A Verified Garbage Collector for Gallina

Shengyi Wang†, Anshuman Mohan†, Qinxiang Cao‡, Aquinas Hobor†

APLAS NIER
December 1, 2019
Verify **graph-manipulating** programs
written in **executable C**
with **machine-checked** correctness proofs

Hard, but ubiquitous in critical areas
VST + CompCert + 25000 LOC library

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

[Wang et. al., PACMPL OOPSLA 2019]
This Talk

Certified Coq

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
Mutator allocs in special subheap
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 and try again
Our Garbage Collector

- 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:
  - \texttt{forward} copies individual objects
  - \texttt{do\_scan} repairs copied objects
  - \texttt{forward\_roots} kick-starts the collection
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

```
roots | (1,1) | (2,2) | (3,1)
```

Diagram showing the allocation and markings in the nursery and heap.
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_root
  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`

```plaintext
do_gen
  forward_roots
  forward
do_scan
  forward
reset_gen
```
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
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 \((\text{VType}, \text{EType}, \text{vvalid}, \text{evalid}, \text{src}, \text{dst})\)
Instantiating GC\_Graph

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

**GC\_PreGraph:**

\[ VType := \text{nat} \times \text{nat} \]
\[ EType := VType \times \text{nat} \]
\[ src := \text{fst} \]
\[ dst := \text{unrestricted} \]
\[ \forall v.\ vvalid(\gamma, v) \iff \text{graph\_has\_v}(\gamma, v) \]
\[ \forall v, out.\ evalid(\gamma, (v, out)) \iff \]
\[ vvalid(\gamma, v) \land \text{In}\ out\ (\text{get\_edges}(\gamma, v)) \]
Instantiating GC_Graph

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

\[ \text{GC\_Graph: } \text{GC\_PreGraph as shown} \]

\[ \begin{align*}
\text{VL} & := \text{raw\_vert\_block} \\
\text{EL} & := \text{unit} \\
\text{GL} & := \text{list gen\_info}
\end{align*} \]

Definition

\[ \text{raw\_fld} := Z + \text{GC\_Ptr}. \]

Record \text{raw\_vert\_block} :=
\[ \begin{align*}
\{ & \text{raw\_mark: bool;} \\
& \text{copied\_vertex: VType;} \\
& \text{raw\_flds}: \text{list raw\_fld;} \\
& (* \text{elided} *) \}
\end{align*} \]

Record \text{gen\_info} :=
\[ \begin{align*}
& \{ \text{s\_addr: val;} \\
& \text{s\_ok: isptr s\_addr;} \\
& \text{num\_vert: nat;} \\
& (* \text{elided} *) \}\n\end{align*} \]
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 */
        }
    }
}
\[
\begin{align*}
\forall \gamma, \text{from, to, } v, n. \ & \text{gc\_graph}(\gamma) \land \text{compat}(\gamma, \text{from}, \text{to}) \land \\
& s = \text{start}(\gamma, \text{from}) \land l = s + \text{gensz}(\gamma, \text{from}) \land \\
& n = \text{nxtaddr}(\text{to}) \land p = \text{vaddr}(\gamma, v) + n \\
\end{align*}
\]

\[
\left\{ \begin{array}{l}
\phi_1 \\
\forall \gamma'. \ \text{gc\_graph}(\gamma') \land \gamma' = \text{upd\_edge}(\gamma, e, \text{copy}(\gamma, v)) \\
\text{compat}(\gamma', \text{from}, \text{to}) \land \text{fwd\_relation}(\gamma, \gamma', \text{from}, \text{to}, v, n)
\end{array} \right\} \Rightarrow \phi_1
\]

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

	\phi_1 \\

	\forall \gamma'. \ \text{gc\_graph}(\gamma') \land \gamma' = \text{upd\_edge}(\gamma, e, \text{copy}(\gamma, v)) \\
	\text{compat}(\gamma', \text{from}, \text{to}) \land \text{fwd\_relation}(\gamma, \gamma', \text{from}, \text{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);

    \[
    \begin{align*}
    \phi_1 & \land \exists \gamma', v'. \ \text{gc\_graph}(\gamma') \land v' = \text{copied\_vertex}(\gamma, to) \land \\
    \gamma' & = \text{copy\_vertex}(\gamma, to, v, v') \land \text{compat}(\gamma', \text{from}, to) \\
    \end{align*}
    \] \quad \text{def } \phi_2

    Hd_val(v) = 0; Field(v, 0) = p2iop((void *)new);
    *p = p2iop((void *)new);

    \[
    \begin{align*}
    \phi_2 & \land \exists \gamma''. \ \text{gc\_graph}(\gamma'') \land \gamma'' = \text{upd\_edge}(\gamma', e, v') \land \\
    \text{compat}(\gamma'', \text{from}, to) \land \text{fwd\_relation}(\gamma, \gamma'', \text{from}, to, v, n) \\
    \end{align*}
    \]
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_fwded_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

A composition of these gives us our isomorphism.
Moving Towards Isomorphism

But the journey is far from easy!
A brief look at \texttt{semi_iso}:

```
\texttt{\begin{array}{c}
\text{a} \\
\text{b} \\
\end{array}} \quad \text{\texttt{?}} \quad \texttt{\begin{array}{c}
\text{a'} \\
\text{b} \\
\end{array}}
```
The general iterative strategy:

\[
\gamma \triangleright \gamma \\
\gamma \triangleright \gamma_i \quad \gamma_i \rightsquigarrow \gamma_{i+1} \\
\gamma \triangleright \gamma_{i+1} \\
\gamma_\alpha \triangleright \gamma_\omega
\]
A specific example:

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

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**: for all roots1 roots2 g1 g2,

\[
\text{garbage\_collect\_relation \ roots1 \ roots2 \ g1 \ g2} \rightarrow \\
\text{gc\_graph\_iso \ g1 \ roots1 \ g2 \ roots2}.
\]

The graphs are isomorphic

up to the vertices reachable from roots

The space between \text{n} and \text{l} is available for \text{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
Bugs in the source C code

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

- Overflow in the following calculation:  
  
  ```c
  int space_size =
      h->spaces[i].limit - h->spaces[i].start;
  if difference > 2^{31}
  Fixed by adjusting nursery size
  ```
Undefined behavior in C

Double-bounded pointer comparisons:

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

Resolved using CompCert’s `extcall_properties`
Undefined behavior in C

A classic OCaml trick to disambiguate int/ptr:

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

Essentially, assume that pointers are even-aligned.

Consider:

```c
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

1: GCGraph.v
2: gc_correct.v
3: spatial_gcgraph.v
4: verif_is_block.v
5: verif_conversion.v
6: verif_create_heap.v
7: verif_create_space.v
8: verif_do_generation.v
9: verif_do_scan.v
10: verif_forward.v
11: verif_forward_roots.v
12: verif_garbage_collect.v
13: verif_make_tinfo.v
14: verif_resume.v
Future Work

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?