1 Feasibility of Mathematical Proofs

The concept of *feasibility* in computation has been extensively studied in the past century of the development of theoretical computer science. It is generally accepted that a problem is feasible if and only if there is a polynomial-time algorithm that solves the problem.

At least by the TCS community

One of the most important reasons to work with this definition is that it is independent of the exact model of computation; it does not matter whether you are considering single-tape Turing machines, multi-tape Turing machines, RAMs, or any standard programming language you like. Indeed, it is conjectured (known as the *extended Church-Turing thesis*) that the informal notion of feasible computation is exactly captured by the class P, or polynomial-time Turing machines [Coo90].

An exception is that quantum computers may be stronger.

In this section, we will try to informally understand the notion of *feasible mathematical proofs* by considering several examples, which will lead us to a fruitful but less well-known research program called bounded arithmetic.

1.1 Constructivism Tradition of Feasible Mathematics

According to the original paper of Cook [Coo75], one motivation of introducing the notion of feasible proofs is from the tradition of constructive mathematics, or constructivism in the philosophy of mathematics.

Another motivation (that is intentionally omitted) is to prove $\mathsf{P} \neq \mathsf{NP}$.

Rather than viewing mathematics as a "World of Ideas" of Plato that is eternal, unchanging, and independent of human observation, constructive mathematicians assert that mathematics is all about the objects that can be constructed by human intelligence. We quote from the book of Bishop [Bis67]:

"Platonism"

> "The positive integers and their arithmetic are presupposed by the very nature of our intelligence and, we are tempted to believe, by the very nature of intelligence in general. The development of the positive integers from the primitive concept of the unit, the concept of adjoining a unit, and the process of mathematical induction carries complete conviction. In the words of Kronecker, the positive integers were created by God."

Here we provide two examples to explain the meaning of viewing mathematics as a construction of human intelligence.

These are the BHK interpretation of quantifiers, named after Brouwer, Heyting, and Kolmogorov.

1. The existence quantifier should be interpreted as a *procedure* that effectively constructs the quantified object, instead of merely its "existence" as an object in the mathematics "World of Ideas". In particular, if $\varphi(x, y)$ is a formula such that $\forall x\ \exists y\ \varphi(x, y)$ is provable, there must be a *procedure* that (given n that substitutes x) effectively construct an m that substitutes y as well as a *proof* of $\varphi(n, m)$.

2. A *proof* of a universally quantified formula $\forall x\ \varphi(x)$ must provides a *procedure* that effectively generates a *proof* of $\varphi(n)$ given any n that substitutes the quantified variable x.

The interpretation of "effective procedures" depends on your understanding of human intelligence, and is certainly controversial. For instance, intuitionists believe that the law of excluded middle $\varphi \vee \neg\varphi$ is troublesome and should be excluded (see [BPI22] for a comprehensive survey on variants of constructivism).

As computer scientists, however, it is natural to imagine that the effective procedures are just feasible algorithms, or, under the extended Church-Turing thesis, polynomial-time algorithms. This yields the very basic notion of *feasible mathematics*, or *feasibly constructive mathematics*, as an interpretation of constructive mathematics by treating the "effective procedures" as polynomial-time algorithms. In such case, we are assuming a very minimum mean of intelligence, or the intelligence of a humble computer scientist [Dij72] that is skeptical of anything that cannot be constructed in polynomial time.

As the adversary model in modern cryptography.

Example 1.1. Under this interpretation, the exponential function cannot be talked about by the computer scientist; that is, the sentence $\forall x\ \exists y\ \log_2(y) = x$ is not provable. This is because the function that outputs y given x is not a polynomial-time function (in its input length).

Example 1.2. Similarly, the following version of the pigeonhole principle is unlikely to be trusted by the computer scientist: For every function $f : \{0,1\}^n \to \{0,1\}^{n-1}$ (that can be talked about), there are distinct strings $x, y \in \{0,1\}^n$ such that $f(x) = f(y)$. Suppose, towards a contradiction, that such principle is considered true, then there must be a polynomial-time algorithm that given a function f outputs $x \neq y$ such that $f(x) = f(y)$. This breaks a widely believed cryptographic primitive: collision-resistant hash function.

The two examples above seem problematic: As a cost of being humble, the computer scientist, or, the *feasible mathematician*, is unable to even define an exponentially long string, or understanding the pigeonhole principle.

This seems like a slippery slope for those who are skeptical about working with (say) ZFC set theory and the existence of large cardinals.

It turns out that feasible mathematics is "strong and weak": Though it cannot prove the pigeonhole principle, it is known to prove the PCP theorem [Pic15] and many known complexity lower bounds (see, e.g., the table in [PS21]). One exception is that the $\Omega(n^2)$ lower bound for deciding Palindrome on single-tape Turing machines [CLO24] is unlikely to be feasibly provable, as it requires a form of pigeonhole principle.

Another thing that kind of bothers us is that the issue appears right after we interpret "the effective procedure", without even identifying what are "the proofs" that the effective procedures are manipulating. As computer scientists, we have our definition of "proofs", or the complexity class NP, that requires two properties:

1. The length of the proof is polynomial in the length of the statement.
2. The proof is *verifiable* by a polynomial-time algorithm.

These two properties are necessary. After all, feasible mathematicians cannot construct exponentially long strings (see Example 1.1), and they do not trust verification procedure that is not a polynomial-time algorithm! The only thing we are not sure about is the meaning of *verifiable* in the second property, which we will explore from scratch in the next sub-section.

Indeed, we will spend three sections on exploring it :)

1.2 Informal Postulates of Feasibility

1.2.1 Postulate of Feasible Functions

Hoping to understand the exact meaning of feasible mathematics, we start by considering informally what functions, axioms, and inference rules should we be allowed to use when we are doing feasible mathematics.

Question 1.1. Is the addition of two numbers allowed to talk about when we are doing feasible mathematical proofs?

The answer to this question should certainly be *yes*. The justification is as follows. On the one hand, the addition of two numbers is clearly a feasible operation. On the other hand, there is not much mathematics we can do if we are not even allowed to talk about addition. Moreover, comparably simple functions, such as the multiplication of two numbers, the bit-length function $|\cdot|$, etc., should all be allowed to talk about when we are doing feasible mathematics.

Please consult a primary school student if you have any questions :)

Extending Question 1.1, we further ask:

Question 1.2. Should we be allowed to talk about every feasible function and algorithm for them?

Note that predicates (languages) are just Boolean-valued functions.

As computer scientists, we will certainly want to say *yes*; after all, no one wants to define a mathematical theory that cannot be used to talk about their favorite algorithm! However, we will have to note that this will be a technical challenge: there are algorithms that we do not know whether they run in polynomial time.

Example 1.3. It has been open for decades whether a *deterministic* polynomial-time algorithm could output an n-bit prime number given 1^n. The Cramér's conjecture, which is true under generalized Riemann hypothesis, implies that the following simple algorithm suffices:

- Starting from $N \leftarrow 2^n$, we call the AKS primality testing algorithm to check whether N is a prime number. If so, the algorithm halts and outputs N; otherwise, it sets $N \leftarrow N + 1$ and continues.

Although this algorithm (say, represented by a Turing machine) is likely to be feasible, we should refrain from using it in feasible mathematics as:

1. The Cramér's conjecture could be *wrong*, in which case we introduce an infeasible function in feasible mathematics.
2. The Cramér's conjecture could be *true* but *infeasible*. This could potentially be bad as our framework of feasible mathematics may no longer enjoy certain good properties that it is supposed to have.

The existence of an n-bit prime follows from the Bertrand-Chebyshev theorem.

Taking a step back, we should at least be allowed to talk about functions that are not only feasible, but also *clearly demonstrates* their feasibility through its construction. For instance, one would generally agree that functions constructed in the following ways clearly demonstrate their feasibility:

- The composition of feasible functions is still feasible. For any constant k, if $f(x_1, \ldots, x_k)$ and $g_1(y_1), \ldots, g_k(y_k)$ are feasible functions, $f(g_1(y_1), \ldots, g_k(y_k))$ is also a feasible function.

It is confusing here, but should be more clear in subsequent examples.

- For any constant c, any function $f_M(x)$ defined by a Turing machine $M(x)$ with a timer that makes the machine halt after $|x|^c$ steps is feasible.

Note that $x \mapsto |x|^c$ is feasible, as it is a composition of the bit-length function and multiplication.

- Suppose that f is feasible, the function $\sigma_f(n)$ that either outputs the smallest number $x < |n|$ such that $f(x) \neq 0$, or outputs $|n|$ if $\forall x < |n|\ f(x) = 0$, is a feasible function.

This leads to our first postulate of feasible mathematics:

Postulate 1. Feasible mathematics are allowed to talk about functions that are feasible and clearly demonstrate their feasibility through their constructions.

We stress that the exact meaning of the phrase "clearly demonstrates their feasibility" can be interpreted in different ways, which will probably lead to multiple formal definitions of feasible mathematics.

We will go back to this later.

Here we make a short remark. We say in Postulate 1 that functions with clear demonstrations of feasibility are allowed in feasible mathematics. To formalize meaningful mathematical statements, we also need to specify the *atomic predicates* that are allowed to use. The equality predicate ("="), for instance, should certainly be allowed. Moreover, for any predicate $P(\vec{x})$ (e.g. $\leq$, $\wedge$) that can be decidable by a feasible algorithm (that can clearly demonstrate their feasibility), we can define a function $g_P(\vec{x})$ that outputs 1 (resp. 0) when the property $P(\vec{x})$ holds (resp. does not hold), and formalize the predicate as $P(\vec{x})$ as $g_P(\vec{x}) = 1$. Therefore, without loss of generality, we can work with only one atomic predicate "=".

1.2.2 Postulate of Definition Axioms

Now we consider the axioms that should be allowed to be used in feasible mathematics, starting from the following question:

Question 1.3. Should we treat the definition axiom $n + (m + 1) = (n + m) + 1$ of addition as an axiom of feasible mathematics?

Well, in any case, we will certainly need the rule when we are doing interesting mathematics, and it looks pretty basic. Let's take it for granted at this point that this rule and comparably simple rules such as the definition axiom of multiplication (i.e. $n \times (m + 1) = n \times m + n$) can be used as axioms in feasible mathematics.

Moreover, as (by Postulate 1) we are allowed to work with functions that are far more complicated than addition and multiplication, it is natural to further include the definition axioms of those functions as the axioms. But what does it mean by "definition axioms"?

Recall that a feasible function f is allowed to be talked about only if it clearly demonstrates its feasibility through its constructions. This specific mean of "construction" should be a couple of mathematical facts that uniquely specify the function f; for instance, it could be a set of equations that inductively defines f (see Example 1.4), or the code of a Turing machine that computes f. Since we take it for granted that these mathematical facts "clearly demonstrate" the feasibility of f, these facts must be able to be used (i.e. be axioms) when we are doing mathematics about f — they are what we mean by "definition axioms" here.

"we" := feasible mathematicians.

Example 1.4. Consider a function f that is inductively defined as

$$f(0) := 0 \tag{1}$$
$$f(2n) := f(n) + 2n \tag{2}$$
$$f(2n+1) := f(n) + 2n + 1 \tag{3}$$

Arguably, this is a feasible function that clearly demonstrates its feasibility through this particular inductive definition. In this case, there should be no confusion that Equations (1) to (3) are the definition axioms of f.

Note that in each recursive call to evaluate f the length of the input is reduced by 1.

This leads to the second postulate of feasible mathematics.

Postulate 2. For any feasible function that clearly demonstrates its feasibility through its construction, the mathematical facts along with the particular construction that enables the "clear demonstration" are allowed to be used as axioms of feasible mathematics.

1.2.3 Postulate of Structural Induction

Do we need other axioms rather than the definition axioms to do meaningful mathematical reasoning? Let's imagine the simplest task of proving an inequality regarding function f we defined in Example 1.4: $\forall x\ f(x) \le 2x$. Since $f(x)$, $x \mapsto 2x$, and the comparison predicate "$\le$" are all feasible and can arguably demonstrate their feasibility under suitable definitions, this sentence is allowed to be talked about in feasible mathematics.

However, it is unclear how we should prove it only from their definition axioms. If we consider the last bit of n in its binary encoding, we need to prove that $f(0) \le 0$ (which is easy), and that

$$\forall x\ f(2x) \le 2 \cdot 2x, \quad \forall x\ f(2x+1) \le 2 \cdot (2x+1).$$

We can certainly apply the definition axiom of f to unfold them as:

$$\forall x\ f(x) + 2x \le 2 \cdot 2x, \quad \forall x\ f(x) + 2x + 1 \le 2 \cdot (2x+1).$$

Let's assume that feasible mathematicians know basic arithmetic, which seems like a minor assumption, it suffices to prove that

$$\forall x\ f(x) \le 2x, \quad \forall x\ f(x) \le 2x + 1.$$

Performing a case study on the last bit of x and applying the definition equation does not make much progress in proving the inequality $f(x) \le 2x$.

This hints that we will have to include another tool in mathematics to prove something interesting: the principle of mathematical induction. Concretely, we will need an induction on the length of x in its binary encoding and apply the definition axioms of f:

- (Base): $f(0) \le 2 \cdot 0$, which is true as $f(0) = 0$, $2 \cdot 0 = 0$, and $0 \le 0$.
- (Induction Case 1): $f(x) \le 2x \to f(2x) \le 2 \cdot 2x$. Assume that $f(x) \le 2x$, we will prove $f(2x) \le 2 \cdot 2x$. By the definition axiom (2), we know that $f(2x) = f(x) + 2x$, then

$$f(2n) = f(n) + 2n \le 2n + 2n = 2 \cdot 2n.$$

Exercise: feasibly prove $b \le c$ implies $b + a \le c + a$.

- (Induction Case 2): $f(x) \leq 2x \to f(2x+1) \leq 2 \cdot (2x+1)$. Assume that $f(x) \leq 2x$, we will prove $f(2x+1) \leq 2 \cdot (2x+1)$. By the definition axiom (3), we know that $f(2x+1) = f(x) + 2x + 1$, then

$$f(2x+1) = f(x) + 2x + 1 \leq 2x + (2x+1) \leq (2x+1) + (2x+1) = 2 \cdot (2x+1).$$

Cheers! By induction on the binary representation of the variable x, we proved the first meaningful theorem.

But should we accept induction in feasible mathematics? Take the proof above as an example. As we proved that $\forall x\ f(x) \leq 2x$, by the constructivism interpretation of the universal quantifier (see Section 1.1), there must be an efficient "procedure" that generates a verifiable "proof" of $f(x) \leq 2x$ given n. Moreover, by our interpretation of "feasible mathematics", the "procedure" must be a feasible (i.e. polynomial-time) algorithm, and the "proof" must be of polynomial length that is in some sense "verifiable" by a polynomial-time algorithm (i.e. an NP-style proof).

We claim that the answer is positive as long as the predicate for induction is "verifiable" in the sense that given an input m that substitutes the variable you are performing induction on, there is a feasible (i.e. polynomial-time) algorithm that verifies the predicate. In this example, the predicate is $f(x) \leq 2x$ and we are performing an induction on the variable x; given any m, we can certainly evaluate $f(x)$ and check whether it is smaller than $2x$ feasibly. In particular, if we work with the only predicate "=", this is equivalent to saying that the predicate for induction must be of form $g(x) = h(x)$ for functions g, h that clearly demonstrates their feasibility.

Exercise: The predicate for induction in the correctness proof of Dijkstra's algorithm is not verifiable.

Unfortunately, we are unable to provide a convincing clarification for this claim until we formally define the "proof" and the mean of "verifiability". Here is a possibly confusing informal explanation if you insist: Since the induction is performed over the length of the binary representation of x, the "chain of reasoning" is of length $\mathsf{poly}(|x|)$. Therefore, for any concrete input m that substitutes x, there is a convincing proof of length $\mathsf{poly}(|m|)$ that clearly demonstrates the fact "$f(m) \leq 2m$".

We will then give the third (and the last) postulate:

Postulate 3. For any functions g, h that clearly demonstrate its feasibility through its construction, induction on the binary representation of x to prove the statement $g(x) = h(x)$ is allowed in feasible mathematics.

Recall that we are working with only one predicate "=".

For the example above, one can think of $g(x)$ as the function checking whether "$f(x) \leq 2x$", and $h(x) = 1$.

Note that we also call the induction principle a "structural induction" as it is a structural induction over the definition of (the binary representation of) numbers.

For simplicity, we will stop using the long sentence "a function f that clearly demonstrates its feasibility through a particular construction". We will simply call f a "feasibly constructible function", and use the term the "feasible construction of f" to refer to the particular construction that demonstrates its feasibility.

1.3 An Example: the Induction Principle

To understand the power of feasible mathematics with the three postulates we have, we will consider whether the following induction principle can be feasibly proved:

Question 1.4. Suppose that f, g are functions that we are allowed to talk about (in particular, it is a feasible function), and $\varphi(n)$ is the sentence "$f(n) = g(n)$". Is the following induction rule, i.e.,

$$\varphi(0) \wedge \Big(\forall n\ (\varphi(n) \to \varphi(n+1))\Big) \to \forall n\ \varphi(n)$$

considered feasible?

We note that this induction principle is not formulated as the structural induction over a feasibly defined function. In particular, the "chain of reasoning" from $\varphi(0)$ and $\varphi(n) \to \varphi(n+1)$ to $\forall n\ \varphi(n)$ seems to be exponentially long. That is, for every n, we need to derive $\varphi(n)$ from:

$$\Big\{\varphi(0), \quad \varphi(0) \to \varphi(1), \quad \varphi(1) \to \varphi(2), \quad \ldots, \quad \varphi(n-1) \to \varphi(n)\Big\},$$

which consists of $n + 1 = |n|^{\omega(1)}$ formulas.

Here, we will demonstrate that this induction rule is indeed feasible.

Theorem 1.1 (informal)**.** *Let f, g be functions that clearly demonstrate their feasibility through constructions, and $\varphi(n) :=$ "$f(n) = g(n)$". It is feasible provable that*

$$\varphi(0) \wedge \Big(\forall n\ (\varphi(n) \to \varphi(n+1))\Big) \to \forall n\ \varphi(n). \tag{4}$$

Instead of considering Equation (4) directly, we will work with the following sentence that is logically equivalent to it:

Exercise: prove the logical equivalence.

$$\forall n\ \exists m\ (\varphi(0) \wedge \neg\varphi(n) \to \varphi(m) \wedge \neg\varphi(m+1)). \tag{5}$$

(Let's assume that feasible mathematicians are fine with changing a formula into a logically equivalent one.)

This is true as long as we stick to the standard first-order predicate logic to formulate feasible mathematics.

Recall that the constructivism interpretation requires that a proof of $\forall x\ \exists y\ \varphi(x, y)$ must carry a procedure that finds y such that $\varphi(x, y)$ given x as well as a proof of $\varphi(x, y)$. This means that we need to find a feasibly constructible function $h(n)$ and prove feasibly that

$$\forall n\ (\varphi(0) \wedge \neg\varphi(n) \to \varphi(h(n)) \wedge \neg\varphi(h(n)+1)). \tag{6}$$

What does it mean by proving Equation (6)? Viewing $n \mapsto h(n)$ as an algorithmic task, it means that given an n such that $\varphi(0)$ is true and $\varphi(n)$ is false, we need to find an m such that $\varphi(m)$ is true and $\varphi(m+1)$ is false. The algorithm should be feasibly constructible; in particular, it should run in time $\mathsf{poly}(|n|)$.

Indeed, it is not hard to see that a simple binary search algorithm works (see Algorithm 1).

Equivalently, one may implement the binary search by a recursive (instead of iterative) algorithm. In either case, the algorithm *clearly* runs in polynomial time through its construction, as the quantity $r - \ell$ is halved in each recursive call (iteration). Therefore the function (let's call it $h(n)$) computed by this algorithm is feasibly constructible.

It remains to demonstrate that Equation (6) is feasibly provable. The only tools we can use are two postulates: the postulate of definition axioms and the postulate

If this is not clear to you, think of the most natural way to prove it without worrying about the feasibility.

Input: n such that Equation (5) holds
Output: m such that $\varphi(m) \wedge \neg\varphi(m+1)$

```
ℓ ← 0, r ← n;
while ℓ + 1 < r do
    m ← ⌊(ℓ + r)/2⌋;
    if φ(m) is true then
        ℓ ← m
    else
        r ← m
    end
end
return ℓ
```

Algorithm 1: Binary Search for $n \mapsto h(n)$

of structural induction. Needless to say, we will need to perform an induction over the "while" loop on some properties regarding ℓ and r.

Arguably, the right way to perform induction over Algorithm 1, which realizes the informal discussion above that the length of the "chain of reasoning" needs to be a polynomial to the input length, is to prove a *loop invariant*: $\varphi(\ell) \wedge \neg\varphi(r) \wedge r > \ell$. This is true before the program enters the loop, and (by the definition axioms of the operations inside the loop) if the property holds in the i-th iteration, it also holds in the $(i+1)$-th iteration. Finally, when we leave the loop, we will have:

$$\varphi(\ell) \wedge \neg\varphi(r) \wedge r > \ell \wedge \ell + 1 \geq r$$

which implies immediately that $r = \ell+1$, and thus ℓ (the output $h(n)$ of Algorithm 1) satisfies $\varphi(\ell) \wedge \neg\varphi(\ell+1)$, i.e., Equation (6).

Equivalently, if we formulate the algorithm in a recursive manner, we can introduce a variable z indicating that we will perform $|z|$ iterations of the binary research, and prove by induction on the binary representation of z that the invariant holds.

We stress that there is one additional property to be checked by the feasible mathematician: the loop invariant (i.e. the property in structural induction) must be formalizable in the form $f(x) = g(x)$ for some feasibly constructible functions $f(x), g(x)$. Since both $\varphi(\ell)$, $\neg\varphi(r)$, and $r > \ell$ are pretty easy to be checked by feasibly constructible functions, we can easily construct some feasible $f(x)$ (where x encodes the pair (ℓ, r)) that checks the loop invariant, and formalize the loop invariant as $f(x) = 1$. All the rewinding (e.g. parse x as a pair and the evaluation of "$\varphi(\ell), \varphi(r), r > \ell$") should be feasibly provable from the definition axioms under suitable formalization. Therefore, we conclude (from the informal postulates) that Theorem 1.1 should be true.

1.4 A Technical Perspective: Applications of Feasible Mathematics

How can feasible mathematics (and mathematicians) be useful? Besides the philosophic satisfaction from proposing (and playing with) an interpretation of constructivism, we now provide an incomplete list of reasons for computer scientists to study feasible mathematics (or to be feasible mathematicians).

Unprovability of complexity conjectures. Complexity theory is blessed (or cursed) by many unproven conjectures (e.g. P versus NP, circuit lower bounds, BPP versus P) that seem intractable despite decades of efforts. Therefore, it is reasonable to ask whether our current toolkit is inherently unable to resolve these conjectures. Informal barriers such as relativization [BGS75] and natural proofs [RR97] are insightful and intuitive but do not consider a robust notion of proofs, which makes them (somewhat) limited in scope.

For instance, a relatively simple technique called hardness magnification (see, e.g., [CHO+22]) is not captured by natural proofs.

A more natural approach is to prove the unprovability (or even independence) of the conjectures from a well-defined mathematical theory. In this case, feasible mathematics serves as the "minimum" theory that we should consider: We should at least be able to define feasible functions to state the conjecture. Indeed, we can already show the unprovability of many upper and lower bounds from feasible mathematics; we refer interested readers to [Oli24, Section 5] and the references therein for more details.

Connection to (propositional) proof complexity. The notion of feasibility we introduced considers the complexity of mathematical proofs. Another measure, namely the length of proofs in *propositional proof systems*, is also interesting from both theoretical and practical perspectives. Theoretically, the question originates from the Cook-Reckhow program towards $\mathsf{P} \neq \mathsf{NP}$ by proving stronger and stronger lower bounds against propositional proof systems (see, e.g., [Kra19]). The length of propositional proofs is also related to the efficiency of SAT solvers that are widely used in practice, see, e.g., [BN21].

The relationship between our (i.e. Cook's [Coo75]) notion of feasibility and the length of propositional proofs is more or less similar to the relationship between Turing machines and circuits; our notion of feasible mathematics deals with (quantified) sentences such as $\forall x\ f(x) = g(x)$, while the length of propositional proofs is defined for formulas with a fixed number of variables (or families of formulas). Indeed, this connection will be made formal in subsequent sections (also see [Kra19]). This connection helps bridge results in these two lines of research and already yields fruitful results (see, e.g., [Ajt94, Kra01, Kra10, Kha22]).

Attacking open problems through logical insights. Sometimes the results from logic translate to useful techniques in theoretical computer science. Here, we mention two recent lines of results.

A line of work (see, e.g., [KKMP21, Kor21, Kor22]) studied a search problem called *range avoidance*. This problem is inspired by the combinatorial principle called "dual weak pigeonhole principle", namely there is no surjection from a small domain to a much larger range, that was extensively studied in the literature of bounded arithmetic (see, e.g., [CLO24] and the references therein). Many techniques that are natural in logic translate directly to complexity-theoretic results (see, e.g., [Kor22]). Moreover, the study of this problem leads to new unconditional circuit lower bounds [CHR24, Li24].

Indistinguishability obfuscation ($i\mathcal{O}$) [BGI+12] provides a formal definition of obfuscating programs while preserving the functionality. Intuitively, it means that we can obfuscate two functionally equivalent programs so that no bounded adversary can distinguish one from the other. Basing $i\mathcal{O}$ on cryptographic assumptions encounters a sub-exponential loss in security parameters as there is no efficient veri-

fication of functional equivalence. It turns out that if the functional equivalence can be *feasibly proved*, this sub-exponential loss can be avoided [JJ22]. (Subsequently, a similar idea is used to construct SNARGs [JKLV24] from LWE.)

1.5 A Philosophical Perspective: Feasible Mathematics, Programmers, and Computers

At the end of the section, I want to mention an interpretation of feasible mathematics as we postulated that, as far as I know, has not been discussed in the literature.

Our definition of feasible mathematics (which is intended to capture the intuition behind Cook's theory PV [Coo75]) is more or less the *minimum* theory to reason about polynomial-time algorithms. It only includes the definition axioms of the algorithms as well as the induction principle on the binary representation of numbers for predicates that can be effectively verified. (Recall that in the postulate of structural induction, the property must be an equation $f(x) = g(x)$ for feasibly constructive functions $f(x)$ and $g(x)$, and thus the equation can be verified efficiently given x.) If a property about the algorithm, e.g., its correctness, can be proved in feasible mathematics, it indicates that such a proof merely requires minimum power of reasoning, and in this sense, it is an "easy" property about the algorithm.

There is another sub-area of computer science — the research on programming languages — that particularly cares about proving the correctness of programs. In developing computer programs, a *specification* (either formally written or informally described) defines what is the intended behavior of the programs, and the programmers aim to *ensure* that the program satisfies the specification, i.e., it is *correct*. One of the main purposes of the design of modern programming languages is to make the programmers' lives easier, and in particular, it would be great if the correctness of the programs follows closely from the construction of the programs. I quote from E. W. Dijkstra's Turing award lecture *The Humble Programmer* [Dij72]:

> "The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand."

In particular, Dijkstra's famous "go to statement considered harmful" [Dij68] is supported by the fact that the correctness of a program with go to statements can not "grow hand in hand"; it does not follow immediately from the structure of the programs, and in fact, programs with go to statements barely have any structure that can be easily understood by humans. A program written in modern programming languages, instead, usually clearly demonstrates their feasibility and correctness.

I would like to imagine feasible mathematics as the logic of the humble programmers as described by Dijkstra: The correctness proof and program can grow hand in hand if and only if the correctness proof is feasible. Putting it more carefully, I conjecture that

> A statement is feasibly provable, if and only if it is possible to write a feasible (i.e. polynomial-time) program M in a suitable programming language such that

- the statement encodes the correctness of M in a natural sense;
- the correctness proof and M can "grow hand in hand" by a common programmer.

If this conjecture is true, it indicates that the past decades of education practice in programming are in fact teaching people to be *feasible mathematicians*, although it has never been explicitly mentioned by teachers and students.

There are two clarifications of the conjecture.

Firstly, the correctness of many classical algorithms is not known to be feasibly provable. In fact, Dijkstra's algorithm on single-source shortest path is not known to be feasibly provably correct, as the standard correctness proof involves induction on infeasible statements: At the end of the i-th step, if we updated the node u_i to have distance $d(u_i)$, there is no path from the source to u_i shorter than $d(u_i)$; and the fact that "there is no shorter path" can not be feasibly verified. This does not falsify the conjecture unless one thinks that the correctness of Dijkstra's algorithm is something that naturally comes to the mind of a common programmer the first time they wrote Dijkstra's algorithm. I would guess that one of the main reasons for people to think that Dijkstra's algorithm (and some other classical algorithms) is tricky and intellectually satisfying is that the proof is beyond the mind of common programmers, i.e., feasible mathematics.

If your assumption to "common programmers" is stronger than mine, maybe you could reformulate the conjecture with respect to a certain level of Buss's hierarchy, rather than Cook's PV.

Secondly, the conjecture does not mean that feasible mathematics is *easy*. For instance, we know that the PCP theorem (in a natural formalization) is provable in Cook's theory PV [Pic15] and thus is feasibly provable. However, it is obviously terrifying to think that the PCP theorem is easy. The only claim I made is that the feasible proof can "grow" while writing a feasible program, while the final program could be terribly hard and thus the proof terribly hard even if the statement we want to prove (i.e. the specification of the program) is simple. Computer programs (such as modern web browsers and operating systems) can certainly be terribly hard and the programs can be much more complicated compared to their specifications.

Although this conjecture is hard to verify, there are concrete tasks that may help to support it. I believe that the correctness of most programs, or at least most parts of most programs, can be formalized in a natural sense and proved in feasible theories such as PV [Coo75]. In particular, the correctness of compilers of structured programming languages that allow "correctness proof and program grow hand in hand", if formalized properly, should be provable in PV. In subsequent sections we will partially demonstrate why this could be true: Starting from Cook's definition of PV that supports few programming functionalities, we will show how to construct a much stronger "programming language" that supports (for instance) data structures much as *lists* and *maps*, and how the correctness proof in the strong programming language translates back to a correctness proof in PV.

References

[Ajt94] Miklós Ajtai. The complexity of the pigeonhole principle. *Comb.*, 14(4):417–433, 1994.

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in p. *Annals of mathematics*, pages 781–793, 2004.

[BGI+12] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6:1–6:48, 2012.

[BGS75] Theodore Baker, John Gill, and Robert Solovay. Relativizations of the p=?np question. *SIAM Journal on computing*, 4(4):431–442, 1975.

[Bis67] Errett Bishop. Foundations of constructive analysis. 1967.

[BN21] Sam Buss and Jakob Nordström. Proof complexity and SAT solving. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 233–350. IOS Press, 2021.

[BPI22] Douglas Bridges, Erik Palmgren, and Hajime Ishihara. Constructive Mathematics. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Fall 2022 edition, 2022.

[CHO+22] Lijie Chen, Shuichi Hirahara, Igor Carboni Oliveira, Ján Pich, Ninad Rajgopal, and Rahul Santhanam. Beyond natural proofs: Hardness magnification and locality. *J. ACM*, 69(4):25:1–25:49, 2022.

[CHR24] Lijie Chen, Shuichi Hirahara, and Hanlin Ren. Symmetric exponential time requires near-maximum circuit size. In Bojan Mohar, Igor Shinkar, and Ryan O'Donnell, editors, *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, Vancouver, BC, Canada, June 24-28, 2024*, pages 1990–1999. ACM, 2024.

[CLO24] Lijie Chen, Jiatu Li, and Igor Carboni Oliveira. Reverse mathematics of complexity lower bounds. *Electron. Colloquium Comput. Complex.*, pages TR24–060, 2024.

[Cob65] Alan Cobham. The intrinsic computational difficulty of functions. 1965.

[Coo75] Stephen A. Cook. Feasibly constructive proofs and the propositional calculus (preliminary version). In William C. Rounds, Nancy Martin, Jack W. Carlyle, and Michael A. Harrison, editors, *Proceedings of the 7th Annual ACM Symposium on Theory of Computing, May 5-7, 1975, Albuquerque, New Mexico, USA*, pages 83–97. ACM, 1975.

[Coo90] Stephen A Cook. *Computational complexity of higher type functions*. American Mathematical Society, 1990.

[Dij68] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968.

[Dij72] Edsger W Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[JJ22] Abhishek Jain and Zhengzhong Jin. Indistinguishability obfuscation via mathematical proofs of equivalence. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*, pages 1023–1034. IEEE, 2022.

[JKLV24] Zhengzhong Jin, Yael Kalai, Alex Lombardi, and Vinod Vaikuntanathan. Snargs under LWE via propositional proofs. In Bojan Mohar, Igor Shinkar, and Ryan O'Donnell, editors, *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, Vancouver, BC, Canada, June 24-28, 2024*, pages 1750–1757. ACM, 2024.

[Kha22] Erfan Khaniki. Nisan-wigderson generators in proof complexity: New lower bounds. In Shachar Lovett, editor, *37th Computational Complexity Conference, CCC 2022, July 20-23, 2022, Philadelphia, PA, USA*, volume 234 of *LIPIcs*, pages 17:1–17:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[KKMP21] Robert Kleinberg, Oliver Korten, Daniel Mitropolsky, and Christos H. Papadimitriou. Total functions in the polynomial hierarchy. In James R. Lee, editor, *12th Innovations in Theoretical Computer Science Conference, ITCS 2021, January 6-8, 2021, Virtual Conference*, volume 185 of *LIPIcs*, pages 44:1–44:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[Kor21] Oliver Korten. The hardest explicit construction. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 433–444. IEEE, 2021.

[Kor22] Oliver Korten. Derandomization from time-space tradeoffs. In Shachar Lovett, editor, *37th Computational Complexity Conference, CCC 2022, July 20-23, 2022, Philadelphia, PA, USA*, volume 234 of *LIPIcs*, pages 37:1–37:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[Kra01] Jan Krajíček. On the weak pigeonhole principle. *Fundamenta Mathematicae*, 1(170):123–140, 2001.

[Kra10] Jan Krajícek. A form of feasible interpolation for constant depth frege systems. *J. Symb. Log.*, 75(2):774–784, 2010.

[Kra19] Jan Krajíček. *Proof complexity*, volume 170. Cambridge University Press, 2019.

[Li24] Zeyong Li. Symmetric exponential time requires near-maximum circuit size: Simplified, truly uniform. In Bojan Mohar, Igor Shinkar, and Ryan O'Donnell, editors, *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, Vancouver, BC, Canada, June 24-28, 2024*, pages 2000–2007. ACM, 2024.

[Oli24] Igor Carboni Oliveira. Meta-mathematics of computational complexity theory. 2024.

[Pic15] Ján Pich. Logical strength of complexity theory and a formalization of the PCP theorem in bounded arithmetic. *Log. Methods Comput. Sci.*, 11(2), 2015.

[PS21] Ján Pich and Rahul Santhanam. Strong co-nondeterministic lower bounds for NP cannot be proved feasibly. In *Symposium on Theory of Computing* (STOC), pages 223–233, 2021.

[RR97] Alexander A. Razborov and Steven Rudich. Natural proofs. *J. Comput. Syst. Sci.*, 55(1):24–35, 1997.