Mechanized Verification of the Correctness and Asymptotic Complexity of Programs

Armaël Guéneau

under the supervision of Arthur Charguéraud and François Pottier

Computer programs: cooking recipes, but for computers?

Mom's easy apple pie

- Slice 6 apples
- Mix with 3/4C sugar, 2T flour, 3/4T cinnamon, 1T lemon juice
- Transfer between two pie crusts
- Bake 40 min at 425°F

Computing the lengths of two lists

```
let length_sum l1 l2 =
  let x = length l1 in
  let y = length l2 in
  x + y
```

Computer programs: cooking recipes, but for computers?

```c
static __latent_entropy int dup_mmap(struct mm_struct *mm,
                                        struct mm_struct *oldmm)
{
        struct vm_area_struct *mpnt, *tmp, *prev, **pprev;
        struct rb_node **rb_link, *rb_parent;
        int retval;
        unsigned long charge;
        LIST_HEAD(uf);

        uprobe_start_dup_mmap();
        if (down_write_killable(&oldmm->mmap_sem)) {
                retval = -EINTR;
                goto fail_uprobe_end;
        }
        flush_cache_dup_mm(oldmm);
        uprobe_dup_mmap(oldmm, mm);
        /*
         * Not linked in yet - no deadlock potential:
         */
        down_write_nested(&mm->mmap_sem, SINGLE_DEPTH_NESTING);

        /* No ordering required: file already has been exposed. */
        RCU_INIT_POINTER(mm->exe_file, get_mm_exe_file(oldmm));

        mm->total_vm = oldmm->total_vm;
        mm->data_vm = oldmm->data_vm;
        mm->exec_vm = oldmm->exec_vm;
        mm->stack_vm = oldmm->stack_vm;

        rb_link = &mm->mm_rb.rb_node;
        rb_parent = NULL;
        pprev = &mm->mmap;
        retval = ksm_fork(mm, oldmm);
        if (retval)
                goto out;
        retval = khugepaged_fork(mm, oldmm);
        if (retval)
                goto out;

        prev = NULL;
        for (mpnt = oldmm->mmap; mpnt; mpnt = mpnt->vm_next) {
                struct file *file;

                if (mpnt->vm_flags & VM_DONTCOPY) {
                        vm_stat_account(mm, mpnt->vm_flags, -vma_pages
                        continue;
```

```c
if (mpnt->vm_flags & VM_ACCOUNT) {
        unsigned long len = vma_pages(mpnt);

        if (security_vm_enough_memory_mm(oldmm, len)) /*
                goto fail_nomem;
        charge = len;
}
tmp = vm_area_dup(mpnt);
if (!tmp)
        goto fail_nomem;
retval = vma_dup_policy(mpnt, tmp);
if (retval)
        goto fail_nomem_policy;
tmp->vm_mm = mm;
retval = dup_userfaultfd(tmp, &uf);
if (retval)
        goto fail_nomem_anon_vma_fork;
if (tmp->vm_flags & VM_WIPEONFORK) {
        /* VM_WIPEONFORK gets a clean slate in the child
        tmp->anon_vma = NULL;
        if (anon_vma_prepare(tmp))
                goto fail_nomem_anon_vma_fork;
} else if (anon_vma_fork(tmp, mpnt))
        goto fail_nomem_anon_vma_fork;
tmp->vm_flags &= ~(VM_LOCKED | VM_LOCKONFAULT);
tmp->vm_next = tmp->vm_prev = NULL;
file = tmp->vm_file;
if (file) {
        struct inode *inode = file_inode(file);
        struct address_space *mapping = file->f_mapping;

        get_file(file);
        if (tmp->vm_flags & VM_DENYWRITE)
                atomic_dec(&inode->i_writecount);
        i_mmap_lock_write(mapping);
        if (tmp->vm_flags & VM_SHARED)
                atomic_inc(&mapping->i_mmap_writable);
        flush_dcache_mmap_lock(mapping);
        /* insert tmp into the share list, just after mp
        vma_interval_tree_insert_after(tmp, mpnt,
                        &mapping->i_mmap);
        flush_dcache_mmap_unlock(mapping);
        i_mmap_unlock_write(mapping);
}

/*
 * Clear hugetlb-related page reserves for children. Thi
 * affects MAP_PRIVATE mappings. Faults generated by the
 * are not guaranteed to succeed, even if read-only
 */
```

```ocaml
and eqappr cv_pb l2r infos (lft1,st1) (lft2,st2) cuniv =
  Control.check_for_interrupt ();
  (* First head reduce both terms *)
  let ninfos = infos_with_reds infos.cnv_inf betaiotazeta in
  let (hd1, v1 as appr1) = whd_stack ninfos infos.lft_tab (fst st1)
  let (hd2, v2 as appr2) = whd_stack ninfos infos.rgt_tab (fst st2)
  let appr1 = (lft1, appr1) and appr2 = (lft2, appr2) in
  (** We delay the computation of the lifts that apply to the head o
      with [el_stack] inside the branches where they are actually us
  match (fterm_of hd1, fterm_of hd2) with
    (* case of leaves *)
    | (FAtom a1, FAtom a2) ->
      (match kind a1, kind a2 with
        | (Sort s1, Sort s2) ->
          if not (is_empty_stack v1 && is_empty_stack v2) then
            anomaly (Pp.str "conversion was given ill-typed ter
          sort_cmp_universes (info_env infos.cnv_inf) cv_pb s1 s
        | (Meta n, Meta m) ->
          if Int.equal n m
          then convert_stacks l2r infos lft1 lft2 v1 v2 cuniv
          else raise NotConvertible
        | _ -> raise NotConvertible)
    | (FEvar ((ev1,args1),env1), FEvar ((ev2,args2),env2)) ->
      if Evar.equal ev1 ev2 then
        let el1 = el_stack lft1 v1 in
        let el2 = el_stack lft2 v2 in
        let cuniv = convert_stacks l2r infos lft1 lft2 v1 v2 cuniv
        convert_vect l2r infos el1 el2
          (Array.map (mk_clos env1) args1)
          (Array.map (mk_clos env2) args2) cuniv
      else raise NotConvertible

    (* 2 index known to be bound to no constant *)
    | (FRel n, FRel m) ->
      let el1 = el_stack lft1 v1 in
      let el2 = el_stack lft2 v2 in
      if Int.equal (reloc_rel n el1) (reloc_rel m el2)
      then convert_stacks l2r infos lft1 lft2 v1 v2 cuniv
      else raise NotConvertible

    (* 2 constants, 2 local defined vars or 2 defined rels *)
    | (FFlex fl1, FFlex fl2) ->
      (try
         let cuniv = conv_table_key infos.cnv_inf fl1 fl2 cuniv in
         convert_stacks l2r infos lft1 lft2 v1 v2 cuniv
       with NotConvertible | Univ.UniverseInconsistency _ ->
       (* else the oracle tells which constant is to be expanded *
```

Computer: cooking recipes, but for computers? (2)

Real-world programs are usually very large.

Computer: cooking recipes, but for computers? (2)

Real-world programs are usually very large.

Can one trust the execution of that code to “do the right thing”?

Computer: cooking recipes, but for computers? (2)

Real-world programs are usually very large.

Can one trust the execution of that code to “do the right thing”?

What does it mean to do the right thing?

Computer: cooking recipes, but for computers? (2)

Real-world programs are usually very large.

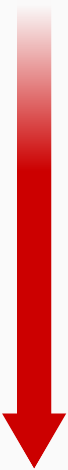
Can one trust the execution of that code to “do the right thing”?

What does it mean to do the right thing?

“The right thing”: a **specification**, written in a formal language.

What do we expect from a program?

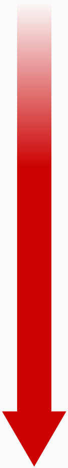
less
confidence



more
confidence

What do we expect from a program?

less
confidence



more
confidence

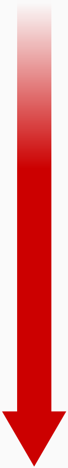
Safety (does not crash)

What do we expect from a program?

less
confidence

Safety (does not crash)

Partial correctness (returns a correct result; might not terminate)



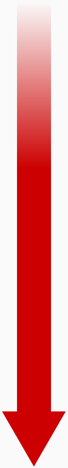
more
confidence

What do we expect from a program?

Safety	(does not crash)
Partial correctness	(returns a correct result; might not terminate)
Total correctness	(always returns a correct result)

What do we expect from a program?

less confidence



more confidence

Safety	(does not crash)
Partial correctness	(returns a correct result; might not terminate)
Total correctness	(always returns a correct result)
Complexity bound	(runs in a predictable amount of time)

What do we expect from a program?

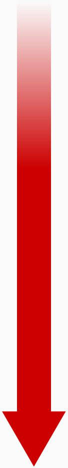
less confidence

Safety	(does not crash)
Partial correctness	(returns a correct result; might not terminate)
Total correctness	(always returns a correct result)
Complexity bound	(runs in a predictable amount of time)
Real-time bound	(runs within a precise time budget)

more confidence

What do we expect from a program?

less confidence



more confidence

Safety	(does not crash)
Partial correctness	(returns a correct result; might not terminate)
Total correctness	(always returns a correct result)
Complexity bound	(runs in a predictable amount of time)
Real-time bound	(runs within a precise time budget)
Security	(e.g. timing side channel)

What do we expect from a program?

less confidence

Safety	(does not crash)
Partial correctness	(returns a correct result; might not terminate)
Total correctness	(always returns a correct result)
Complexity bound	(runs in a predictable amount of time)
Real-time bound	(runs within a precise time budget)
Security	(e.g. timing side channel)
Fault tolerant	(resists to hardware faults)

more confidence

What do we expect from a program?

less confidence

in this work

Safety	(does not crash)
Partial correctness	(returns a correct result; might not terminate)
Total correctness	(always returns a correct result)
Complexity bound	(runs in a predictable amount of time)
Real-time bound	(runs within a precise time budget)
Security	(e.g. timing side channel)
Fault tolerant	(resists to hardware faults)

more confidence

An illustrative example: Binary Search

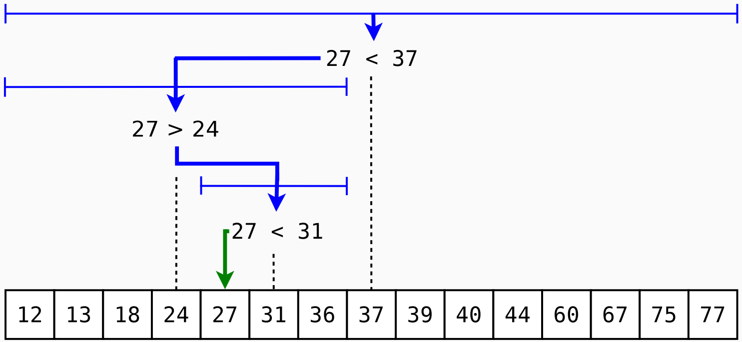
Consider a *sorted* array of integers:

12	13	18	24	27	31	36	37	39	40	44	60	67	75	77

Question: is 27 in the array? If so, at which index?

An illustrative example: Binary Search (2)

At each step, reduce by half the segment to search by comparing 27 with the middle element.



A tentative binary search implementation

```
(* search in array a for x, in the range [i, j) *)
(* returns the index of x, or -1 if not found *)
let rec bsearch (a: int array) x i j =
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = a.(k) then k
    else if x < a.(k) then bsearch a x i k
    else bsearch a x (i+1) j
```

- We can test this program on example input data
- We can formally prove its (total) functional correctness

A tentative binary search implementation

```
(* search in array a for x, in the range [i, j) *)
(* returns the index of x, or -1 if not found *)
let rec bsearch (a: int array) x i j =
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = a.(k) then k
    else if x < a.(k) then bsearch a x i k
    else bsearch a x (i+1) j
```

- We can test this program on example input data
- We can formally prove its (total) functional correctness
- Yet, something is wrong...

A tentative binary search implementation (2)

On an array containaing 1 billion elements:

- A correct binary search should do at most 30 recursive calls ($2^{30} \simeq 1$ billion)
- On some inputs, the code shown performs 1 billion recursive calls

A tentative binary search implementation (3)

```
(* search in array a for x, in the range [i, j) *)
(* returns the index of x, or -1 if not found *)
let rec bsearch (a: int array) x i j =
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = a.(k) then k
    else if x < a.(k) then bsearch a x i k
    else bsearch a x (i+1) j
```

buggy, should be k+1

A tentative binary search implementation (4)

In summary, on an array of size n:

- We expect $O(\log n)$ recursive calls;
- But our program does up to n recursive calls.

A tentative binary search implementation (4)

In summary, on an array of size n:

- We expect $O(\log n)$ recursive calls;
- But our program does up to n recursive calls.

less confidence

in this work

Safety	(does not crash)
Partial correctness	(returns a correct result; might not terminate)
Total correctness	(always returns a correct result)
Complexity bound	**(runs in a predictable amount of time)**

Formal verification of correctness and complexity of a program

Step 1

State a **program specification** that characterizes the intended behavior: functional correctness **and** runtime complexity

Formal verification of correctness and complexity of a program

Step 1

State a **program specification** that characterizes the intended behavior: functional correctness **and** runtime complexity

Step 2

Prove a theorem relating concrete code to the specification

Formal verification of correctness and complexity of a program

Step 1

State a **program specification** that characterizes the intended behavior: functional correctness **and** runtime complexity

Step 2

Prove a theorem relating concrete code to the specification

Two kinds of possible human mistakes:

- in math results used in the analysis; or
- when relating the concrete code to the abstract algorithm

Use a **proof assistant** (Coq) to mechanically check every step of the proof

How do we specify a program's running time?

Option 1: as an upper bound on the wall-clock time.

Useful for embedded systems, but not realistic for commodity hardware.

How do we specify a program's running time?

Option 1: as an upper bound on the wall-clock time.

Useful for embedded systems, but not realistic for commodity hardware.

Option 2: as a number of cycles for an idealized machine model.

Knuth:

"Merge sort runs in $10N \log N + 4.92N$. [This bound] can be reduced to $9N \log N$ at the expense of a somewhat longer program."

How do we specify a program's running time?

Option 1: as an upper bound on the wall-clock time.

Useful for embedded systems, but not realistic for commodity hardware.

Option 2: as a number of cycles for an idealized machine model.

Knuth:

"Merge sort runs in $10N \log N + 4.92N$. [This bound] can be reduced to $9N \log N$ at the expense of a somewhat longer program."

Option 3: as a number of function calls in a high-level language.

More abstract, but still has modularity issues.

How do we specify a program's running time?

Option 4: specify the running time using **asymptotic** complexity.

Describe the "order of growth" of the running time as inputs grow large
e.g. $O(\log n), O(n), O(n \log n), O(n^2)$, ...

Less precise, but informative enough in many cases.

Advantages of asymptotic complexity specifications

Specifications capturing asymptotic costs:

- have been **widely applied** to a large class of programs and algorithms;
- are **independent** of the machine, runtime system and the details of the implementation;
- allow **modular reasoning**. Abstract over implementation details.

In this thesis

Goal: specify and prove that programs compute a correct result with a bounded asymptotic runtime.

Proofs should be:

- static;
- machine-checked;
- hardware- and runtime- independent;
- modular.

In this thesis

Goal: specify and prove that programs compute a correct result with a bounded asymptotic runtime.

Proofs should be:

- static;
- machine-checked;
- hardware- and runtime- independent;
- modular.

Contribution:

A step forward for the verification of the **correctness and complexity** of **imperative, higher-order** programs with **subtle invariants and analysis**, at a **reasonable cost**.

Details of the contribution

1. A formal account of *O*()

Existing:
single-variate O (math, programs), multi-variate O on paper

Contributed:
Coq library for single and multi-variate O,
with lemmas useful for program analysis

Contributions

2. A methodology for complexity proofs

Existing:

- manual verification without $O()$ abstraction
- automated analysis restricted to polynomial bounds

Contributed:

- general asymptotic bounds
- with semi-automated cost inference
- implemented as an extension of CFML
  (Separation Logic framework in Coq)

Contributions

3. Case studies

Existing:
polynomial or logarithmic bounds, simple algorithms (quicksort), or interactive verification without O

Contributed:
several algorithms, including a state-of-the-art graph algorithm with nontrivial correctness and complexity

Outline of the rest of the talk

Reasoning with abstract cost functions

Semi-automatic inference of cost functions

Separation Logic with Time Credits

Case study—an Incremental Cycle Detection Algorithm

Reasoning with abstract cost functions

Informal reasoning principles on O can be abused

```
let rec bsearch a x i j =
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = a.(k) then k
    else if x < a.(k) then
      bsearch a x i k
    else
      bsearch a x (k+1) j
```

Claim:

`bsearch a x i j` costs $O(1)$.

Informal reasoning principles on O can be abused

```
let rec bsearch a x i j =
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = a.(k) then k
    else if x < a.(k) then
      bsearch a x i k
    else
      bsearch a x (k+1) j
```

Claim:

`bsearch a x i j` costs $O(1)$.

Proof:

By induction on $j - i$:

Informal reasoning principles on O can be abused

```
let rec bsearch a x i j =
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = a.(k) then k
    else if x < a.(k) then
      bsearch a x i k
    else
      bsearch a x (k+1) j
```

Claim:

`bsearch a x i j` costs $O(1)$.

Proof:

By induction on $j - i$:

- $j - i \leqslant 0$: $\quad O(1)$.

Informal reasoning principles on O can be abused

```
let rec bsearch a x i j =
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = a.(k) then k
    else if x < a.(k) then
      bsearch a x i k
    else
      bsearch a x (k+1) j
```

Claim:

`bsearch a x i j` costs $O(1)$.

Proof:

By induction on $j - i$:

- $j - i \leqslant 0$: $\quad O(1)$.
- $j - i > 0$: $\quad O(1) + O(1) + O(1) = O(1)$.

Informal reasoning principles on O can be abused

```
let rec bsearch a x i j =
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = a.(k) then k
    else if x < a.(k) then
      bsearch a x i k
    else
      bsearch a x (k+1) j
```

Claim:

`bsearch a x i j` costs $O(1)$.

Proof:

By induction on $j - i$:

- $j - i \leqslant 0$: $\quad O(1)$.
- $j - i > 0$: $\quad O(1) + O(1) + O(1) = O(1)$.

Where is the catch?

Informal reasoning principles on O can be abused

```
let rec bsearch a x i j =
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = a.(k) then k
    else if x < a.(k) then
      bsearch a x i k
    else
      bsearch a x (k+1) j
```

Claim:

`bsearch a x i j` costs $O(1)$.

Proof:

$\boxed{\text{By induction on } j - i\text{:}}$ ...but which statement are we proving?

- $j - i \leqslant 0$: $\quad O(1)$.
- $j - i > 0$: $\quad O(1) + O(1) + O(1) = O(1)$.

Meaning of $O(1)$

What we just proved:

$\forall i\, j$, $\exists\, c$, "`bsearch a x i j`" performs at most c function calls

Meaning of $O(1)$

What we just proved:

$\forall i\, j$, $\exists\, c$, "`bsearch a x i j`" performs at most c function calls

What "$O(1)$" means:

$\exists\, c$, $\forall i\, j$, "`bsearch a x i j`" performs at most c function calls

Meaning of $O(\log n)$

Informal specification: "`bsearch a x i j`" runs in $O(\log(j-i))$.

Meaning of $O(\log n)$

Informal specification: "`bsearch a x i j`" runs in $O(\log(j - i))$.

Meaning: there exists a **cost function** f such that,

- for every `a`, `x`, `i`, `j`, "`bsearch a x i j`" performs at most $f(j - i)$ function calls
- $f \in O(\lambda n.\ \log n)$.

Construction of the cost function

Option 1: The user somehow guesses a suitable cost function.
Here, "$\lambda n.\ 3 \log n + 4$" works.

Construction of the cost function

Option 1: The user somehow guesses a suitable cost function.
Here, "$\lambda n.\ 3 \log n + 4$" works.

Option 3: The cost function is automatically inferred by some clever algorithm... Restricted to specific classes of programs.

Construction of the cost function

Option 1: The user somehow guesses a suitable cost function.
Here, "$\lambda n.\ 3 \log n + 4$" works.

Option 2: Semi-automatically construct the cost function as the proof progresses.

Option 3: The cost function is automatically inferred by some clever algorithm... Restricted to specific classes of programs.

Semi-automatic synthesis of cost functions

Our approach to this problem

Part 1:

- Synthesize a cost function with the same structure as the code
- For recursive functions, recurrence equations are synthesized
- Accounting details are automatically synthesized
- User input is requested when some over-approximation is required

Part 2:

- In a second step, prove a $O()$ bound for the inferred cost function

Constraint inferred on the cost function f

```
let rec bsearch a x i j =
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = Array.get a k then k
    else if x < Array.get a k
      then bsearch a x i k
      else bsearch a x (k+1) j
```

```
f n >= 1 + (                              where n = j-i
         if n <= 0 then 0 else
           0 + 1 + max 0 (
             1 + max (f (n/2))
                     (f (n - n/2 - 1))
           )
       )
```

Interactive construction of the cost function f

```
if j <= i then -1 else
  let k = i + (j - i) / 2 in
  if x = Array.get a k then k
  else if x < Array.get a k
    then bsearch a x i k
    else bsearch a x (k+1) j
```

```
f (j-i) >= 1 + …
```

a hole ("…") is implemented as an evar in Coq

Interactive construction of the cost function f

```
if j <= i then -1 else
  let k = i + (j - i) / 2 in
  if x = Array.get a k then k
  else if x < Array.get a k
    then bsearch a x i k
    else bsearch a x (k+1) j
```

```
f (j-i) >= 1 + (if j <= i then … else …)
```

Interactive construction of the cost function f

```
if j <= i then -1 else
  let k = i + (j - i) / 2 in
  if x = Array.get a k then k
  else if x < Array.get a k
    then bsearch a x i k
    else bsearch a x (k+1) j
```

---

f (j-i) >= 1 + (if j ~~<=~~ i then … else …)

Interactive construction of the cost function f

```
if j <= i then -1 else
  let k = i + (j - i) / 2 in
  if x = Array.get a k then k
  else if x < Array.get a k
    then bsearch a x i k
    else bsearch a x (k+1) j
```

```
f (j-i) >= 1 + (if (j-i) <= 0 then … else …)
```

Interactive construction of the cost function f

```
if j <= i then -1 else
  let k = i + (j - i) / 2 in
  if x = Array.get a k then k
  else if x < Array.get a k
    then bsearch a x i k
    else bsearch a x (k+1) j
```

```
f (j-i) >= 1 + (if (j-i) <= 0 then 0 else …)
```

Interactive construction of the cost function f

```
if j <= i then -1 else
  let k = i + (j - i) / 2 in
  if x = Array.get a k then k
  else if x < Array.get a k
    then bsearch a x i k
    else bsearch a x (k+1) j
```

```
f (j-i) >= 1 + (
             if (j-i) <= 0 then 0 else
               0 + …
           )
```

Interactive construction of the cost function f

```
if j <= i then -1 else
  let k = i + (j - i) / 2 in
  if x = Array.get a k then k
  else if x < Array.get a k
    then bsearch a x i k
    else bsearch a x (k+1) j
```

```
f (j-i) >= 1 + (
             if (j-i) <= 0 then 0 else
               0 + 1 + …
           )
```

Interactive construction of the cost function f

```
if j <= i then -1 else
  let k = i + (j - i) / 2 in
  if x = Array.get a k then k
  else if x < Array.get a k
    then bsearch a x i k
    else bsearch a x (k+1) j
```

```
f (j-i) >= 1 + (
             if (j-i) <= 0 then 0 else
               0 + 1 + max … …
           )
```

Interactive construction of the cost function f

```
if j <= i then -1 else
  let k = i + (j - i) / 2 in
  if x = Array.get a k then k
  else if x < Array.get a k
    then bsearch a x i k
    else bsearch a x (k+1) j
```

```
f (j-i) >= 1 + (
             if (j-i) <= 0 then 0 else
               0 + 1 + max 0 …
           )
```

Interactive construction of the cost function f

```
if j <= i then -1 else
  let k = i + (j - i) / 2 in
  if x = Array.get a k then k
  else if x < Array.get a k
    then bsearch a x i k
    else bsearch a x (k+1) j
```

```
f (j-i) >= 1 + (
                 if (j-i) <= 0 then 0 else
                   0 + 1 + max 0 (1 + …)
               )
```

Interactive construction of the cost function f

```
if j <= i then -1 else
  let k = i + (j - i) / 2 in
  if x = Array.get a k then k
  else if x < Array.get a k
    then bsearch a x i k
    else bsearch a x (k+1) j
```

```
f (j-i) >= 1 + (
            if (j-i) <= 0 then 0 else
              0 + 1 + max 0 (1 + max … …)
          )
```

Interactive construction of the cost function f

```
if j <= i then -1 else
  let k = i + (j - i) / 2 in
  if x = Array.get a k then k
  else if x < Array.get a k
    then bsearch a x i k
    else bsearch a x (k+1) j
```

```
f (j-i) >= 1 + (
              if (j-i) <= 0 then 0 else
                0 + 1 + max 0 (
                  1 + max (f ((j-i)/2)) …
                )
            )
```

Interactive construction of the cost function f

```
if j <= i then -1 else
  let k = i + (j - i) / 2 in
  if x = Array.get a k then k
  else if x < Array.get a k
    then bsearch a x i k
    else bsearch a x (k+1) j
```

```
f (j-i) >= 1 + (
             if (j-i) <= 0 then 0 else
               0 + 1 + max 0 (
                 1 + max (f ((j-i)/2))
                         (f ((j-i) - (j-i)/2 - 1))
               )
           )
```

Interactive construction of the cost function f

```
if j <= i then -1 else
  let k = i + (j - i) / 2 in
  if x = Array.get a k then k
  else if x < Array.get a k
    then bsearch a x i k
    else bsearch a x (k+1) j
```

```
f n     >= 1 + (
             if n <= 0 then 0 else
               0 + 1 + max 0 (
                 1 + max (f (n/2))
                         (f (n - n/2 - 1))
               )
           )
```

From cost equation to asymptotic bound

For `bsearch`, there remains to find a $f \in O(\lambda n.\ \log n)$ such that:

$$\forall n.\ f(n)\ \geqslant\ 1 + \begin{cases} 0 \quad \text{if } n \leqslant 0 \\ 1 + \max(0, 1 + \max(f(\frac{n}{2}), f(n - \frac{n}{2} - 1))) \end{cases}$$

From cost equation to asymptotic bound

For `bsearch`, there remains to find a $f \in O(\lambda n.\ \log n)$ such that:

$$\forall n.\ f(n)\ \geqslant\ 1 + \begin{cases} 0 & \text{if } n \leqslant 0 \\ 1 + \max(0, 1 + \max(f(\frac{n}{2}), f(n - \frac{n}{2} - 1))) \end{cases}$$

- Use the “Master Theorem”, when applicable
  (available in Isabelle/HOL, not yet in Coq)
- Substitution method: guess that there is a solution of the form $a \log n + b$, inject it and resolve.

The substitution method in action

$$\exists f : \mathbb{Z} \to \mathbb{Z}.$$

$$\forall n.\ f(n) \geqslant 1 + \begin{cases} 0 & \text{if } n \leqslant 0 \\ 1 + \max(0, 1 + \max(f(\frac{n}{2}), f(n - \frac{n}{2} - 1))) \end{cases}$$

$$\wedge\ f \in O(\lambda n.\ \log n)$$

The substitution method in action

$$\exists f : \mathbb{Z} \to \mathbb{Z}.$$

$$\begin{aligned}
&\quad \text{monotonic } f \\
&\wedge\ \forall n.\ f(n) \geqslant 0 \\
&\wedge\ \forall n.\ n \leqslant 0 \implies f(n) \geqslant 1 \\
&\wedge\ \forall n.\ n \geqslant 1 \implies f(n) \geqslant f(\tfrac{n}{2}) + 3 \\
&\wedge\ f \in O(\lambda n.\ \log n)
\end{aligned}$$

The substitution method in action

$$
\begin{aligned}
\exists a\ b : \mathbb{Z}.& \\
&\quad f(n) = a \log n + b \\
&\wedge \text{monotonic } f \\
&\wedge \forall n.\ f(n) \geqslant 0 \\
&\wedge \forall n.\ n \leqslant 0 \implies f(n) \geqslant 1 \\
&\wedge \forall n.\ n \geqslant 1 \implies f(n) \geqslant f(\tfrac{n}{2}) + 3 \\
&\wedge f \in O(\lambda n.\ \log n)
\end{aligned}
$$

The substitution method in action

$$
\begin{aligned}
&\exists a\ b : \mathbb{Z}.\\
&\qquad f(n) = a \log n + b \quad \text{(issue when } n = 0\text{)}\\
&\quad \wedge\ \text{monotonic } f\\
&\quad \wedge\ \forall n.\ f(n) \geqslant 0\\
&\quad \wedge\ \forall n.\ n \leqslant 0 \implies f(n) \geqslant 1\\
&\quad \wedge\ \forall n.\ n \geqslant 1 \implies f(n) \geqslant f(\tfrac{n}{2}) + 3\\
&\quad \wedge\ f \in O(\lambda n.\ \log n)
\end{aligned}
$$

The substitution method in action

$$
\begin{aligned}
\exists a\ b\ c : \mathbb{Z}.& \\
& f(n) = \text{if } n > 0 \text{ then } a \log n + b \text{ else } c \\
\wedge\ & \text{monotonic } f \\
\wedge\ & \forall n.\ f(n) \geqslant 0 \\
\wedge\ & \forall n.\ n \leqslant 0 \implies f(n) \geqslant 1 \\
\wedge\ & \forall n.\ n \geqslant 1 \implies f(n) \geqslant f(\tfrac{n}{2}) + 3 \\
\wedge\ & f \in O(\lambda n.\ \log n)
\end{aligned}
$$

The substitution method in action

$$
\begin{aligned}
\exists a\, b\, c : \mathbb{Z}.&\\
& f(n) = \text{if } n > 0 \text{ then } a \log n + b \text{ else } c\\
\wedge\ & \text{monotonic } f\\
\wedge\ & \forall n.\ f(n) \geqslant 0\\
\wedge\ & \forall n.\ n \leqslant 0 \implies f(n) \geqslant 1\\
\wedge\ & \forall n.\ n \geqslant 1 \implies f(n) \geqslant f(\tfrac{n}{2}) + 3\\
\wedge\ & \text{True}
\end{aligned}
$$

The substitution method in action

$$
\begin{aligned}
&\exists a\ b\ c : \mathbb{Z}. \\
&\qquad f(n) = \text{if } n > 0 \text{ then } a \log n + b \text{ else } c \\
&\quad \wedge\ a \geqslant 0 \ \wedge\ b \geqslant c \\
&\quad \wedge\ b \geqslant 0 \ \wedge\ c \geqslant 0 \\
&\quad \wedge\ c \geqslant 1 \\
&\quad \wedge\ b \geqslant c + 3 \ \wedge\ a \geqslant 3 \\
&\quad \wedge\ \text{True}
\end{aligned}
$$

The substitution method in action

$\exists a\ b\ c : \mathbb{Z}.$

$$
\begin{aligned}
&\wedge\ a \geqslant 0 \ \wedge\ b \geqslant c \\
&\wedge\ b \geqslant 0 \ \wedge\ c \geqslant 0 \\
&\wedge\ c \geqslant 1 \\
&\wedge\ b \geqslant c + 3 \ \wedge\ a \geqslant 3 \\
&\wedge\ \mathrm{True}
\end{aligned}
$$

The substitution method in action

$$\exists a\ b\ c : \mathbb{Z}.$$

$$
\begin{aligned}
&\wedge\ a \geqslant 0 \ \ \wedge\ \ b \geqslant c \\
&\wedge\ b \geqslant 0 \ \ \wedge\ \ c \geqslant 0 \\
&\wedge\ c \geqslant 1 \\
&\wedge\ b \geqslant c + 3 \ \ \wedge\ \ a \geqslant 3 \\
&\wedge\ \mathrm{True}
\end{aligned}
$$

Can be solved automatically.

The user does not have to manually provide values for a, b, and c.

Separation Logic with Time Credits

Linking code to cost assertions

Program specifications using Separation Logic

precondition program postcondition

$$\{P\}\ \ t\ \ \{Q\}$$

Linking code to cost assertions

Program specifications using Separation Logic with Time Credits

precondition program postcondition

$$\{\$n \star P\} \;\; t \;\; \{Q\}$$

Linking code to cost assertions

Program specifications using Separation Logic with Time Credits

precondition

program

postcondition

$$\{\$n \star P\}\ \ t\ \ \{Q\}$$

time credits

Time Credits: resources in separation logic

$$\$n$$

- $\$n$ describes the right to perform n function calls or loop iterations
- $\$(n + m) = \$n \star \$m$
- $\$0 = \mathrm{emp}$

Time Credits: resources in separation logic

$$\$n$$

- $\$n$ describes the right to perform n function calls or loop iterations
- $\$(n+m) = \$n \star \$m$
- $\$0 = \mathrm{emp}$
- Credits are not duplicable: $\$1 \;\not\Longrightarrow\; \$1 \star \$1$
- Enable amortized complexity analysis

Using time credits in the specification of `bsearch`

Specification of the complexity of `bsearch` using time credits:

$$\exists f : \mathbb{Z} \to \mathbb{Z}.\quad \begin{cases} f \in O(\lambda n.\ \log n) \\ \forall a\ x\ i\ j.\ \{\$(f(j-i)) \star \ldots\}\ (\texttt{bsearch a x i j})\ \{\ldots\} \end{cases}$$

Contribution: Possibly Negative Time Credits

Separation Logic with Time Credits in $\mathbb{N}$:

$$\begin{array}{rrcl} & \$0 & \equiv & \mathrm{emp} \\ \forall m\, n \in \mathbb{N}. & \$(m+n) & \equiv & \$m \star \$n \\ \forall n \in \mathbb{N}. & \$n & \Vdash & \mathrm{emp} \end{array}$$

My extension: Possibly Negative Time Credits in $\mathbb{Z}$:

$$\begin{array}{rrcl} & \$0 & \equiv & \mathrm{emp} \\ \forall m\, n \in \mathbb{Z}. & \$(m+n) & \equiv & \$m \star \$n \\ \forall n \in \mathbb{Z}. & \$n \star [n \geqslant 0] & \Vdash & \mathrm{emp} \end{array}$$

Corollary: $\$n \equiv \$m \star \$(n-m)$

Possibly Negative Time Credits enable simpler specifications

```
let index_of (v: 'a) (a: 'a array): int =
  (* returns the index of the first occurrence of v in a *)
```

Possibly Negative Time Credits enable simpler specifications

```
let index_of (v: 'a) (a: 'a array): int =
  (* returns the index of the first occurrence of v in a *)
```

$\forall a.\ \{\$(|a| + 1)\}$ `index_of` $v\ a$ $\{\lambda i.\ \mathrm{emp}\}$

Possibly Negative Time Credits enable simpler specifications

```
let index_of (v: 'a) (a: 'a array): int =
  (* returns the index of the first occurrence of v in a *)
```

$\forall a.\ \{\$(|a| + 1)\}$ `index_of` $v\ a$ $\{\lambda i.\ \mathrm{emp}\}$ (too coarse)

Possibly Negative Time Credits enable simpler specifications

```
let index_of (v: 'a) (a: 'a array): int =
  (* returns the index of the first occurrence of v in a *)
```

$\forall a.\ \{\$(|a|+1)\}$ `index_of` $v\ a$ $\{\lambda i.\ \mathrm{emp}\}$ (too coarse)

$\forall a.\ \{\$(|a|+1)\}$ `index_of` $v\ a$ $\{\lambda i.\ \$(|a|-i)\}$

Possibly Negative Time Credits enable simpler specifications

```
let index_of (v: 'a) (a: 'a array): int =
  (* returns the index of the first occurrence of v in a *)
```

$\forall a.\ \{\$(|a|+1)\}\ \texttt{index\_of}\ v\ a\ \{\lambda i.\ \mathrm{emp}\}$ (too coarse)

$\forall a.\ \{\$(|a|+1)\}\ \texttt{index\_of}\ v\ a\ \{\lambda i.\ \$(|a|-i)\}$ (restrictive?)

Possibly Negative Time Credits enable simpler specifications

```
let index_of (v: 'a) (a: 'a array): int =
  (* returns the index of the first occurrence of v in a *)
```

$\forall a.\ \{\$(|a|+1)\}$ `index_of` $v\ a\ \ \{\lambda i.\ \mathrm{emp}\}$ (too coarse)

$\forall a.\ \{\$(|a|+1)\}$ `index_of` $v\ a\ \ \{\lambda i.\ \$(|a|-i)\}$ (restrictive?)

$\forall a.\ \text{let}\ k := \min\{i \mid a.(i) = v\}\ \text{in}$
$\{\$(k+1)\}$ `index_of` $v\ a\ \ \{\lambda i.\ [i = k]\}$

Possibly Negative Time Credits enable simpler specifications

```
let index_of (v: 'a) (a: 'a array): int =
  (* returns the index of the first occurrence of v in a *)
```

$\forall a.\ \{\$(|a|+1)\}$ `index_of` $v\ a\ \{\lambda i.\ \mathrm{emp}\}$ (too coarse)

$\forall a.\ \{\$(|a|+1)\}$ `index_of` $v\ a\ \{\lambda i.\ \$(|a|-i)\}$ (restrictive?)

$\forall a.$ let $k := \min\{i \mid a.(i) = v\}$ in
$\{\$(k+1)\}$ `index_of` $v\ a\ \{\lambda i.\ [i = k]\}$ (too complicated)

Possibly Negative Time Credits enable simpler specifications

```
let index_of (v: 'a) (a: 'a array): int =
  (* returns the index of the first occurrence of v in a *)
```

$\forall a.\ \{\$(|a|+1)\}$ `index_of` $v\ a$ $\{\lambda i.\ \mathrm{emp}\}$ (too coarse)

$\forall a.\ \{\$(|a|+1)\}$ `index_of` $v\ a$ $\{\lambda i.\ \$(|a|-i)\}$ (restrictive?)

$\forall a.$ let $k := \min\{i \mid a.(i) = v\}$ in
$\{\$(k+1)\}$ `index_of` $v\ a$ $\{\lambda i.\ [i = k]\}$ (too complicated)

$\forall a.\ \{\mathrm{emp}\}$ `index_of` $v\ a$ $\{\lambda i.\ \$(-i-1)\}$

Possibly Negative Time Credits enable simpler specifications

```
let index_of (v: 'a) (a: 'a array): int =
  (* returns the index of the first occurrence of v in a *)
```

$\forall a.\ \{\$(|a|+1)\}\ \texttt{index\_of}\ v\ a\ \{\lambda i.\ \mathrm{emp}\}$ (too coarse)

$\forall a.\ \{\$(|a|+1)\}\ \texttt{index\_of}\ v\ a\ \{\lambda i.\ \$(|a|-i)\}$ (restrictive?)

$\forall a.\ \mathrm{let}\ k := \min\{i \mid a.(i) = v\}\ \mathrm{in}$
$\{\$(k+1)\}\ \texttt{index\_of}\ v\ a\ \{\lambda i.\ [i = k]\}$ (too complicated)

$\boxed{\forall a.\ \{\mathrm{emp}\}\ \texttt{index\_of}\ v\ a\ \{\lambda i.\ \$(-i-1)\}}$

Time Credits in $\mathbb{Z}$: benefits

- Simpler specifications
  (when the cost depends on the result)
- Significant reduction of the number of intermediate side-conditions
  (can accumulate debts and pay them off once at the end)
- Simpler loop invariants
  (no need to justify that a number of credits is positive at each step)

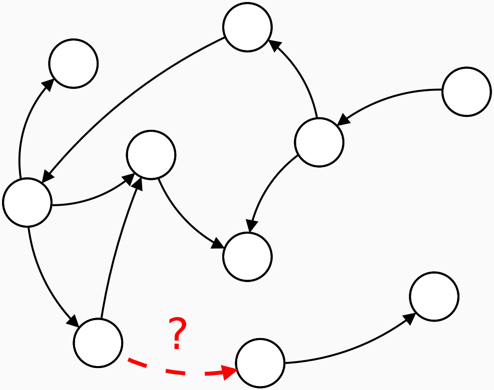
Case Study: an Incremental Cycle Detection Algorithm

Our main case study

Verification of a state-of-the-art incremental cycle detection algorithm due to Bender, Fineman, Gilbert and Tarjan (2016).

Our main case study

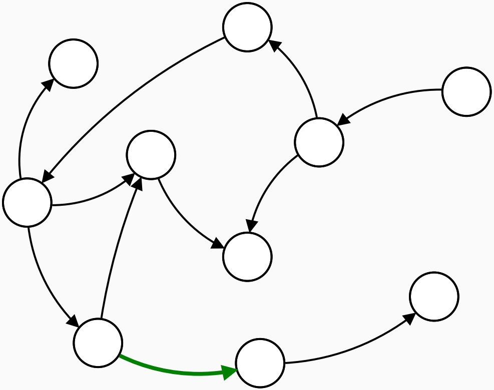
Verification of a state-of-the-art incremental cycle detection algorithm due to Bender, Fineman, Gilbert and Tarjan (2016).



The problem: checking for acyclicity of a dynamically constructed graph

Our main case study

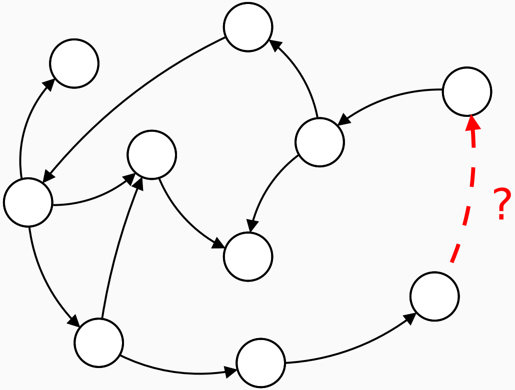
Verification of a state-of-the-art incremental cycle detection algorithm due to Bender, Fineman, Gilbert and Tarjan (2016).



The problem: checking for acyclicity of a dynamically constructed graph

Our main case study

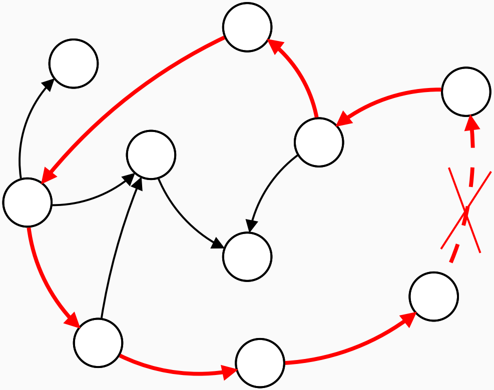
Verification of a state-of-the-art incremental cycle detection algorithm due to Bender, Fineman, Gilbert and Tarjan (2016).



The problem: checking for acyclicity of a dynamically constructed graph

Our main case study

Verification of a state-of-the-art incremental cycle detection algorithm due to Bender, Fineman, Gilbert and Tarjan (2016).



The problem: checking for acyclicity of a dynamically constructed graph

Minimal OCaml interface

```
type add_edge_result =
  | EdgeAdded
  | EdgeCreatesCycle

val add_edge_or_detect_cycle :
  graph -> vertex -> vertex -> add_edge_result
```

Our main case study (2)

A state-of-the-art algorithm:

- non-trivial implementation (200 lines of compact OCaml code)
- subtle complexity analysis
- used in Coq (universe constraints) and Dune (build dependencies)

Incremental Cycle Detection: Complexity

Naive algorithm: $O(m)$ traversal at each arc insertion.
Inserting m arcs costs $O(m^2)$.

Using Bender et al.'s algorithm, inserting m arcs costs:

$$O(m \cdot \min(\sqrt{m}, n^{2/3}))$$

Or:

- $O(m\sqrt{m})$ for sparse graphs;
- $O(mn^{2/3})$ for dense graphs.

Incremental Cycle Detection: Complexity

Naive algorithm: $O(m)$ traversal at each arc insertion.
Inserting m arcs costs $O(m^2)$.

Using Bender et al.'s algorithm, inserting m arcs costs:

$$O(m \cdot \min(\sqrt{m}, n^{2/3}))$$

Or:

- $O(m\sqrt{m})$ for sparse graphs;
- $O(mn^{2/3})$ for dense graphs.

⚠ Specifies the **cost of a sequence of operations**.
No closed formula for the amortized cost of a single operation.

Toplevel specification (functional correctness only)

“$\mathrm{IsDAG}\ g\ G$”: a **Separation Logic predicate** describing the algorithm’s data structure, at address g, representing the graph G.

Toplevel specification (functional correctness only)

"$\mathrm{IsDAG}\, g\ G$": a **Separation Logic predicate** describing the algorithm's data structure, at address g, representing the graph G.

$$
\begin{array}{ll}
\forall g\, G\, v\, w. & \text{let } m := |\text{edges } G| \text{ in} \\
& \text{let } n := |\text{vertices } G| \text{ in} \\
& v, w \in \text{vertices } G \ \wedge\ (v, w) \notin \text{edges } G \implies \\
& \left\{ \ \mathrm{IsDAG}\, g\ G \ \right\} \\
& (\texttt{add\_edge\_or\_detect\_cycle}\ g\ v\ w) \\
& \left\{ \begin{array}{l} \lambda\, \text{res. match res with} \\ \quad \mid \text{EdgeAdded} \Rightarrow \mathrm{IsDAG}\, g\ (G + (v, w)) \\ \quad \mid \text{EdgeCreatesCycle} \Rightarrow [w \longrightarrow^*_G v]) \end{array} \right\}
\end{array}
$$

Toplevel specification (correctness and complexity)

"$\mathrm{IsDAG}\,g\,G$": a **Separation Logic predicate** describing the algorithm's data structure, at address g, representing the graph G.

$$
\begin{array}{rl}
\exists \psi. & \boxed{\psi \in O(m \cdot \min(\sqrt{m}, n^{2/3}) + n)} \;\wedge \\
\forall g\, G\, v\, w. & \text{let } m := |\text{edges } G| \text{ in} \\
& \text{let } n := |\text{vertices } G| \text{ in} \\
& v, w \in \text{vertices } G \;\wedge\; (v, w) \notin \text{edges } G \implies \\
& \left\{ \; \mathrm{IsDAG}\, g\, G \star \boxed{\$(\psi\,(m+1, n) - \psi\,(m, n))} \; \right\} \\
& (\texttt{add\_edge\_or\_detect\_cycle}\; g\; v\; w) \\
& \left\{ \begin{array}{l} \lambda\, \text{res. match res with} \\ \quad \mid \text{EdgeAdded} \Rightarrow \mathrm{IsDAG}\, g\, (G + (v, w)) \\ \quad \mid \text{EdgeCreatesCycle} \Rightarrow [w \longrightarrow_G^* v] \end{array} \right\}
\end{array}
$$

Case Study: Summary

Final result

- A formally verified OCaml library for incremental cycle detection
- Succinct specification
- Robust proof (no hardcoded constants or manual accounting)
- Code has been integrated in Dune, fixing some complexity bugs

Contributions

- State-of-the-art result on verified graph algorithms
- A crucial improvement to the algorithm to make it truly incremental

Conclusion

Summary

In this talk:

- Motivation for the verification of complexity using O
- Cost functions and their inference
- Possibly Negative Time Credits
- A large case study

Summary

In this talk:

- Motivation for the verification of complexity using O
- Cost functions and their inference
- Possibly Negative Time Credits
- A large case study

More in the manuscript:

- Specific challenges related to multivariate O
- Summation lemmas for the analysis of `for`-loops
- More case studies

Perspectives

Further automation

- in Coq: high-level reasoning on synthesized cost expressions (master theorem, simplification procedures)
- integration with automated complexity analysis tools
- integration of the approach in more automated verification tools

Perspectives

Further automation

- in Coq: high-level reasoning on synthesized cost expressions (master theorem, simplification procedures)
- integration with automated complexity analysis tools
- integration of the approach in more automated verification tools

Implement support to allow extracting concrete complexity bounds

Perspectives

Further automation

- in Coq: high-level reasoning on synthesized cost expressions (master theorem, simplification procedures)
- integration with automated complexity analysis tools
- integration of the approach in more automated verification tools

Implement support to allow extracting concrete complexity bounds

Even more challenging applications:

- space complexity
- concurrent programs
- cache-oblivious algorithms