Formalizing Asymptotic Complexity Claims via Deductive Program Verification

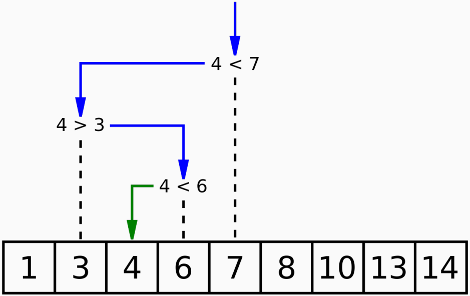
Armaël Guéneau

Gallium

Formalizing **Asymptotic Complexity** Claims via Deductive Program Verification

Recall our undergrad algorithm courses...

"Is the value 4 present in this sorted array?"



"Binary search finds the element in time $O(\log n)$"

Homework: implement binary search

```
(* Requires arr to be a sorted array of integers.
   Returns k such that i <= k < j and arr.(k) = v
   or -1 if there is no such k. *)
let rec bsearch (arr: int array) v i j =
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if v = arr.(k) then k
    else if v < arr.(k) then bsearch arr v i k
    else bsearch arr v (i+1) j
```

Homework: implement binary search

```
(* Requires arr to be a sorted array of integers.
   Returns k such that i <= k < j and arr.(k) = v
   or -1 if there is no such k. *)
let rec bsearch (arr: int array) v i j =
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if v = arr.(k) then k
    else if v < arr.(k) then bsearch arr v i k
    else bsearch arr v (i+1) j
```

```
# bsearch [|1;3;4;6;7;8;10;13;14|] 4 0 9;;
- : int = 2
```

It works! We could even prove that it **always** works.

Homework: implement binary search

```
(* Requires arr to be a sorted array of integers.
   Returns k such that i <= k < j and arr.(k) = v
   or -1 if there is no such k. *)
let rec bsearch (arr: int array) v i j =
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if v = arr.(k) then k
    else if v < arr.(k) then bsearch arr v i k
    else bsearch arr v (i+1) j
```

But there is a complexity bug…

Homework: implement binary search

```
(* Requires arr to be a sorted array of integers.
   Returns k such that i <= k < j and arr.(k) = v
   or -1 if there is no such k. *)
let rec bsearch (arr: int array) v i j =
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if v = arr.(k) then k
    else if v < arr.(k) then bsearch arr v i k
    else bsearch arr v (k+1) j
```

Homework: implement binary search

```
(* Requires arr to be a sorted array of integers.
   Returns k such that i <= k < j and arr.(k) = v
   or -1 if there is no such k. *)
let rec bsearch (arr: int array) v i j =
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if v = arr.(k) then k
    else if v < arr.(k) then bsearch arr v i k
    else bsearch arr v (k+1) j
```

Complexity bugs can be critical

`http://ocert.org/advisories/ocert-2011-003.html`

"Denial of Service via Algorithmic Complexity Attacks", S. Crosby, D. Wallach

One of the things we do at Gallium...

Machine-checked proofs of programs

One of the things we do at Gallium...

Machine-checked proofs of programs

including their algorithmic complexity

Formalizing Asymptotic Complexity Claims via Deductive Program Verification

What is formal program verification?

- People write programs

What is formal program verification?

- People write programs
- Programs contain bugs (ie. sometimes, they do not behave as expected)

What is formal program verification?

- People write programs
- Programs contain bugs (ie. sometimes, they do not behave as expected)

Formal verification is a set of techniques for:

What is formal program verification?

- People write programs
- Programs contain bugs (ie. sometimes, they do not behave as expected)

Formal verification is a set of techniques for:

- Writing a wishlist about a program (aka specification)

What is formal program verification?

- People write programs
- Programs contain bugs (ie. sometimes, they do not behave as expected)

Formal verification is a set of techniques for:

- Writing a wishlist about a program (aka specification)
- Checking the program against this wishlist ("is the program correct?")

Formalizing Asymptotic Complexity Claims via Deductive Program Verification

Deductive verification

From the code of the program and the specification, deduce a set of proof obligations, and try to prove those.

- Automated proofs: FramaC (C), Verifast (C, Java), Infer (C, C++, Obj-C, Java), Why3 (OCaml)
- Interactive proofs, using “proof assistants”: Coq, Isabelle

What are typical properties written in a specification?

- The program does not crash when you run it

What are typical properties written in a specification?

- The program does not crash when you run it
- The program terminates (does not get stuck in a loop)

What are typical properties written in a specification?

- The program does not crash when you run it
- The program terminates (does not get stuck in a loop)
- The program computes the right result

What are typical properties written in a specification?

- The program does not crash when you run it
- The program terminates (does not get stuck in a loop)
- The program computes the right result
- The program does not leak secrets (e.g. for crypto primitives)

What are typical properties written in a specification?

- The program does not crash when you run it
- The program terminates (does not get stuck in a loop)
- The program computes the right result
- The program does not leak secrets (e.g. for crypto primitives)
- **The program does not use too much time**

What are typical properties written in a specification?

- The program does not crash when you run it
- The program terminates (does not get stuck in a loop)
- The program computes the right result
- The program does not leak secrets (e.g. for crypto primitives)
- **The program does not use too much time**
- The program does not use too much space, network bandwidth...

Formalizing Asymptotic Complexity Claims via Deductive Program Verification

Asymptotic time guarantees

We're interested in high-level time analysis

Asymptotic time guarantees

We're interested in high-level time analysis

- Not "binary search terminates in less than 5ms"

Asymptotic time guarantees

We're interested in high-level time analysis

- Not "binary search terminates in less than 5ms"
- Rather "binary search runs in $O(\log n)$ steps"

Typical paper proofs rely on informal reasoning principles – which can easily be abused

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

Flawed proof:

`bsearch arr v i j` costs $O(1)$.

Typical paper proofs rely on informal reasoning principles – which can easily be abused

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

Flawed proof:

`bsearch arr v i j` costs $O(1)$.

By induction on $j - i$:

Typical paper proofs rely on informal reasoning principles – which can easily be abused

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

Flawed proof:

`bsearch arr v i j` costs $O(1)$.

By induction on $j - i$:

- $j - i \leq 0$: line 2 is $O(1)$. OK!

Typical paper proofs rely on informal reasoning principles – which can easily be abused

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

Flawed proof:

`bsearch arr v i j` costs $O(1)$.

By induction on $j - i$:

- $j - i \leq 0$: line 2 is $O(1)$. OK!
- $j - i > 0$: $O(1)$ (l.3-5) $+ O(1)$ (l.6) $+ O(1)$ (l.8) $= O(1)$. OK!

Typical paper proofs rely on informal reasoning principles – which can easily be abused

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

Flawed proof:

`bsearch arr v i j` costs $O(1)$.

(actual cost: $O(\log(j - i))$)

“By induction on $j - i$” …but which statement are we proving?

Typical paper proofs rely on informal reasoning principles – which can easily be abused

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

Flawed proof:

`bsearch arr v i j` costs $O(1)$.

(actual cost: $O(\log(j-i))$)

"By induction on $j - i$" …but which statement are we proving?

$\forall n,\ \exists c,$" `bsearch` costs c" $\neq$ $\exists c,\ \forall n,$" `bsearch` costs c"

Reasoning about O in a proof assistant

Using a proof assistant steers us clear of these abuses...

but maybe also from the simplicity of paper proofs.

Formally, what are we trying to prove?

"`bsearch arr v i j` runs in $O(\log(j-i))$ steps."

Formally, what are we trying to prove?

“`bsearch arr v i j` runs in $O(\log(j-i))$ steps.”

Formally, what are we trying to prove?

"there exists a cost function $f \in O(\log n)$ such that
for every `arr`, `v`, `i`, `j`,
`bsearch arr v i j` runs in at most $f(j - i)$ steps."

Formally, what are we trying to prove?

"there exists a cost function $f \in O(\log n)$ such that
for every `arr`, `v`, `i`, `j`,
`bsearch arr v i j` runs in at most $f(j - i)$ steps."

First step of the proof: exhibit a concrete cost function?

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

Concrete cost function?

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

Concrete cost function? $2\log(j-i)+1$?

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

Concrete cost function? $2\log(j-i)+1$? $3\log(j-i)+4$?

Our approach to this problem

- Interactive proofs (using Coq)

Our approach to this problem

- Interactive proofs (using Coq)
- Convince Coq to postpone the moment where the concrete cost function is provided

Our approach to this problem

- Interactive proofs (using Coq)
- Convince Coq to postpone the moment where the concrete cost function is provided
- Start proving the program and its invariants without knowing the cost function

Our approach to this problem

- Interactive proofs (using Coq)
- Convince Coq to postpone the moment where the concrete cost function is provided
- Start proving the program and its invariants without knowing the cost function
- At the same time, infer the cost function from the code of the program

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

```
cost (j-i) = 1 + …
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

To finish the proof

Solve this equation, and prove that $cost(n)$ is $O(\log n)$: by hand, or using the "Master theorem".

Conclusion

Machine-checked proofs of the asymptotic complexity of programs.

- Implemented as a Coq library, to verify OCaml programs
- Similar approach implemented in Isabelle at TUM (Munich)
- The approach could be applied to other languages (eg. C, C++, Java)