Formal Complexity Verification

Shayan Pardis

12 December 2023

1 Motivation

Time complexity is one of the most fundamental concepts in algorithms. Having a tool that can automatically analyze and reason about time complexity would be valuable for many reasons.
First, this can identify overlooked time-complexity bottlenecks in large code bases and prevent human errors (e.g. can be used in CI/CD pipelines).
Second, we can use the same idea in synthesis to synthesize functions with an upper bound on time complexity. This idea will not only help prun the search tree but will also guarantee the performance of the synthesized function. Not surprisingly, solving this problem in general is undecidable (it can be used to solve the Halting problem). However, humans can easily deduce the complexity of simple programs such as the example below.

```
tick = 0
for i in range(n):
  for j in range(m):
    tick += 1
```

However, trying to solve this problem using Hoare triple and verification conditions and synthesizing the loop invariants using Sygus is simply not tractable because (1) loop invariants are "deep" and (2) there are two loop invariants that depend on each other and synthesizer should synthesize both of them together.
This is the loop invariant that the synthesizer has to come up with:

$$I_0 : tick_0 - i_0 \times m = tick - i \times m$$

$$I_1 : tick_0 - i_0 = tick - i$$

This suggests that we need a better framework to tackle this problem that is intractable using the current framework.

2 Related Work

ReSynKnoth et al. [2020] This work provides a framework to reason about resource allocation in data structures. They augment each type with a potential. This potential is used to compensate for the cost of accessing the data structure. There are two main differences between ReSyn and this project. First, ReSyn focuses on data structure but our focus recursive functions whose inputs are can even be one integer and involve mathematical arithmatics. Second, ReSyn synthesizes a program given the complexity. However, our goal is to verify the complexity of a given program.

3 Defining the problem

3.1 Choice of the Language

Choosing the language in which we solve the complexity verification problem can be critical as it defines the scope of the problem. Here, we use a simple functional language whose syntax is similar to Python and that supports (1) function definition (2) function call (3) if-else statements (4) simple arithmetic on numbers.
the limitation of this language is (1) there is no loop (2) variables are all integers (3) functions do not return anything.
We define the time complexity to be the complexity of the number of function calls.
We can observe that this can be sufficient to express a wide class of programs that we are interested in exploring.
For example merge sort can be expressed as below:

```
def merge(n):
  if n <= 0:
    pass
  merge(n-1)

def sort(n):
  if n <= 1:
    pass
  sort(n/2)
  sort(n/2)
  merge(n)
```

and two nested loops can be expressed as:

```
def loop1(n):
  if n <= 0:
    pass
  loop1(n-1)

def loop2(n, m):
  if n <= 0:
    pass
  loop2(n-1, m)
  loop1(m)
```

This language brings more structure to the problem that we are trying to solve and thus will make it more tractable.

3.2 Formalization of the task

Let $\mathbb{F}$ be the space of functions we are interested in and let $\mathbb{T}$ be the space of time complexity functions of members of $\mathbb{F}$, and let $\mathbb{T}_A = \{cT | T \in \mathbb{T}\}$
Let our environment be $\sigma : \mathbb{F} \to \mathbb{T}_A$ which maps each function to its time complexity.
We define $T_1 \preceq T_2$ to h $\exists_{n_0} \forall_{n > n_0} T_1(n) \leq T_2(n)$
Define $\mathcal{I}_\sigma : \mathbb{F} \to \mathbb{T}_A$ be the induction step. More specifically $\mathcal{I}_\sigma(f) = T$ means that if we calculate the complexity of function $f \in \mathbb{F}$ recursively using the time complexities that we have in environment σ the result would be $T \in \mathbb{T}$
Our problem is given functions $f_1, ..., f_n$ find σ such that $\forall_i \mathcal{I}_\sigma(f_i)) \preceq \sigma(f_i)$
In simple terms, propose a time function $\sigma(f)$ for each function f that can be proven using the induction step.
For example for the merge sort example that we had before:

$$\mathcal{I}_\sigma(merge) = ite(n \leq 0, 0, 1 + \sigma_{merge}(n-1))$$

$$\mathcal{I}_\sigma(sort) = ite(n \leq 1, 0, 2 + 2\sigma_{sort}(\frac{n}{2}) + 1 + \sigma_{merge}(n))$$

4 Methodology

4.1 verifying σ

The first step to solving this problem is being able to verify whether for a given σ the expression $\forall_i \mathcal{I}_\sigma(f_i) \preceq \sigma(f_i)$ holds.
To achieve this we have to use the derivation rules below to translate each program to its characteristic induction step $I_\sigma(f)$

- $V_\sigma(if(e, f_{true}, f_{false})) = ite(e, V_\sigma(f_{true}), V_\sigma(f_{false}))$
- $V_\sigma(f_1; f_2) = V_\sigma(f_1) + V_\sigma(f_2)$
- $V_\sigma(f; f \in \sigma) = 1 + \sigma(f)$

Notice the +1 in the last line which is to consider the cost of calling function f itself.
After translating the program to $\mathcal{I}_\sigma$ then we use CVC5 to verify the expression below:

$$ACC_\sigma = \exists_{n_0} \forall_i \forall_{input > n_0)} \mathcal{I}_\sigma(f_i) \leq \sigma(f_i)$$

4.2 verifying σ given all complexity classes

Now assume that we don't know exact σ but we know it up to a constant factor. In other words we know $\sigma' : \mathbb{F} \to \mathbb{T}$ (notice the difference between $\mathbb{T}, \mathbb{T}_A$)
for coefficients $c_1, ..., c_n$ define $\sigma(f_i) = c_i \sigma'(f_i)$
then we just check $\exists_{c_1,...,c_n} ACC_\sigma$
This can be done using CVC5 similar to above.

4.3 inferring σ

The problem becomes more interesting if the complexity of all functions is not given beforehand and the program needs to infer the ones that are not given.
Although we can limit the set of functions in $\mathbb{T}$ to simple cases (e.g. linear, quadratic, combination of inputs) the synthesis problem becomes exponentially harder as the number of functions increases.
More specifically the synthesis problem is searching over the space of $\mathbb{T}^n$ given that we have n functions. Additionally, if the verification for σ fails, without any feedback from the verifier, the synthesizer essentially has to blindly search over the space of $\mathbb{T}^n$ (e.g. we cannot do counter example-based synthesis).

4.3.1 zeroth approach zeroth approach: using sketch

failed because sketch synthesizes the constants based on a small set of examples and not for all possible inputs. As a result, sketch is always able to pick large enough constants that satisfy our complexity problem formulation for small inputs.

4.3.2 first approach: utilizing sink nodes

If there is a function f such that f is just calling itself and no other function then we can first calculate the time complexity of f independent of others. This suggests the approach below:
Consider the dependency graph of function calls. Topologically sort the nodes and start solving from source nodes. We also have to slightly modify the verification problem to verify Θ , not O. Otherwise, this approach will overestimate the time complexities.
This approach helps aggressively prune the search space however this does not solve the full problem where we have loops in the dependency graph of function calls (e.g. two functions that call each other recursively).

4.3.3 second approach: iterate and find the fixed point

Lets define $E : (\mathbb{F} \to \mathbb{T}_A) \to (\mathbb{F} \to \mathbb{T}_A)$ to be an operator that takes in environment σ_0 and outputs environment σ_1 such that $\forall_i \sigma_1(f_i) = \mathcal{I}_{\sigma_0}(f_i)$
This is similar to the fix point problem because it is as if we are trying to find a σ which is fix point of E (this is not a rigorous statement). However, the idea of setting $\sigma_{new} = E(\sigma_{old})$ iteratively and hoping that it will converge to the answer fails. We make a small modification. Instead of changing σ to $E(\sigma)$ all at once, at each step we only change one member of σ
We observe that $\mathcal{I}$ operator has the nice property of being increasing meaning that if $\forall_f\ \sigma_0(f) \preceq \sigma_1(f)$ then $\forall_f\ \mathcal{I}_{\sigma_0}(f) \preceq \mathcal{I}_{\sigma_1}(f)$
Thus let the answer to the problem be $\sigma^\star$ and we start σ such that $\forall_f\ \sigma_0(f) \preceq \sigma_\star(f)$ then if at each step we increase $\sigma(f)$ (for a given f) the minimum possible that is needed, we will never exceed $\sigma_\star$
Example:

```python
def f(n):
  if n <= 0:
    pass
  g(n-1)

def g(n):
  if n <= 0:
    pass
  f(n-1)
```

Here we have:
$\mathcal{I}_\sigma(f) = 1 + ite(n \le 0, 0, \sigma_g(n-1))$
$\mathcal{I}_\sigma(g) = 1 + ite(n \le 0, 0, \sigma_f(n-1))$
for simplicity ignore $n \le 0$ because we only consider values of $n \ge n_0$

- iteration 0:

  start with all complexity functions to be $O(1)$

  $\sigma_f = c_f, \sigma_g = c_g$

- iteration 1: fix σ_g and change σ_f

  $1 + c_g \le \sigma_f(n)$
  $1 + \sigma_f(n-1) \le c_g$

  simplify:

  $2 + \sigma_f(n-1) \le \sigma_f(n)$

  the lower bound candidate is $\sigma_f(n) \in O(n)$ thus we set $\sigma_f(n) = c_f n$

- iteration 2: fix σ_f and change σ_g

  $1 + \sigma_g(n-1) \leq c_f n$
  $1 + c_f(n-1) \leq \sigma_g(n)$

  simplify:

  $1 + \sigma_g(n-1) \leq \sigma_g(n) + c_f - 1$

  The lower bound candidate is $\sigma_g(n) \in O(n)$ thus we set $\sigma_g(n) = c_g n$

- iteration 3:

  SMT-Solver validates the σ_f, σ_g

This approach eliminates the need to synthesize the complexity functions of the whole σ at the same time thus eliminating the need to search over the space of $\mathbb{T}^n$
Note that the simplification step is very important since without eliminating all other constants that we don't use, the inequalities will be infeasible.

4.3.4 finding the lower bound complexity function

The fix-point approach requires us to have a way to find the lower bound of $\sigma(f)$ for a given f that would satisfy a set of inequalities. Our initial proposal involves defining a POSET over $\mathbb{T}$ and doing a lattice search over that space. At each step maintain t_{lb}, t_{ub} such that t_{ub} satisfies the inequality and t_{lb} doesn't. Pick a random t such that $t_{lb} \prec t \prec t_{ub}$ and then based on the response from the SMT-Solver whether set $t_{lb} := t$ or $t_{ub} := t$
However, there is a problem with this approach. In case that verifier rejects t we cannot set $t_{lb} := t$. Consider the example below:

$$\begin{tikzcd}
\boxed{t_{lb} = O(1)} \arrow[r] \arrow[dr] & \boxed{O(n)} \arrow[r] & \boxed{t = O(n^2)} \arrow[d] \\
& \boxed{answer = O(m)} \arrow[r] & \boxed{t_{ub} = O(n^2m^2)}
\end{tikzcd}$$

Here in this example, if $t = O(n^2)$ fails we cannot set it to t_{lb} because we have not explored all the branches.

5 implementation

The implementation is available in https://github.com/Shayan-P/Formal-Complexity-Verification. We have implemented a parser over python code limited by the constraints mentioned in the language section. Each function should be annotated with its complexity class and verification using SMT-Solver verifies whether the given annotations can correctly be the time complexity of the functions.

6 Limitation

- The current version is limited to only verifying the annotated time complexities since we did not find an effective method for the mentioned inference problem.
- The approaches mentioned above are limited to a set of known and commonly used complexity functions since synthesizing arbitrary mathematical functions is difficult.

- The theorems available in the SMT-Solver is an important limiting factor. Some very effective solvers handle linear theorems (LIA) but for example, reasoning about exponentials or logarithms is a much harder task. One of the functions that we were targeting to be able to analyze using this framework was GCD function which has logarithmic time and requires extra knowledge about module operation. But it turns out to be a very challenging task as well/
- The fix point idea mentioned above addresses the issue of synthesizing multiple invariants at the same time. However, synthesizing one single invariant can also be challenging depending on the problem. However, addressing this issue was not within the scope of this project.

References

T. Knoth, D. Wang, A. Reynolds, J. Hoffmann, and N. Polikarpova. Liquid resource types, 2020.