endNew}[$ is “free for allocation”.

The Proof

We will have

The Proof

We will have

- The sets mentioned before

The Proof

We will have

- The sets mentioned before
- Relations head and tail that record the initial heap

The Proof

We will have

- The sets mentioned before
- Relations head and tail that record the initial heap
- φ, a bijection,

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

The Proof

We will have

- The sets mentioned before
- Relations head and tail that record the initial heap
- φ, a bijection,

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

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

The Proof (2)

Analysis of each set:

The Proof (2)

Analysis of each set:

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

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

The Proof (2)

Analysis of each set:

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

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

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

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

The Proof (2)

Analysis of each set:

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

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

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

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

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

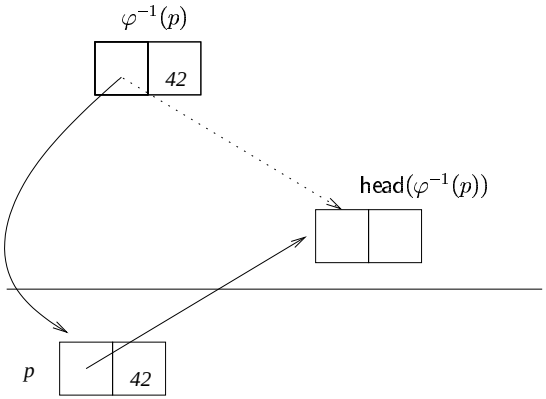
$$A_{\mathsf{Fr}} \equiv \forall_* p \in \mathsf{FREE}.\ p \mapsto -, -$$

The Proof (3)

Analysis of each set, ct'd:

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

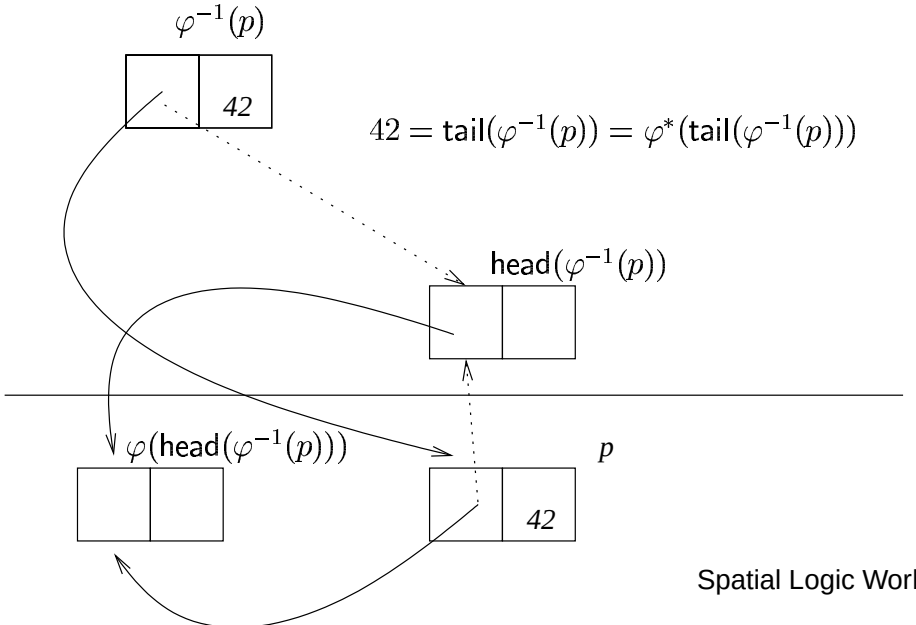
$$A_{\mathsf{Un}} \equiv \forall_* p \in \mathsf{UNFIN}.\ ((\exists q.(p,q) \in \mathsf{head} \circ \varphi^{\mathsf{T}} \wedge p \mapsto q) * (\exists q'.(p,q') \in \mathsf{tail} \circ \varphi^{\mathsf{T}} \wedge p+4 \mapsto q'))$$



The Proof (4)

- 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 $\circledcirc$:

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



The Proof (5)

The Precondition:

$$
\begin{aligned}
&\mathsf{InitAss} \equiv \\
&\mathsf{Ptr}(\mathsf{startNew}) \wedge \mathsf{Ptr}(\mathsf{endNew}) \wedge \mathsf{Ptr}(\mathsf{root}) \wedge \mathsf{Disjoint}(\mathsf{OLD}, \mathsf{NEW}) \wedge \\
&\mathsf{SbSet}(\mathsf{ALIVE}, \mathsf{OLD}) \wedge \mathsf{Reachable}(\mathsf{ALIVE}, \mathsf{root}) \wedge \\
&\#\mathsf{NEW} = \#\mathsf{OLD} \wedge \mathsf{PtrRg}(\mathsf{head}, \mathsf{ALIVE}) \wedge \mathsf{PtrRg}(\mathsf{tail}, \mathsf{ALIVE}) \wedge \\
&\mathsf{Tfun}(\mathsf{head}, \mathsf{ALIVE}) \wedge \mathsf{Tfun}(\mathsf{tail}, \mathsf{ALIVE}) \wedge \\
&((\forall_* p \in \mathsf{ALIVE}.\ ((\exists q.\ (p, q) \in \mathsf{head} \wedge p \mapsto q) * \\
&\quad (\exists q.\ (p, q') \in \mathsf{tail} \wedge p + 4 \mapsto q'))) * \\
&(\forall_* p \in \mathsf{NEW}. p \mapsto -, -) * \mathsf{T})
\end{aligned}
$$

The T deals with “unreachable” cells (they are framed out).

The Proof (6)

The Invariant:

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

The Proof (7)

Remarks about the Proof:

The Proof (7)

Remarks about the Proof:

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

The Proof (7)

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 logically that,

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

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.