

# A Verified Garbage Collector for Gallina

Shengyi Wang<sup>†</sup>, Anshuman Mohan<sup>†</sup>, Qinxiang Cao<sup>‡</sup>, Aquinas Hobor<sup>†</sup>



APLAS NIER  
December 1, 2019

Verify **graph-manipulating** programs  
written in **executable C**  
with **machine-checked** correctness proofs

Verify **graph-manipulating** programs  
written in **executable C**  
with **machine-checked** correctness proofs

Hard, but ubiquitous in critical areas



## **Certifying Graph-Manipulating C Programs via Localizations within Data Structures**

SHENGYI WANG, National University of Singapore, Singapore

QINXIANG CAO, Shanghai Jiao Tong University, China

ANSHUMAN MOHAN, National University of Singapore, Singapore

AQUINAS HOBOR, National University of Singapore, Singapore

VST + CompCert + 25000 LOC library



## Certifying Graph-Manipulating C Programs via Localizations within Data Structures

SHENGYI WANG, National University of Singapore, Singapore

QINXIANG CAO, Shanghai Jiao Tong University, China

ANSHUMAN MOHAN, National University of Singapore, Singapore

AQUINAS HOBOR, National University of Singapore, Singapore

VST + CompCert + 25000 LOC library

Powerful enough to verify **executable code**  
against **realistic specifications**  
expressed with **mathematical graphs**



## Certifying Graph-Manipulating C Programs via Localizations within Data Structures

SHENGYI WANG, National University of Singapore, Singapore

QINXIANG CAO, Shanghai Jiao Tong University, China

ANSHUMAN MOHAN, National University of Singapore, Singapore

AQUINAS HOBOR, National University of Singapore, Singapore

VST + CompCert + 25000 LOC library

Powerful enough to verify **executable code**  
against **realistic specifications**  
expressed with **mathematical graphs**

[Wang *et. al.*, PACMPL OOPSLA 2019]



Gallina  $\rightsquigarrow$  CompCert C  $\rightsquigarrow$  Assembly



Gallina  $\rightsquigarrow$  CompCert C  $\rightsquigarrow$  Assembly

Gallina assumes **infinite** memory  
but CompCert C has a **finite** heap





Gallina  $\rightsquigarrow$  CompCert C  $\rightsquigarrow$  Assembly

Gallina assumes **infinite** memory  
but CompCert C has a **finite** heap

Solution: garbage collect the CompCert C code



Gallina  $\rightsquigarrow$  CompCert C  $\rightsquigarrow$  Assembly

Gallina assumes **infinite** memory  
but CompCert C has a **finite** heap

Solution: garbage collect the CompCert C code

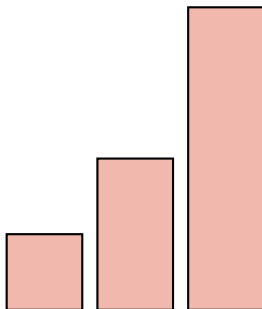
New problem: verify the garbage collector

# Our Garbage Collector

GC has jurisdiction over the heap



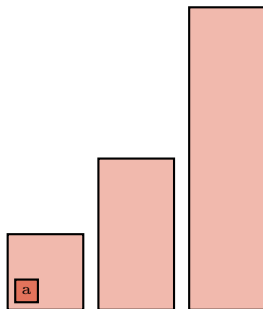
GC has jurisdiction over the heap



# Our Garbage Collector

GC has jurisdiction over the heap

Mutator **allocs** in special subheap

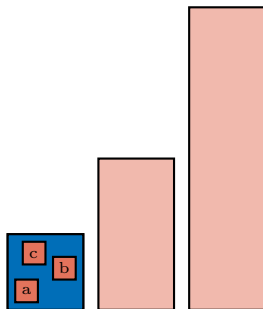


# Our Garbage Collector

GC has jurisdiction over the heap

Mutator **allocs** in special subheap

If subheap is full

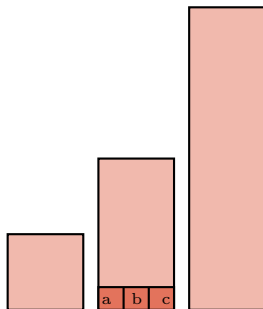


# Our Garbage Collector

GC has jurisdiction over the heap

Mutator **allocs** in special subheap

If subheap is full **call GC**

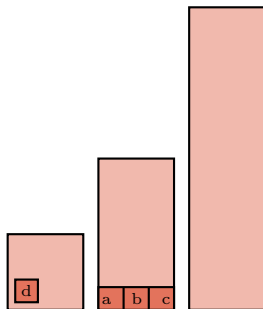


# Our Garbage Collector

GC has jurisdiction over the heap

Mutator **allocs** in special subheap

If subheap is full **call GC** and try again



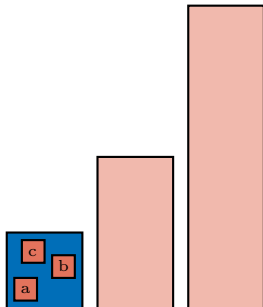


- 12 generations, doubling in size
- Functional mutator: no back pointers

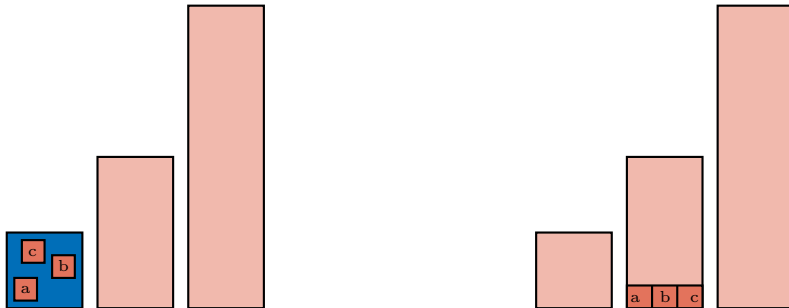
- 12 generations, doubling in size
- Functional mutator: no back pointers
- Cheney's mark-and-copy collects gen to next
- Potentially triggers cascade of pairwise collections

- 12 generations, doubling in size
- Functional mutator: no back pointers
- Cheney's mark-and-copy collects gen to next
- Potentially triggers cascade of pairwise collections
- Three key functions:
  - `forward` copies individual objects
  - `do_scan` repairs copied objects
  - `forward_roots` kick-starts the collection

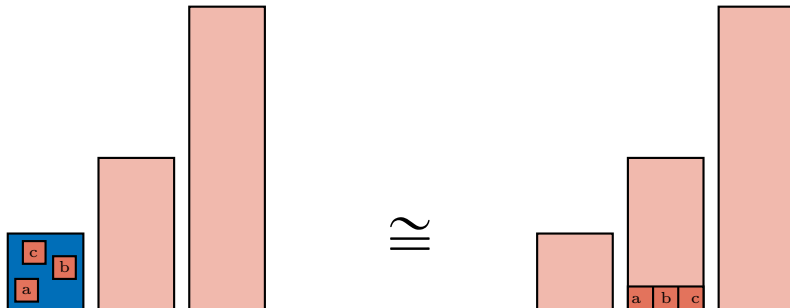
*Primum non nocere*: first, do no harm



*Primum non nocere*: first, do no harm

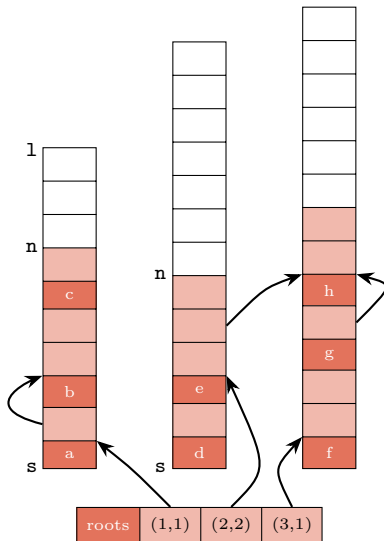


*Primum non nocere*: first, do no harm



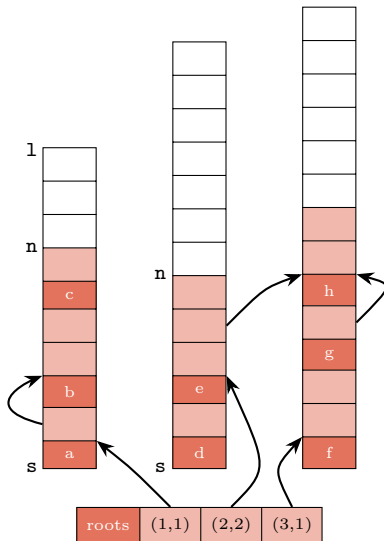
# Overview of Operations

Nursery cannot fit alloc



# Overview of Operations

Nursery cannot fit `alloc`  
`do_gen`



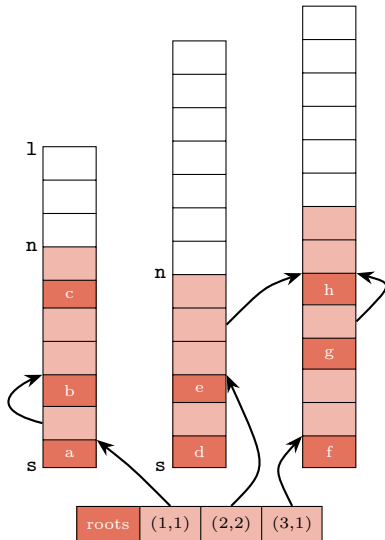


# Overview of Operations

Nursery cannot fit alloc

do\_gen

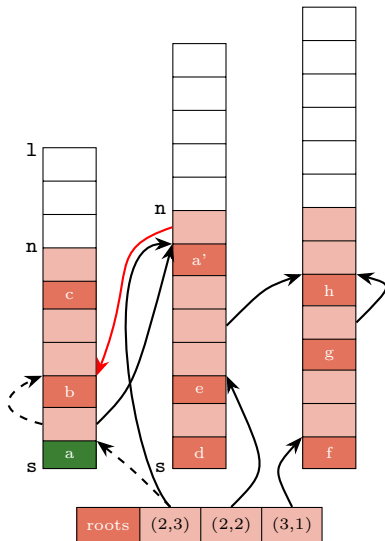
forward\_roots



# Overview of Operations

Nursery cannot fit alloc

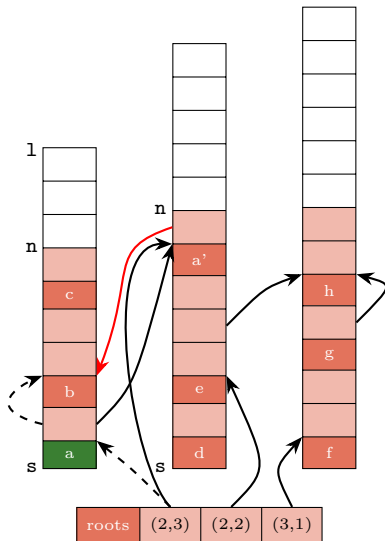
```
do_gen  
  forward_roots  
    forward
```



# Overview of Operations

Nursery cannot fit alloc

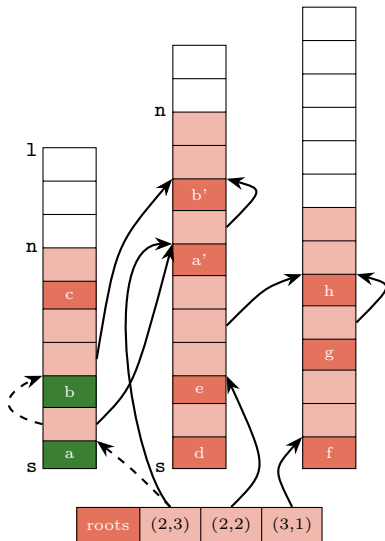
```
do_gen
  forward_roots
    forward
  do_scan
```



# Overview of Operations

Nursery cannot fit alloc

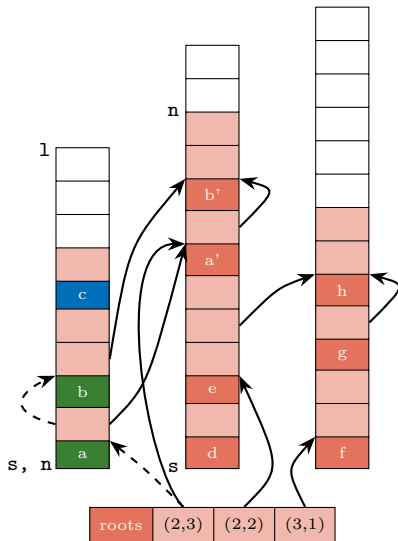
```
do_gen  
  forward_roots  
    forward  
do_scan  
  forward
```



# Overview of Operations

Nursery cannot fit alloc

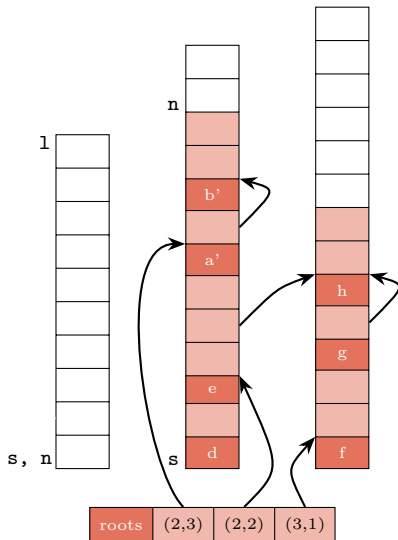
```
do_gen  
  forward_roots  
    forward  
do_scan  
  forward  
reset_gen
```



# Overview of Operations

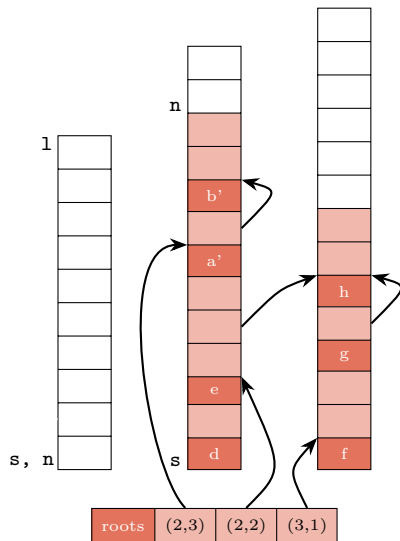
Nursery cannot fit alloc

```
do_gen  
  forward_roots  
    forward  
do_scan  
  forward  
reset_gen
```



# Overview of Operations

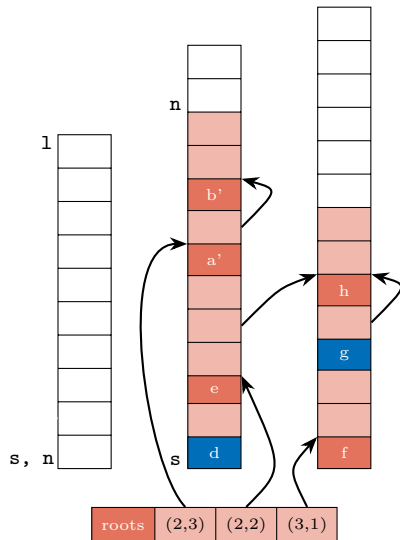
## Non-Concerns



# Overview of Operations

## Non-Concerns

more garbage

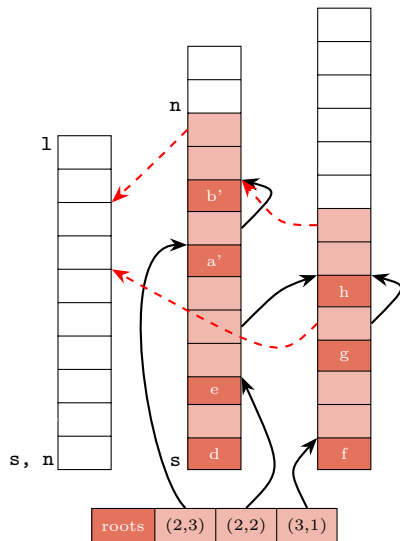




# Overview of Operations

## Non-Concerns

- more garbage
- backward pointers

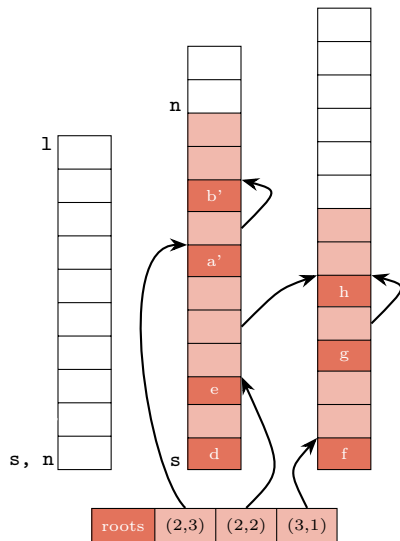


# Overview of Operations

## Non-Concerns

- more garbage
- backward pointers

## Sources of Complexity



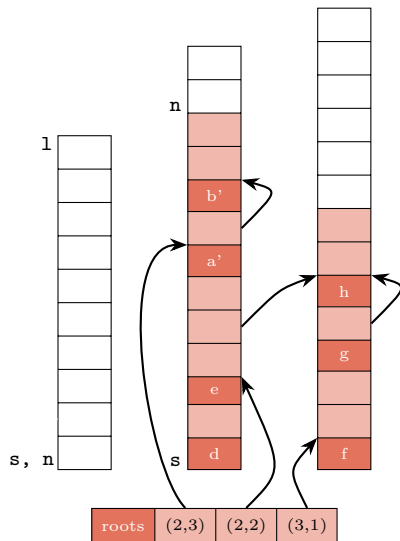
# Overview of Operations

## Non-Concerns

- more garbage
- backward pointers

## Sources of Complexity

- variable-length objects



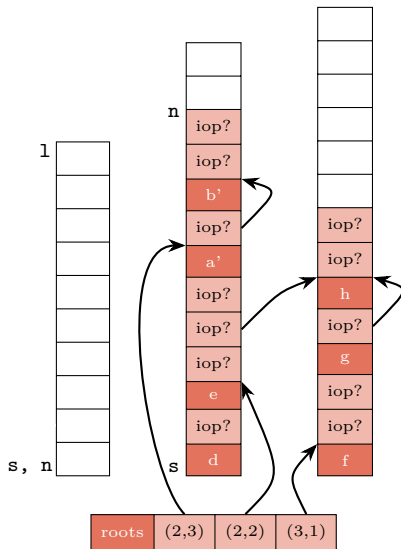
# Overview of Operations

## Non-Concerns

- more garbage
- backward pointers

## Sources of Complexity

- variable-length objects
- disambiguate int/ptr



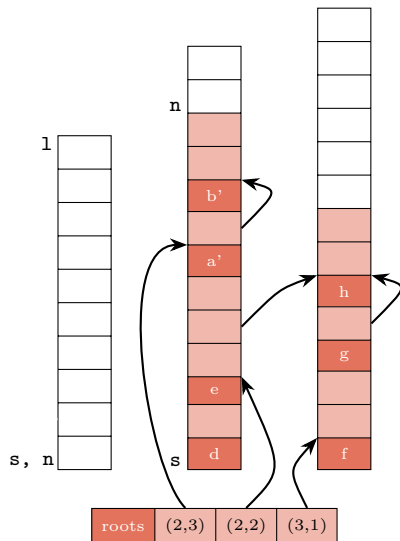
# Overview of Operations

## Non-Concerns

- more garbage
- backward pointers

## Sources of Complexity

- variable-length objects
- disambiguate int/ptr
- determine  $v$ 's gen



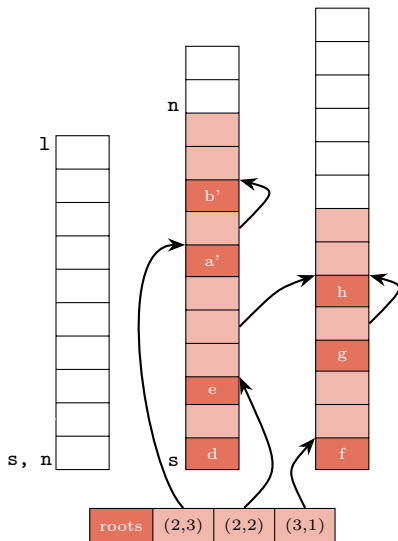
# Overview of Operations

## Non-Concerns

- more garbage
- backward pointers

## Sources of Complexity

- variable-length objects
- disambiguate int/ptr
- determine  $v$ 's gen
- determine gen size



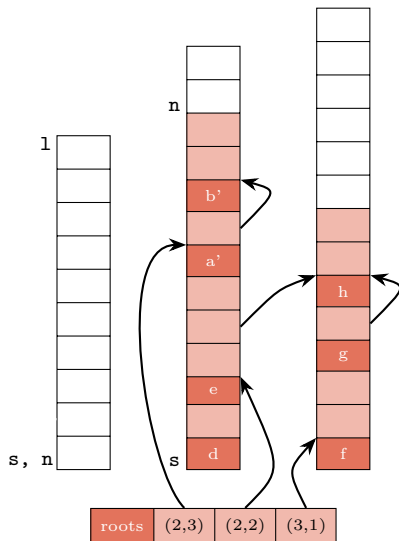
# Overview of Operations

## Non-Concerns

- more garbage
- backward pointers

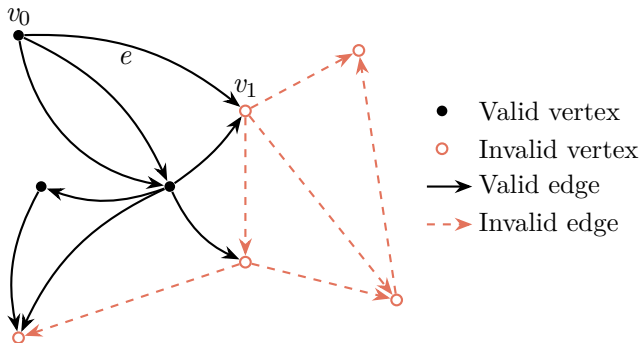
## Sources of Complexity

- variable-length objects
- disambiguate int/ptr
- determine  $v$ 's gen
- determine gen size
- what if `malloc` fails?



# Instantiating GC\_Graph

A PreGraph is a hextuple ( $VType$ ,  $EType$ ,  $vvalid$ ,  $evalid$ ,  $src$ ,  $dst$ )





A PreGraph is a hextuple (VType, EType, vvalid, evalid, src, dst)

**GC\_PreGraph:** VType := nat \* nat  
EType := VType \* nat  
src := fst  
dst := *unrestricted*  
 $\forall v. \text{vvalid}(\gamma, v) \Leftrightarrow \text{graph\_has\_v}(\gamma, v)$   
 $\forall v, out. \text{evalid}(\gamma, (v, out)) \Leftrightarrow$   
 $\text{vvalid}(\gamma, v) \wedge \text{In } out \text{ (get\_edges}(\gamma, v))$

A LabeledGraph is a quadruple (PreGraph, VL, EL, GL)

**GC\_Graph:** GC\_PreGraph as shown

VL := raw\_vert\_block

EL := unit

GL := list gen\_info

# Instantiating GC\_Graph

A LabeledGraph is a quadruple (PreGraph, VL, EL, GL)

**GC\_Graph:** GC\_PreGraph as shown

VL := raw\_vert\_block

EL := unit

GL := list gen\_info

Definition

raw\_fld := Z + GC\_Ptr.

Record raw\_vert\_block :=  
{ raw\_mark: bool;  
 copied\_vertex: VType;  
 raw\_flds: list raw\_fld;  
 (\* elided \*) }.

Record gen\_info :=  
{ s\_addr: val;  
 s\_ok: isptr s\_addr;  
 num\_vert: nat;  
 (\* elided \*) }.

forward is everywhere!

```
void forward (value *s, *l, **n, *p) {  
  value * v; value va = *p;  
  if(Is_block(va)) {  
    v = (value*)iop2ptr(va);  
    if(Is_from(s, l, v)) {  
      header_t hd = Hd_val(v);  
      if(hd == 0) {  
        *p = Field(v,0);  
      } else { /* elided */  

```

forward is everywhere!

forward is **robust**

```
void forward (value *s, *l, **n, *p) {  
  value * v; value va = *p;  
  if(Is_block(va)) {  
    v = (value*)iop2ptr(va);  
    if(Is_from(s, l, v)) {  
      header_t hd = Hd_val(v);  
      if(hd == 0) {  
        *p = Field(v,0);  
      } else { /* elided */  

```

forward is everywhere!

forward is **robust**

pointer?

```
void forward (value *s, *l, **n, *p) {  
  value * v; value va = *p;  
  if(Is_block(va)) {  
    v = (value*)iop2ptr(va);  
    if(Is_from(s, l, v)) {  
      header_t hd = Hd_val(v);  
      if(hd == 0) {  
        *p = Field(v,0);  
      } else { /* elided */  

```

forward is everywhere!

forward is **robust**

pointer?      in from space?

```
void forward (value *s, *l, **n, *p) {  
  value * v; value va = *p;  
  if(Is_block(va)) {  
    v = (value*)iop2ptr(va);  
    if(Is_from(s, l, v)) {  
      header_t hd = Hd_val(v);  
      if(hd == 0) {  
        *p = Field(v,0);  
      } else { /* elided */  

```

forward is everywhere!

forward is **robust**

pointer?      in from space?      already forwarded?

```
void forward (value *s, *l, **n, *p) {  
  value * v; value va = *p;  
  if(Is_block(va)) {  
    v = (value*)iop2ptr(va);  
    if(Is_from(s, l, v)) {  
      header_t hd = Hd_val(v);  
      if(hd == 0) {  
        *p = Field(v,0);  
      } else { /* elided */  

```



## forward: a Deep Dive

forward is everywhere!

forward is **robust**

pointer?      in from space?      already forwarded?  
and **versatile**

```
void forward (value *s, *l, **n, *p) {  
  value * v; value va = *p;  
  if(Is_block(va)) {  
    v = (value*)iop2ptr(va);  
    if(Is_from(s, l, v)) {  
      header_t hd = Hd_val(v);  
      if(hd == 0) {  
        *p = Field(v,0);  
      } else { /* elided */  

```

## forward: a Deep Dive

forward is everywhere!

forward is **robust**

pointer?      in from space?      already forwarded?  
and **versatile**  
called on root set

```
void forward (value *s, *l, **n, *p) {  
  value * v; value va = *p;  
  if(Is_block(va)) {  
    v = (value*)iop2ptr(va);  
    if(Is_from(s, l, v)) {  
      header_t hd = Hd_val(v);  
      if(hd == 0) {  
        *p = Field(v,0);  
      } else { /* elided */  

```

## forward: a Deep Dive

forward is everywhere!

forward is **robust**

pointer?      in from space?      already forwarded?  
and **versatile**  
called on root set      called on heap

```
void forward (value *s, *l, **n, *p) {  
  value * v; value va = *p;  
  if(Is_block(va)) {  
    v = (value*)iop2ptr(va);  
    if(Is_from(s, l, v)) {  
      header_t hd = Hd_val(v);  
      if(hd == 0) {  
        *p = Field(v,0);  
      } else { /* elided */  

```

$$\left\{ \begin{array}{l} \forall \gamma, from, to, v, n. \text{gc\_graph}(\gamma) \wedge \text{compat}(\gamma, from, to) \wedge \\ s = \text{start}(\gamma, from) \wedge l = s + \text{gensz}(\gamma, from) \wedge \\ n = \text{nxtaddr}(to) \wedge p = \text{vaddr}(\gamma, v) + n \end{array} \right\} \stackrel{\text{def}}{=} \phi_1$$

```
void forward (value *s, *l, **n, *p) {  
  /* elided */  
  if(hd == 0) {  
    *p = Field(v,0);  
  }
```

$$\left\{ \begin{array}{l} \forall \gamma, from, to, v, n. \text{gc\_graph}(\gamma) \wedge \text{compat}(\gamma, from, to) \wedge \\ s = \text{start}(\gamma, from) \wedge l = s + \text{gensz}(\gamma, from) \wedge \\ n = \text{nxtaddr}(to) \wedge p = \text{vaddr}(\gamma, v) + n \end{array} \right\} \stackrel{\text{def}}{=} \phi_1$$

```
void forward (value *s, *l, **n, *p) {
```

```
  /* elided */
```

```
  if(hd == 0) {
```

```
    *p = Field(v,0);
```

$$\left\{ \begin{array}{l} \phi_1 \wedge \exists \gamma'. \text{gc\_graph}(\gamma') \wedge \gamma' = \text{upd\_edge}(\gamma, e, \text{copy}(\gamma, v)) \wedge \\ \text{compat}(\gamma', from, to) \wedge \text{fwd\_relation}(\gamma, \gamma', from, to, v, n) \end{array} \right\}$$

```
}
```

```
else {  
  int i; int sz; value *new; sz = size(hd);  
  new = *next+1; *next = new+sz; Hd_val(new) = hd;  
  for(i = 0; i < sz; i++)  
    Field(new, i) = Field(v, i);  
}
```

```

else {
  int i; int sz; value *new; sz = size(hd);
  new = *next+1; *next = new+sz; Hd_val(new) = hd;
  for(i = 0; i < sz; i++)
    Field(new, i) = Field(v, i);

```

$$\left\{ \begin{array}{l} \phi_1 \wedge \exists \gamma', v'. \text{gc\_graph}(\gamma') \wedge v' = \text{copied\_vertex}(\gamma, to) \wedge \\ \gamma' = \text{copy\_vertex}(\gamma, to, v, v') \wedge \text{compat}(\gamma', from, to) \end{array} \right\} \stackrel{\text{def}}{=} \phi_2$$

```

else {
  int i; int sz; value *new; sz = size(hd);
  new = *next+1; *next = new+sz; Hd_val(new) = hd;
  for(i = 0; i < sz; i++)
    Field(new, i) = Field(v, i);
  {
 $\phi_1 \wedge \exists \gamma', v'. \text{gc\_graph}(\gamma') \wedge v' = \text{copied\_vertex}(\gamma, to) \wedge$ 
 $\gamma' = \text{copy\_vertex}(\gamma, to, v, v') \wedge \text{compat}(\gamma', from, to)$ 
 $\Bigg\} \stackrel{\text{def}}{=} \phi_2$ 
  Hd_val(v) = 0; Field(v, 0) = p2iop((void *)new);
  *p = p2iop((void *)new);
}

```



```

else {
  int i; int sz; value *new; sz = size(hd);
  new = *next+1; *next = new+sz; Hd_val(new) = hd;
  for(i = 0; i < sz; i++)
    Field(new, i) = Field(v, i);
  {
 $\phi_1 \wedge \exists \gamma', v'. \text{gc\_graph}(\gamma') \wedge v' = \text{copied\_vertex}(\gamma, to) \wedge$ 
 $\gamma' = \text{copy\_vertex}(\gamma, to, v, v') \wedge \text{compat}(\gamma', from, to)$ 
 $\} \stackrel{\text{def}}{=} \phi_2$ 
  Hd_val(v) = 0; Field(v, 0) = p2iop((void *)new);
  *p = p2iop((void *)new);
  {
 $\phi_2 \wedge \exists \gamma''. \text{gc\_graph}(\gamma'') \wedge \gamma'' = \text{upd\_edge}(\gamma', e, v') \wedge$ 
 $\text{compat}(\gamma'', from, to) \wedge \text{fwd\_relation}(\gamma, \gamma'', from, to, v, n)$ 
 $\}$ 
}

```

Inductive fwd\_relation from to :

forward\_t -> LGraph -> LGraph -> Prop :=

```
Inductive fwd_relation from to :  
  forward_t -> LGraph -> LGraph -> Prop :=  
| fr_v_not_in : forall v g,  
  vgen v <> from ->  
  fwd_relation from to (inl (inr v)) g g
```

```
Inductive fwd_relation from to :  
  forward_t -> LGraph -> LGraph -> Prop :=  
| fr_v_not_in : forall v g,  
  vgen v <> from ->  
  fwd_relation from to (inl (inr v)) g g  
| fr_e_to_fwded : forall e g,  
  vgen (dst g e) = from ->  
  raw_mark (vlabel g (dst g e)) = true ->  
  let new_g := lgraph_gen_dst g e  
    (copied_vertex (vlabel g (dst g e))) in  
  fwd_relation from to (inr e) g new_g
```

```
| fr_e_to_not_fwdded_Sn : forall e g g',  
  vgen (dst g e) = from ->  
  raw_mark (vlabel g (dst g e)) = false ->  
  let new_g :=  
    lgraph_gen_dst (lgraph_copy1v g (dst g e) to)  
      e (copy1v_new_v g to) in  
  fwd_loop from to  
    (make_fields new_g (copy1v_new_v g to)) new_g g' ->  
  fwd_relation from to (inr e) g g'
```

Similar to `forward_relation`, we have

`forward_roots_relation`

`do_scan_relation`

`do_generation_relation`

`garbage_collect_relation`

Similar to `forward_relation`, we have

`forward_roots_relation`

`do_scan_relation`

`do_generation_relation`

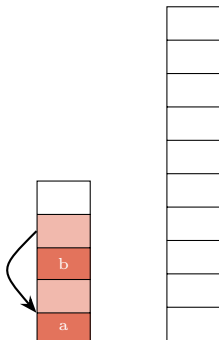
`garbage_collect_relation`

A composition of these gives us our **isomorphism**

# Moving Towards Isomorphism

But the journey is far from easy!

A brief look at `semi_iso`:

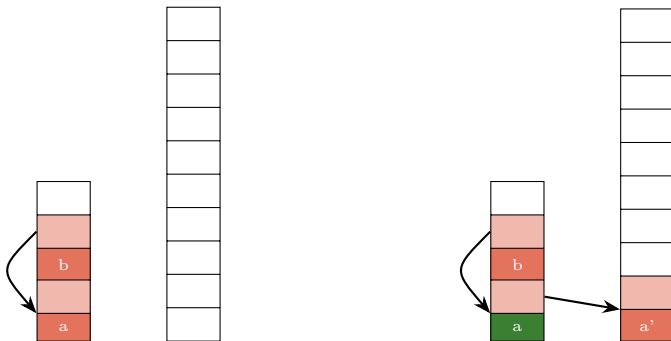




# Moving Towards Isomorphism

But the journey is far from easy!

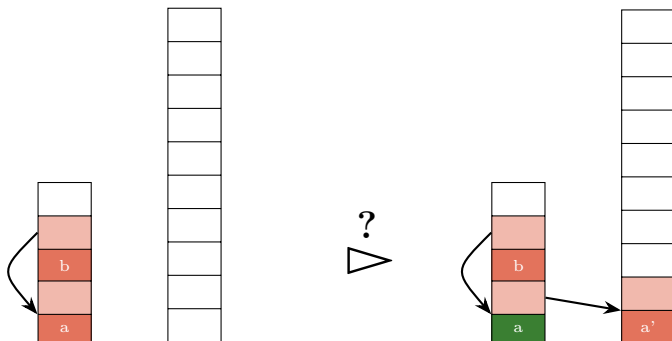
A brief look at `semi_iso`:



# Moving Towards Isomorphism

But the journey is far from easy!

A brief look at `semi_iso`:



The general iterative strategy:

$$\frac{}{\gamma \triangleright \gamma}$$

The general iterative strategy:

$$\frac{}{\gamma \triangleright \gamma} \qquad \frac{\gamma \triangleright \gamma_i \qquad \gamma_i \rightsquigarrow \gamma_{i+1}}{\gamma \triangleright \gamma_{i+1}}$$

The general iterative strategy:

$$\frac{\frac{}{\gamma \triangleright \gamma} \quad \frac{\gamma \triangleright \gamma_i \quad \gamma_i \rightsquigarrow \gamma_{i+1}}{\gamma \triangleright \gamma_{i+1}}}{\gamma_\alpha \triangleright \gamma_\omega}$$

A specific example:

```
Lemma semi_iso_refl: forall g from to,  
  sound_gc_graph g -> semi_iso g g from to nil.
```

A specific example:

```
Lemma semi_iso_refl: forall g from to,  
  sound_gc_graph g -> semi_iso g g from to nil.
```

```
Lemma fwd_rel_semi_iso:  
  forall from to p g1 g2 g3 roots,  
    semi_iso g1 g2 from to l1 ->  
    forward_relation from to p g2 g3 ->  
    semi_iso g1 g3 from to
```

And eventually,

```
Theorem garbage_collect_iso: forall roots1 roots2 g1 g2,  
  ...  
  garbage_collect_relation roots1 roots2 g1 g2 ->  
  gc_graph_iso g1 roots1 g2 roots2.
```



And eventually,

```
Theorem garbage_collect_iso: forall roots1 roots2 g1 g2,  
  ...  
  garbage_collect_relation roots1 roots2 g1 g2 ->  
  gc_graph_iso g1 roots1 g2 roots2.
```

The graphs are isomorphic

up to the vertices reachable from roots

# Moving Towards Isomorphism

And eventually,

```
Theorem garbage_collect_iso: forall roots1 roots2 g1 g2,  
  ...  
  garbage_collect_relation roots1 roots2 g1 g2 ->  
  gc_graph_iso g1 roots1 g2 roots2.
```

The graphs are isomorphic

up to the vertices reachable from roots

The space between `n` and `l` is available for `alloc`

# Moving Towards Isomorphism

And eventually,

```
Theorem garbage_collect_iso: forall roots1 roots2 g1 g2,  
  ...  
  garbage_collect_relation roots1 roots2 g1 g2 ->  
  gc_graph_iso g1 roots1 g2 roots2.
```

The graphs are isomorphic

up to the vertices reachable from roots

The space between `n` and `l` is available for `alloc`

Note that we may still not achieve full isomorphism:

# Moving Towards Isomorphism

And eventually,

```
Theorem garbage_collect_iso: forall roots1 roots2 g1 g2,  
  ...  
  garbage_collect_relation roots1 roots2 g1 g2 ->  
  gc_graph_iso g1 roots1 g2 roots2.
```

The graphs are isomorphic

up to the vertices reachable from roots

The space between `n` and `l` is available for `alloc`

Note that we may still not achieve full isomorphism:

the `graph label` may change to register new vertices

## Moving Towards Isomorphism

And eventually,

```
Theorem garbage_collect_iso: forall roots1 roots2 g1 g2,  
  ...  
  garbage_collect_relation roots1 roots2 g1 g2 ->  
  gc_graph_iso g1 roots1 g2 roots2.
```

The graphs are isomorphic

up to the vertices reachable from roots

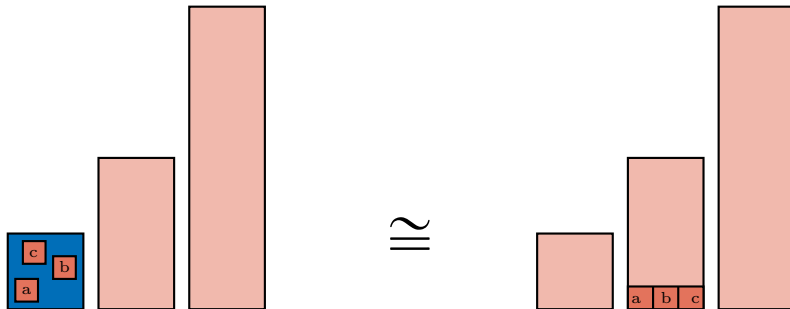
The space between `n` and `l` is available for `alloc`

Note that we may still not achieve full isomorphism:

the **graph label** may change to register new vertices  
and may even grow to accommodate new generations

## Recap: Intuitive Specification

*Primum non nocere*: first, do no harm



- Cheney implemented too conservatively:  
only part of `to` space needs to be  
scanned during `do_scan`

- Cheney implemented too conservatively:  
only part of `to` space needs to be  
scanned during `do_scan`  
Performance doubled



- Cheney implemented too conservatively:  
only part of `to` space needs to be  
scanned during `do_scan`

Performance doubled

- Overflow in the following calculation:

```
int space_size =  
    h->spaces[i].limit - h->spaces[i].start;
```

- Cheney implemented too conservatively:  
only part of `to` space needs to be  
scanned during `do_scan`

Performance doubled

- Overflow in the following calculation:

```
int space_size =  
    h->spaces[i].limit - h->spaces[i].start;  
if difference > 231
```

- Cheney implemented too conservatively:  
only part of `to` space needs to be  
scanned during `do_scan`

Performance doubled

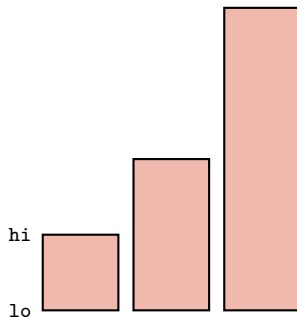
- Overflow in the following calculation:

```
int space_size =  
    h->spaces[i].limit - h->spaces[i].start;  
if difference > 231
```

Fixed by adjusting nursery size

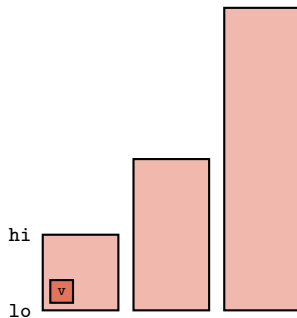
Double-bounded pointer comparisons:

```
int Is_from(value * lo, value * hi, value * v) {  
    return (lo <= v && v < hi); }
```



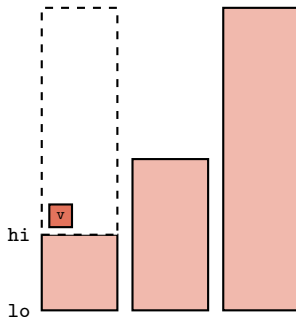
Double-bounded pointer comparisons:

```
int Is_from(value * lo, value * hi, value * v) {  
    return (lo <= v && v < hi); }
```



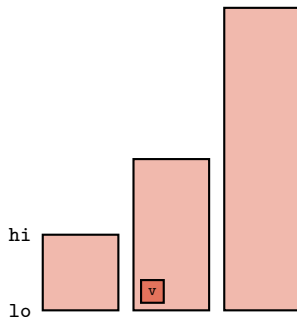
Double-bounded pointer comparisons:

```
int Is_from(value * lo, value * hi, value * v) {  
    return (lo <= v && v < hi);  
}
```



Double-bounded pointer comparisons:

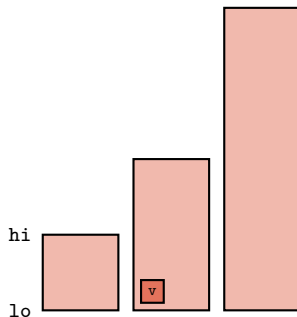
```
int Is_from(value * lo, value * hi, value * v) {  
    return (lo <= v && v < hi); }
```



# Undefined behavior in C

Double-bounded pointer comparisons:

```
int Is_from(value * lo, value * hi, value * v) {  
    return (lo <= v && v < hi); }
```



Resolved using CompCert's `extcall_properties`



A classic OCaml trick to disambiguate int/ptr:

```
int test_int_or_ptr (value x) {  
    return (int)((((intnat)x)&1); }
```

A classic OCaml trick to disambiguate int/ptr:

```
int test_int_or_ptr (value x) {  
    return (int)((((intnat)x)&1); }
```

Essentially, assume that pointers are even-aligned.

A classic OCaml trick to disambiguate int/ptr:

```
int test_int_or_ptr (value x) {  
    return (int)((((intnat)x)&1); }
```

Essentially, assume that pointers are even-aligned.

Consider:

```
void foo() {  
    char a; char b; char* pa = &a; char* pb = &b;  
    if ((pa&1 == 0) && (pb&1 == 0)) { /* elided */ } }
```

A classic OCaml trick to disambiguate int/ptr:

```
int test_int_or_ptr (value x) {  
    return (int)((((intnat)x)&1); }
```

Essentially, assume that pointers are even-aligned.

Consider:

```
void foo() {  
    char a; char b; char* pa = &a; char* pb = &b;  
    if ((pa&1 == 0) && (pb&1 == 0)) { /* elided */ } }
```

True in C, false in exec!

A classic OCaml trick to disambiguate int/ptr:

```
int test_int_or_ptr (value x) {  
    return (int)((((intnat)x)&1); }
```

Essentially, assume that pointers are even-aligned.

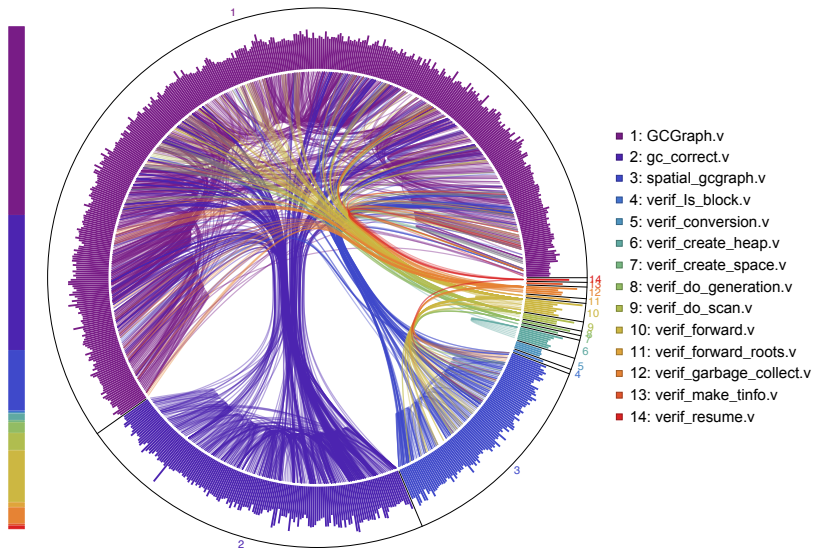
Consider:

```
void foo() {  
    char a; char b; char* pa = &a; char* pb = &b;  
    if ((pa&1 == 0) && (pb&1 == 0)) { /* elided */ } }
```

True in C, false in exec!

Discussing char alignment issues with CompCert

# Reusability: separation between pure and spatial reasoning



Problems of a similar shape

Problems of a **similar shape**  
serialization  
other collectors



Problems of a similar shape  
  serialization  
  other collectors

Towards a verified GC for OCaml

Problems of a **similar shape**

- serialization

- other collectors

Towards a verified GC for **OCaml**

- mutability

- calculate root set

- allow other datatypes

Problems of a **similar shape**  
serialization  
other collectors

Towards a verified GC for **OCaml**  
mutability  
calculate root set  
allow other datatypes

**Further refinements** required in C semantics  
before we can **specify** and **verify** OCaml's GC?