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

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

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

and `eapp cv_pb l2r infos (lft1, st1) (lft2, st2) cuniv =
  Control.check_for Interrupt();
  (* First handler reduce both terms *)
  let ninfos = infos with_reds infos.cnvInf betaiotazeta in
  let (hd1, v1 as appr1) = whd_stack ninfos infos.lft_tab (fst st1) in
  let (hd2, v2 as appr2) = whd_stack ninfos infos.rgt_tab (fst st2) in
  let appr1 = (lft1, appr1) and appr2 = (lft2, appr2) in
  (* We delay the computation of the lifts that apply to the head of
   * [el_stack] inside the branches where they are actually used. *)
  match (form_of_hd1, form_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
          raise NotConvertible;
        else
          (match_kind a1, match_kind a2) as m =>
            if Int.equal m m then
              convert_stacks l2r infos lft1 lft2 v1 v2 cuniv
            else
              raise NotConvertible
      | _ -> raise NotConvertible
    )
  | (FVar ((e1v1, a1v1), env1), FVar ((e2v2, a2v2), env2)) ->
    if FVar.equal e1v1 e2v2 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_vec l2r infos el1 el2 Array.map (mc_close env1) args1
      Array.map (mc_close env2) args2 cuniv
    else
      raise NotConvertible
  | (FConst c1, FConst c2) ->
    if FConst.equal c1 c2 then
      let el1 = el_stack lft1 v1 in
      let el2 = el_stack lft2 v2 in
      if Int.equal (relc_rel n el1) (relc_rel m el2) then
        convert_stacks l2r infos lft1 lft2 v1 v2 cuniv
      else
        raise NotConvertible
  | (FRel n, FRel m) ->
    let el1 = el_stack lft1 v1 in
    let el2 = el_stack lft2 v2 in
    if Int.equal (relc_rel n el1) (relc_rel m el2) then
      convert_stacks l2r infos lft1 lft2 v1 v2 cuniv
    else
      raise NotConvertible
  | (FFcall f1l, FFcall f2l) ->
    try
      let cuniv = conv_table_key infos.cnvInf f1l f2l cuniv in
      convert_stacks l2r infos lft1 lft2 v1 v2 cuniv
    with NotConvertible | Univ.UniverseInconsistency
    | _ else throw epure cuniv
    |> exception_of_cuniv
    (* else the oracle tells which constant is to be expanded.*
  | FLocal e1, FLocal e2 ->
    let escA = escape oracle of infos.cnvInf cnvInf in
    (* try
      let cuniv = conv_table_key infos.cnvInf f1l f2l cuniv in
      convert_stacks l2r infos lft1 lft2 v1 v2 cuniv
    with NotConvertible | Univ.UniverseInconsistency
    | _ else throw epure cuniv
    |> exception_of_cuniv
    (* else the oracle tells which constant is to be expanded.*
  | FIndex i1, FIndex i2 ->
    let escA = escape oracle of infos.cnvInf cnvInf in
    (* try
      let cuniv = conv_table_key infos.cnvInf f1l f2l cuniv in
      convert_stacks l2r infos lft1 lft2 v1 v2 cuniv
    with NotConvertible | Univ.UniverseInconsistency
    | _ else throw epure cuniv
    |> exception_of_cuniv
    (* else the oracle tells which constant is to be expanded.*
  | _ -> raise NotConvertible
```

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

  uprobe_start_dup_mmap();
  if (down_write_kitable(oldmm->mm_map)) {
    retval = -EINTR;
    goto fail_uprobe_end;
  }
  flush_cache_mmap(oldmm);
  uprobe_dup_mmap(oldmm, mm);
  /* Not linked in yet. No deadlock potential. */
  down_write_nested(&mm->mm_map, 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_node;
  rb_parent = NULL;
  pprev = &mm->mm_map;
  retval = kmalloc_page(mm, oldmm);
  if (retval)
    goto out;
  retval = kmapped_page(mm, oldmm);"
Real-world programs are usually very large.
Real-world programs are usually very large.

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

- 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)
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)
- 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)
What do we expect from a program?

- Safety (does not crash)
- Partial correctness (returns a correct result; might not terminate)
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?

- 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)

More confidence means higher expectations, and less confidence means lower expectations.
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)
- Complexity bound (runs in a predictable amount of time)
- Real-time bound (runs within a precise time budget)
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)
- 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?

- **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)
What do we expect from a program?

<table>
<thead>
<tr>
<th>Confidence Level</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less</td>
<td>Safety</td>
<td>(does not crash)</td>
</tr>
<tr>
<td></td>
<td>Partial correctness</td>
<td>(returns a correct result; might not terminate)</td>
</tr>
<tr>
<td></td>
<td>Total correctness</td>
<td>(always returns a correct result)</td>
</tr>
<tr>
<td></td>
<td>Complexity bound</td>
<td>(runs in a predictable amount of time)</td>
</tr>
<tr>
<td></td>
<td>Real-time bound</td>
<td>(runs within a precise time budget)</td>
</tr>
<tr>
<td></td>
<td>Security</td>
<td>(e.g. timing side channel)</td>
</tr>
<tr>
<td></td>
<td>Fault tolerant</td>
<td>(resists to hardware faults)</td>
</tr>
</tbody>
</table>

in this work
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 containing 1 billion elements:

- A correct binary search should do at most 30 recursive calls \(2^{30} \approx 1\) 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
In summary, on an array of size $n$:

- We expect $O(\log n)$ recursive calls;
- But our program does up to $n$ recursive calls.
In summary, on an array of size $n$:

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

<table>
<thead>
<tr>
<th>in this work</th>
<th>less confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safety</td>
<td>(does not crash)</td>
</tr>
<tr>
<td>Partial correctness</td>
<td>(returns a correct result; might not terminate)</td>
</tr>
<tr>
<td>Total correctness</td>
<td>(always returns a correct result)</td>
</tr>
<tr>
<td>Complexity bound</td>
<td>(runs in a predictable amount of time)</td>
</tr>
</tbody>
</table>
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)
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

```ocaml
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

```plaintext
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

```ocaml
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 $j - i$:

- $j - i \leq 0$: $O(1)$. 
Informal reasoning principles on $O$ can be abused

1. \textbf{let rec bsearch a x i j =}
2. \hspace{1em} \textbf{if} j \leq i \textbf{ then } -1 \textbf{ else}
3. \hspace{2em} \textbf{let} k = i + (j - i) / 2 \textbf{ in}
4. \hspace{3em} \textbf{if} x = a.(k) \textbf{ then } k
5. \hspace{3em} \textbf{else if} x < a.(k) \textbf{ then}
6. \hspace{4em} bsearch a x i k
7. \hspace{3em} \textbf{else}
8. \hspace{4em} bsearch a x (k+1) j

\textbf{Claim:}
\hspace{1em} bsearch a x i j costs $O(1)$.

\textbf{Proof:}
\hspace{1em} By induction on $j - i$:
\begin{itemize}
  \item $j - i \leq 0$: $O(1)$.
  \item $j - i > 0$: $O(1) + O(1) + O(1) = O(1)$.
\end{itemize}
Informal reasoning principles on $O$ can be abused

1. let rec bsearch a x i j =
2.   if j <= i then -1 else
3.     let k = i + (j - i) / 2 in
4.     if x = a.(k) then k
5.     else if x < a.(k) then
6.         bsearch a x i k
7.     else
8.         bsearch a x (k+1) j

Claim:

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

Proof:

By induction on $j - i$:

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

Where is the catch?
Informal reasoning principles on $O$ can be abused

```plaintext
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 $j - i$:

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

...but which statement are we proving?
Meaning of $O(1)$

What we just proved:

$$\forall i, j, \exists c, \text{ "bsearch } a \times i \ j\text{" performs at most } c \text{ function calls}$$
Meaning of $O(1)$

What we just proved:

$$\forall i,j, \exists c, \text{ "bsearch } a \times i \text{ } j\text{" performs at most } c \text{ function calls}$$

What “$O(1)$” means:

$$\exists c, \forall i,j, \text{ "bsearch } a \times i \text{ } j\text{" performs at most } c \text{ function calls}$$
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, “λn. 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 0 (f (n/2))
                        (f (n - n/2 - 1))
  )
)```
Interactive construction of the cost function $f$

if $j \leq i$ then $-1$ else
let $k = i + (j - i) / 2$ in
if $x = \text{Array}.\text{get} \ a \ k$ then $k$
else if $x < \text{Array}.\text{get} \ a \ k$
    then $\text{bsearch} \ a \ x \ i \ k$
else $\text{bsearch} \ a \ x \ (k+1) \ j$

\[ f(j-i) \geq 1 + \ldots \]

a hole ("\ldots") is implemented as an evar in Coq
Interactive construction of the cost function $f$

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

\[ f(j-i) \geq 1 + (\text{if } j \leq i \text{ then } \ldots \text{ else } \ldots) \]
```
Interactive construction of the cost function $f$

$$
\text{if } j \leq i \text{ then } -1 \text{ else }
\begin{align*}
\text{let } k & = i + (j - i) / 2 \text{ in } \\
\text{if } x = \text{Array.get } a \ k \text{ then } k \\
\text{else if } x < \text{Array.get } a \ k \\
& \quad \text{then bsearch } a \ x \ i \ k \\
& \quad \text{else bsearch } a \ x \ (k+1) \ j
\end{align*}
$$

$$
f (j-i) \geq 1 + (\text{if } j \leq i \text{ then } \ldots \text{ else } \ldots)
$$
Interactive construction of the cost function \( f \)

\[
\text{if } j \leq i \text{ then } -1 \text{ else }
\]

\[
\text{let } k = i + (j - i) / 2 \text{ in }
\]

\[
\text{if } x = \text{Array.get a k} \text{ then } k
\]

\[
\text{else if } x < \text{Array.get a k}
\]

\[
\text{then } \text{bsearch a x i k}
\]

\[
\text{else } \text{bsearch a x (k+1) j}
\]

\[
f (j-i) \geq 1 + (\text{if } (j-i) \leq 0 \text{ then } ... \text{ else } ...)
\]
Interactive construction of the cost function $f$

```
if \( j \leq i \) then -1 else

let \( k = i + (j - i) / 2 \) in

if \( x = \text{Array}\.get\ a\ k \) then \( k \)
else if \( x < \text{Array}\.get\ a\ k \)
    then bsearch \( a\ x\ i\ k \)
else bsearch \( a\ x\ (k+1)\ j \)
```

\[
f(j-i) \geq 1 + (\text{if } (j-i) \leq 0 \text{ then } 0 \text{ else } ...)
\]
Interactive construction of the cost function $f$

\[
\begin{align*}
\text{if } j &\leq i \text{ then } -1 \text{ else } \\
\text{let } k = i + (j - i) / 2 \text{ in } \\
\text{if } x = \text{Array}.get a k \text{ then } k \\
\text{else if } x < \text{Array}.get a k \\
\quad \text{then bsearch } a x i k \\
\quad \text{else bsearch } a x (k+1) j
\end{align*}
\]

\[
f(j-i) \geq 1 + ( \\
\quad \text{if } (j-i) \leq 0 \text{ then } 0 \text{ else } \\
\quad 0 + \ldots
\)
Interactive construction of the cost function $f$

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

\[f(j-i) \geq 1 + (\]
    \[\text{if } (j-i) \leq 0 \text{ then } 0 \text{ else}
    \]
    \[0 + 1 + ...\]
\[)\]
Interactive construction of the cost function $f$

```plaintext
if $j \leq i$ then -1 else
  let $k = i + (j - i) / 2$ in
  if $x = \text{Array.get} a k$ then $k$
  else if $x < \text{Array.get} a k$
    then $\text{bsearch} a x i k$
    else $\text{bsearch} a x (k+1) j$

---

$f(j-i) \geq 1 + ($

  $if (j-i) \leq 0$ then 0 else
  $0 + 1 + \max \ldots \ldots$

$)$
```
Interactive construction of the cost function $f$

\[
\text{if } j \leq i \text{ then -1 else }
\]

\[
\text{let } k = i + (j - i) / 2 \text{ in }
\]

\[
\text{if } x = \text{Array.get a k then } k
\]

\[
\text{else if } x < \text{Array.get a k }
\]

\[
\text{then bsearch a x i k}
\]

\[
\text{else bsearch a x (k+1) j}
\]

\[
f (j-i) \geq 1 + (\text{if } (j-i) \leq 0 \text{ then } 0 \text{ else }
\]

\[
0 + 1 + \max 0 ...
\]

\)
Interactive construction of the cost function $f$

```plaintext
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
```

$$f(j-i) \geq 1 + (\begin{cases} 0 & (j-i) \leq 0 \\ 0 + 1 + \max 0 (1 + ...) & \end{cases})$$
Interactive construction of the cost function $f$

```plaintext
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 $f$

```plaintext
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
```

$$f(j-i) \geq 1 + (\quad$$
$$\quad \text{if } (j-i) \leq 0 \text{ then } 0 \text{ else}$$
$$\quad \quad 0 + 1 + \max 0 (\quad$$
$$\quad \quad \quad 1 + \max (f((j-i)/2)) \ldots$$
$$\quad \quad )$$
)
Interactive construction of the cost function f

```plaintext
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 0 (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
```

$$f(n) \geq 1 + (\begin{array}{l}
\text{if } n \leq 0 \text{ then } 0 \text{ else }
\left( 0 + 1 + \max 0 \left( 1 + \max (f(n/2)) \right) \right)
\end{array})$$
For bsearch, there remains to find a $f \in O(\lambda n. \log n)$ such that:

$$\forall n. f(n) \geq 1 + \begin{cases} 
0 & \text{if } n \leq 0 \\
1 + \max(0, 1 + \max(f(n/2), f(n - n/2 - 1))) & \text{otherwise}
\end{cases}$$
From cost equation to asymptotic bound

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

$$\forall n. f(n) \geq 1 + \begin{cases} 0 & \text{if } n \leq 0 \\ 1 + \max(0, 1 + \max(f(n/2), f(n - n/2 - 1))) & \end{cases}$$

- 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

\[ \exists f : \mathbb{Z} \rightarrow \mathbb{Z}. \]

\[ \forall n. \ f(n) \geq 1 + \begin{cases} 
0 & \text{if } n \leq 0 \\
1 + \max(0, 1 + \max(f(\frac{n}{2}), f(n - \frac{n}{2} - 1))) & \text{otherwise} 
\end{cases} \]

\[ \land \ f \in O(\lambda n. \ \log n) \]
The substitution method in action

\[ \exists f : \mathbb{Z} \rightarrow \mathbb{Z}. \]

\[ \text{monotonic } f \]
\[ \land \forall n. \; f(n) \geq 0 \]
\[ \land \forall n. \; n \leq 0 \implies f(n) \geq 1 \]
\[ \land \forall n. \; n \geq 1 \implies f(n) \geq f\left(\frac{n}{2}\right) + 3 \]
\[ \land f \in O(\lambda n. \log n) \]
The substitution method in action

\[\exists a, b : \mathbb{Z}.\]

\[f(n) = a \log n + b\]

\[\land \text{monotonic } f\]

\[\land \forall n. \ f(n) \geq 0\]

\[\land \forall n. \ n \leq 0 \implies f(n) \geq 1\]

\[\land \forall n. \ n \geq 1 \implies f(n) \geq f\left(\frac{n}{2}\right) + 3\]

\[\land f \in O(\lambda n. \log n)\]
The substitution method in action

\[ \exists a \ b : \mathbb{Z}. \]
\[ f(n) = a \log n + b \quad (\text{issue when } n = 0) \]
\[ \land \ \text{monotonic } f \]
\[ \land \ \forall n. \ f(n) \geq 0 \]
\[ \land \ \forall n. \ n \leq 0 \implies f(n) \geq 1 \]
\[ \land \ \forall n. \ n \geq 1 \implies f(n) \geq f\left(\frac{n}{2}\right) + 3 \]
\[ \land \ f \in O(\lambda n. \ \log n) \]
The substitution method in action

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

\begin{align*}
  f(n) &= \text{if } n > 0 \text{ then } a \log n + b \text{ else } c \\
  \land \text{monotonic } f \\
  \land \forall n. \ f(n) \geq 0 \\
  \land \forall n. \ n \leq 0 \implies f(n) \geq 1 \\
  \land \forall n. \ n \geq 1 \implies f(n) \geq f\left(\frac{n}{2}\right) + 3 \\
  \land f \in O(\lambda n. \log n)
\end{align*}
The substitution method in action

\[ \exists a, b, c : \mathbb{Z}. \]

\[ f(n) = \text{if } n > 0 \text{ then } a \log n + b \text{ else } c \]
\[ \wedge \text{monotonic } f \]
\[ \wedge \forall n. \ f(n) \geq 0 \]
\[ \wedge \forall n. \ n \leq 0 \implies f(n) \geq 1 \]
\[ \wedge \forall n. \ n \geq 1 \implies f(n) \geq f\left(\frac{n}{2}\right) + 3 \]
\[ \wedge \text{True} \]
The substitution method in action

\[ \exists a \ b \ c : \mathbb{Z} . \]

\[ f(n) = \text{if } n > 0 \text{ then } a \log n + b \text{ else } c \]

\[ \land a \geq 0 \land b \geq c \]

\[ \land b \geq 0 \land c \geq 0 \]

\[ \land c \geq 1 \]

\[ \land b \geq c + 3 \land a \geq 3 \]

\[ \land \text{True} \]
The substitution method in action

\[ \exists a \ b \ c : \mathbb{Z}. \]

\[ \land a \geq 0 \land b \geq c \]
\[ \land b \geq 0 \land c \geq 0 \]
\[ \land c \geq 1 \]
\[ \land b \geq c + 3 \land a \geq 3 \]
\[ \land \text{True} \]
The substitution method in action

\[ \exists a \ b \ c : \mathbb{Z}. \]

\[ \land \ a \geq 0 \ \land \ b \geq c \]
\[ \land \ b \geq 0 \ \land \ c \geq 0 \]
\[ \land \ c \geq 1 \]
\[ \land \ b \geq c + 3 \ \land \ a \geq 3 \]
\[ \land \ True \]

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

\[ \{ P \} t \{ Q \} \]
Linking code to cost assertions

Program specifications using Separation Logic with Time Credits

precondition \[\{n \ast P\}\]  \[t\]  postcondition \[\{Q\}\]
Linking code to cost assertions

Program specifications using Separation Logic with Time Credits

precondition \[\{ n \star P \} \]

program \( t \)

postcondition \( \{ Q \} \)

time credits
Time Credits: resources in separation logic

$n$

- $n$ describes the right to perform $n$ function calls or loop iterations
- $(n + m) = n \times m$
- $0 = \text{emp}$
Time Credits: resources in separation logic

\[ n \]

- \( n \) describes the right to perform \( n \) function calls or loop iterations
- \( (n + m) = n \times m \)
- \( 0 = \text{emp} \)
- Credits are not duplicable: \( 1 \not\implies 1 \times 1 \)
- Enable amortized complexity analysis
Using time credits in the specification of bsearch

Specification of the complexity of bsearch using time credits:

\[ \exists f : \mathbb{Z} \rightarrow \mathbb{Z}. \]

\[ \begin{align*}
    f & \in O(\lambda n. \log n) \\
    \forall a \ x i j. \ (f(j - i)) & \ldots \ (\text{bsearch } a \ x i j) \{ \ldots \}
\end{align*} \]
Contribution: Possibly Negative Time Credits

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

\[
\begin{align*}
0 & \equiv \text{emp} \\
\forall m \ n \in \mathbb{N}. \quad (m + n) & \equiv m \ast n \\
\forall n \in \mathbb{N}. \quad & n \vdash \text{emp}
\end{align*}
\]

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

\[
\begin{align*}
0 & \equiv \text{emp} \\
\forall m \ n \in \mathbb{Z}. \quad (m + n) & \equiv m \ast n \\
\forall n \in \mathbb{Z}. \quad & n \ast [n \geq 0] \vdash \text{emp}
\end{align*}
\]

Corollary: \( n \equiv m \ast (n - m) \)
Possibly Negative Time Credits enable simpler specifications

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

(* returns the index of the first occurrence of v in a *)

∀a. {$(|a| + 1)} index_of v a \{\lambda i. \text{emp}\}
Possibly Negative Time Credits enable simpler specifications

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

∀a. {$(|a| + 1)} index_of v a {λi. emp}                 (too coarse)
```
Possibly Negative Time Credits enable simpler specifications

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

∀a. {$(|a| + 1)} index_of v a {λi. emp}  (too coarse)

∀a. {$(|a| + 1)} index_of v a {λi. $(|a| − i)}
```

Possibly Negative Time Credits enable simpler specifications

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

∀a. {$(|a| + 1)} index_of v a \{λi. \text{emp}\} \quad \text{(too coarse)}

∀a. {$(|a| + 1)} index_of v a \{λi. $(|a| - i)\} \quad \text{(restrictive?)}
```
Possibly Negative Time Credits enable simpler specifications

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

∀a. {$(|a| + 1)} index_of v a {λi. emp}  
    (too coarse)

∀a. {$(|a| + 1)} index_of v a {λi. $(|a| - i)}  
    (restrictive?)

∀a. let k := min {i | a.(i) = v} in
    {$(k + 1)} index_of v a {λi. [i = k]}
```
Possibly Negative Time Credits enable simpler specifications

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

\forall a. \{ |a| + 1 \} \text{index_of} v a \{ \lambda i. \text{emp} \} \tag{too coarse}

\forall a. \{ |a| + 1 \} \text{index_of} v a \{ \lambda i. |a| - i \} \tag{restrictive?}

\forall a. \text{let } k := \text{min} \{ i \mid a.(i) = v \} \text{ in}
  \{ k + 1 \} \text{index_of} v a \{ \lambda i. [i = k] \} \tag{too complicated}
```
Possibly Negative Time Credits enable simpler specifications

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

∀a. {$(|a| + 1)} index_of v a {λi. emp} (too coarse)

∀a. {$(|a| + 1)} index_of v a {λi. $(|a| - i)} (restrictive?)

∀a. let k := min {i | a.i = v} in
{$(k + 1)} index_of v a {λi. [i = k]} (too complicated)

∀a. {emp} index_of v a {λi. $(i - 1)}
```
Possibly Negative Time Credits enable simpler specifications

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

\forall a. \{(|a| + 1)\} \text{index\_of } v \ a \ \{\lambda i. \text{emp}\} \quad \text{(too coarse)}

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

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

\forall a. \{\text{emp}\} \text{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

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

```ocaml
type add_edge_result =
  | EdgeAdded
  | EdgeCreatesCycle
```
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)

“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)

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

\[
\forall g \ G \ v \ w. \quad \text{let } m := |\text{edges } G| \text{ in }
\text{let } n := |\text{vertices } G| \text{ in }
\text{let } v, w \in \text{vertices } G \land (v, w) \notin \text{edges } G \implies
\{ \text{IsDAG } g \ G \\
(\text{add\_edge\_or\_detect\_cycle } g \ v \ w) \\
\lambda \text{res. match res with}
\{ \text{EdgeAdded } \Rightarrow \text{IsDAG } g \ (G + (v, w)) \\
\text{EdgeCreatesCycle } \Rightarrow [w \rightarrow^* G v] \}
\}
\]
Toplevel specification (correctness and complexity)

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

\[
\exists \psi. \quad \psi \in O(m \cdot \min(\sqrt{m}, n^{2/3}) + n) \quad \land \\
\forall g \ G \ v \ w. \quad \text{let } m := |\text{edges } G| \text{ in} \\
\quad \text{let } n := |\text{vertices } G| \text{ in} \\
\quad v, w \in \text{vertices } G \land (v, w) \notin \text{edges } G \implies
\{ \ 	ext{IsDAG} \ g \ G \star \ (\psi(m + 1, n) - \psi(m, n)) \ \\
\text{(add_edge_or_detect_cycle} \ g \ v \ w) \ \\
\lambda \text{res. match res with} \\
\quad | \text{EdgeAdded} \Rightarrow \text{IsDAG} \ g \ (G + (v, w)) \\
\quad | \text{EdgeCreatesCycle} \Rightarrow [w \xrightarrow{G}^* v] \ \\
\} \quad \land \\
\]
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
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
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