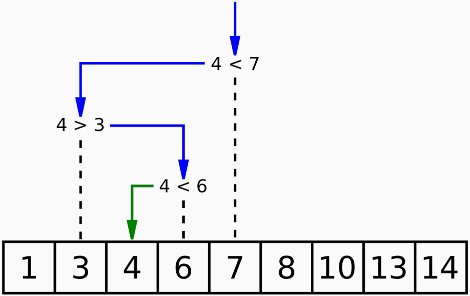
A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification

Armaël Guéneau, Arthur Charguéraud, François Pottier

Inria

Motivational example: binary search



Claim: “binary search finds an element in time $O(\log n)$”

Goal: formalize this claim in Coq for a concrete implementation

Functional correctness

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

- We can test this program
- We can prove functional correctness (Why3, CFML, ...)

Functional correctness

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

- We can test this program
- We can prove functional correctness (Why3, CFML, ...)

Yet, there is a bug

```
(* search for v in the range [i, j) *)
let rec bsearch (a: int array) v i j =
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if v = a.(k) then k
    else if v < a.(k) then bsearch a v i k
    else bsearch a v (i+1) j
```

Can you spot the bug?

Yet, there is a bug

```
(* search for v in the range [i, j) *)
let rec bsearch (a: int array) v i j =
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if v = a.(k) then k
    else if v < a.(k) then bsearch a v i k
    else bsearch a v (i+1) j
```

Can you spot the complexity bug?

Yet, there is a bug

```
(* search for v in the range [i, j) *)
let rec bsearch (a: int array) v i j =
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if v = a.(k) then k
    else if v < a.(k) then bsearch a v i k
    else bsearch a v (i+1) j
```

buggy, should be k+1

Can you spot the complexity bug?

In this talk

Goal: prove OCaml programs, including their asymptotic complexity expressed with $O()$ bounds

State of the art:

- Automatic inference for polynomial bounds
- Interactive proofs using *time credits*, e.g. "`bsearch` costs $3 \log n + 4$"

Issue: conciseness, and modularity of specifications

In this talk (2)

Solution: introduce the $O()$ notation for conciseness and modularity

Challenges:

- How to write specifications?
- What is the meaning of $O()$ in the multivariate case?
- How to do proofs (paper proofs are too informal)?
- How to automate the cost analysis?

Separation Logic with Time Credits

Time Credits: resources in separation logic

- Each function call (or loop iteration) consumes $\$1$
- $\$n$ asserts the ownership of n time credits
- $\$(n + m) = \$n * \$m$
- Credits are not duplicable: $\$1 \not\Rightarrow \$1 * \$1$
- Enables amortized analysis

References:

- Atkey (2011): time credits in Separation Logic
- Charguéraud & Pottier (2015): practical verification framework (CFML), applied to Union-Find

Example of using time credits

A specification *of the complexity* of `bsearch`:

$$\forall i\ j\ a\ v.$$
$$\{\$(3\log(j-i)+4) * \ldots\}\ \texttt{(bsearch a v i j)}\ \{\ldots\}$$

Example of using time credits

A specification *of the complexity* of `bsearch`:

$$\forall i\ j\ a\ v.$$
$$\{\$(3\log(j-i)+4) * \ldots\}\ \texttt{(bsearch a v i j)}\ \{\ldots\}$$

- Conciseness issue: even non dominant terms must appear

Example of using time credits

A specification *of the complexity* of `bsearch`:

$$\forall i\, j\, a\, v.\quad \{\$(3\log(j-i)+4) * \ldots\}\ \texttt{(bsearch a v i j)}\ \{\ldots\}$$

- Conciseness issue: even non dominant terms must appear
- Modularity issue: changing (even slightly) `bsearch` requires updating the specification, and all proofs that depend on it.

Example of using time credits

A specification *of the complexity* of `bsearch`:

$$\forall i\ j\ a\ v.\quad \{\$(3\log(j-i)+4) * \ldots\}\ \texttt{(bsearch a v i j)}\ \{\ldots\}$$

- Conciseness issue: even non dominant terms must appear
- Modularity issue: changing (even slightly) `bsearch` requires updating the specification, and all proofs that depend on it.
- Tempting: $\{\$O(log(j-i)) * \ldots\}$ `(bsearch a v i j)` $\{\ldots\}$

Challenges in reasoning with O

Informal reasoning principles can be abused

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

“Claim”:
`bsearch a v i j` costs $O(1)$.

Informal reasoning principles can be abused

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

"Claim":
`bsearch a v i j` costs $O(1)$.

Proof:

Informal reasoning principles can be abused

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

“Claim”:
`bsearch a v i j` costs $O(1)$.

Proof:
By induction on $j - i$:

Informal reasoning principles can be abused

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

"Claim":
`bsearch a v i j` costs $O(1)$.

Proof:
By induction on $j - i$:

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

Informal reasoning principles can be abused

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

"Claim":
`bsearch a v i j` costs $O(1)$.

Proof:
By induction on $j - i$:

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

Informal reasoning principles can be abused

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

"Claim":
`bsearch a v i j` costs $O(1)$.

Proof:

By induction on $j - i$:

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

Where is the mistake?

Informal reasoning principles can be abused

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

"Claim":
`bsearch a v i j` costs $O(1)$.

Proof:

By induction on $j - i$: ...but which statement are we proving?

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

Meaning of $O(1)$

What we just proved:

$\forall i\, j,\ \exists c,$ `bsearch a v i j` runs in c steps

Meaning of $O(1)$

What we just proved:

$\forall i\,j,\ \exists c,$ `bsearch a v i j` runs in c steps

What "$O(1)$" means:

$\exists c,\ \forall i\,j,$ `bsearch a v i j` runs in c steps

Meaning of $O(\log n)$

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

Meaning of $O(\log n)$

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

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

Meaning of $O(\log n)$

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

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

"**there exists** a cost function $f \in O(\log n)$ such that,
for every `a`, `v`, `i`, `j`,
$\{\$f(j-i) * \ldots\}$ `(bsearch a v i j)` $\{\ldots\}$".

Meaning of $O(\log n)$

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

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

"**there exists** a cost function $f \in O(\log n)$ such that,
for every a, v, i, j,
$\{\$f(j-i) * \ldots\}$ `(bsearch a v i j)` $\{\ldots\}$".

- Meaning of "$f \in O(g)$"?
- How to provide a witness for f?

A generic definition of O

Definition of O

- Single variable case:

$$f \in O(g) \quad \equiv \quad \exists c,\ \exists n_0,\ \forall n \geq n_0,\ |f(n)| \leq c\,|g(n)|$$

  with f of type $\mathbb{N} \to \mathbb{Z}$

- Multivariate case: f of type $\mathbb{N}^k \to \mathbb{Z}$
- In our library: f of type $A \to \mathbb{Z}$, with a *filter* on type A

O as a relation between functions

We define O as a *domination* pre-order between functions of A to $\mathbb{Z}$:

$$f \preceq_A g \quad \equiv \quad \exists c.\ \mathbb{U}_A\, x.\ |f(x)| \leq c\,|g(x)|$$

A must be equipped with a filter $\mathbb{U}_A$

O as a relation between functions

We define O as a *domination* pre-order between functions of A to $\mathbb{Z}$:

$$f \preceq_A g \quad \equiv \quad \exists c.\ \mathbb{U}_A\, x.\ |f(x)| \leq c\, |g(x)|$$

A must be equipped with a filter $\mathbb{U}_A$

- "$\mathbb{U}_A x.P$": "ultimately P" / "P holds of every *sufficiently large* x"
- Can be thought of as a quantifier

O as a relation between functions

We define O as a *domination* pre-order between functions of A to $\mathbb{Z}$:

$$f \preceq_A g \quad \equiv \quad \exists c.\ \mathbb{U}_A\, x.\ |f(x)| \leq c\,|g(x)|$$

A must be equipped with a filter $\mathbb{U}_A$

- "$\mathbb{U}_A x.P$": "ultimately P" / "P holds of every *sufficiently large* x"
- Can be thought of as a quantifier
- A standard notion in math (see e.g. Bourbaki)
- We prove in our library many properties of $\preceq_A$ for an arbitrary filtered type A

Proving specifications: automatic (guided) cost synthesis

Providing the cost function

"**there exists** a cost function $f \in O(\log n)$ such that,
for every `a`, `v`, `i`, `j`,
$\{\$f(j-i) * \ldots\}$ `(bsearch a v i j)` $\{\ldots\}$".

becomes

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

- First step of the proof: exhibit a concrete cost function. Guess "$\lambda n.\ 3 \log n + 4$" from the start?
- It seems desirable to (semi) automatically construct the witness as the proof progresses.

Our approach to this problem

- Convince Coq to postpone the moment where the concrete cost function is provided

Our approach to this problem

- Convince Coq to postpone the moment where the concrete cost function is provided

- Progressively synthesize the cost function while applying the reasoning rules from separation logic

Our approach to this problem

- Convince Coq to postpone the moment where the concrete cost function is provided
- Progressively synthesize the cost function while applying the reasoning rules from separation logic
- The synthesized function has the same structure as the code

Our approach to this problem

- Convince Coq to postpone the moment where the concrete cost function is provided
- Progressively synthesize the cost function while applying the reasoning rules from separation logic
- The synthesized function has the same structure as the code
- Afterwards, prove a $O()$ bound for the cost function

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

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

```
f (j-i) := 1 + …
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

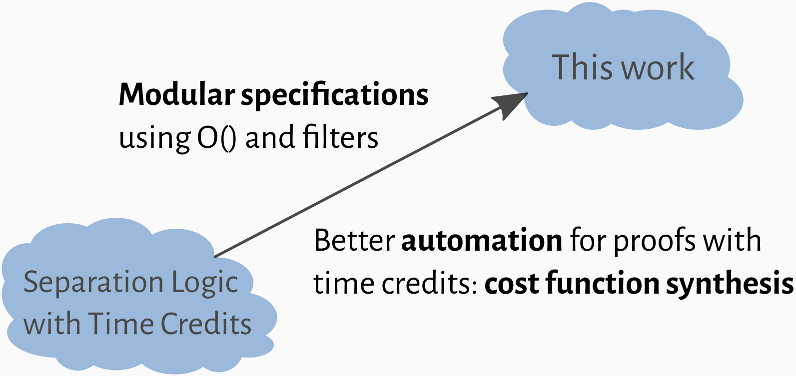
```
if j <= i then -1 else
  let k = i + (j - i) / 2 in
  if v = Array.get a k then k
  else if v < Array.get a k then
     bsearch a v i k
  else
     bsearch a v (k+1) j
```

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

Our cost synthesis achieves the following objectives:

- The user inspects the code only once
- The user can guide the synthesis of the cost function

Summary


This work
Modular specifications
using O() and filters
Separation Logic
with Time Credits
Better **automation** for proofs with
time credits: **cost function synthesis**

Closely related work

- **Howell** (2008), in his book, studies properties and difficulties of $O()$ with multiple variables.
- In Isabelle/HOL: **Zhan & Haslbeck** (2018) implement the same formal framework, with strong focus on automation but no "cost function synthesis". They build on **Eberl**'s (2017) impressive formalization of the Akra-Bazzi theorem.
- **Hoffmann et al.** (2010-2017): automated amortized resource analysis for OCaml. Implemented by **Carbonneaux, Hoffmann & Shao** (2015) with proof certificates checked by Coq.

More in the paper:

- Details about side-conditions for cost functions: monotonic and non-negative
- Clear up some confusion about multivariate $O()$
- Variable substitution in multivariate specifications
- Other case studies: selection sort, Bellman-Ford, Union-Find

`http://gallium.inria.fr/~agueneau/bigO`

Challenging case studies in the works!