A Formal Framework for Naturally Specifying and Verifying Sequential Algorithms

Chengxi Yang[1], Shushu Wu[2], and Qinxiang Cao[2(✉)]

[1] Zhiyuan College, Shanghai Jiao Tong University, Shanghai, China
arcadia-y@sjtu.edu.cn
[2] Shanghai Jiao Tong University, Shanghai, China
{Ciel77,caoqinxiang}@sjtu.edu.cn

Abstract. Current approaches for formal verification of algorithms face important limitations. For specification, they cannot express algorithms naturally and concisely, especially for algorithms with states and flexible control flow. For verification, formal proof based on Hoare logic cannot reflect the logical structure of natural proof. To address these challenges, we introduce a formal framework for naturally specifying and verifying sequential algorithms in Coq. We use the state relation monad to integrate Coq's expressive type system with the flexible control flow of imperative languages. It supports nondeterministic operations and customizable program states, enabling specifying algorithms at an appropriate level of abstraction. For verification, we build a Hoare logic for the monad and propose a novel two-stage proof approach that separates natural logical reasoning from mechanical composition. It reflects the logical structure of natural proof, enhancing modularity and readability. We evaluate the framework by formalizing the Depth-First Search (DFS) algorithm and verifying the Knuth-Morris-Pratt (KMP) algorithm.

Keywords: Formal Verification · Monad · Hoare Logic · Coq

1 Introduction

The formal verification of algorithms aims to formally specify algorithms and mathematically state and prove their functional correctness. It is canonical to formalize sequential algorithms in proof assistants like Coq[1] [17] and Isabelle/HOL [13], and there have been various approaches.

A common approach is to define algorithms as pure functions and prove their correctness using proof assistants' built-in logic, such as Verified Functional

C. Yang and S. Wu—These authors contributed equally to this work.

[1] Currently Coq was recently renamed to "Rocq", see https://rocq-prover.org/about#Name. However, we still use the name "Coq" in this paper since we adopt an older version (8.15.2) for formalization.

P. Rümmer and Z. Wu (Eds.): TASE 2025, LNCS 15841, pp. 49–66, 2026.
https://doi.org/10.1007/978-3-031-98208-8_4

Algorithms [2]. Still, many algorithms are naturally expressed in an imperative sequential form rather than pure functional form. Moreover, the restriction to structural recursion in Coq further complicates the task of specifying algorithms. For instance, even simple graph algorithms like Depth-First Search (DFS) become cumbersome to define under these constraints.

Another approach involves defining a simple imperative language, such as Imp [15], and formalizing algorithms within it. While this method allows for imperative constructs, it fails to leverage Coq's powerful type system, limiting its expressiveness for abstract operations like selecting an arbitrary element from a set.

In addition, the Isabelle Refinement Framework [8] enables users to formulate nondeterministic algorithms in a monadic style. Nevertheless, since it only supports functional and stateless programs, it is not convenient when formulating stateful algorithms (e.g. algorithms with a working set or state machine).

Beyond formalization, current verification approaches face additional limitations. Most frameworks for imperative algorithm verification rely on Hoare logic [5], which structures proofs according to the program's syntactic structure. This often requires grouping all propositions about the current program state into a conjunction and applying Hoare rules corresponding to the current statement. In contrast, natural proofs tend to organize propositions based on their logical relationships, following the proof's natural structure rather than the program's. This discrepancy becomes particularly evident when propositions span different program segments or loops, which Hoare logic struggles to handle elegantly.

Consider, for example, the *match* procedure in the Knuth–Morris–Pratt (KMP) algorithm [7], which finds the first occurrence of a pattern string in a text string. The procedure is shown in Algorithm 1.

Algorithm 1. Match procedure in the KMP algorithm

```
1:  procedure MATCH(patn, text, next)
2:      j = 0
3:      for i from 0 to text.len do
4:          ch = text[i]
5:          loop
6:              if patn[j] = ch then
7:                  j ← j + 1
8:                  break
9:              if j = 0 then
10:                 break
11:             j ← next[j − 1]
12:         if j = patn.len then
13:             return i − patn.len + 1
14:     return −1
```

In Algorithm 1, *patn* represents the pattern string to be located within the text string *text*, while *next* is an array containing shift information critical to

the algorithm's efficiency. It is precomputed by the table-building procedure in the KMP algorithm, and it represents the prefix function[2] of *patn*, which stores for each position j in *patn* the length of the longest proper prefix of $patn[0..j]$ that is also a suffix. *a.len* denotes the length of an array a.

To verify the procedure using the Hoare logic, one usually first provides a loop invariant I for the for-loop such as $jrange \wedge partial\_match \wedge partial\_bound \wedge no\_occur$, where

- *jrange* means that $0 \leq j < next.len$ so j is a valid index for *next* and *patn*.
- *partial_match* means that j is a partial match result for $text[0..i]$[3], i.e. $patn[0..j] = text[i-j..i]$.
- *partial_bound* asserts that j is an upperbound for the partial match result for $text[0..i]$. This with *partial_match* ensures j is the best result.
- *no_occur* states that there's no occurrence of *patn* in $text[0..i]$.

Then she aims to prove the for-loop body preserves I if the loop continues. To do so, she may assert another invariant I' for the inner loop such as $jrange \wedge partial\_match \wedge presuffix\_inv$, where *presuffix_inv* states some proposition that any partial match result k for $text[0..i+1]$ must obey. Next she proves that the inner loop body preserves I', I' can derive some propositions P when the inner loop breaks, and with P the for-loop body preserves I. Finally she can prove some postconditions with I when the procedure returns.

In contrast, a natural proof might proceed as follows: *jrange* trivially holds throughout the for-loop due to preconditions. Based on *jrange*, the inner loop preserves *partial_match* and hence outer loop preserves *partial_match*. Furthermore, we can prove the inner loop also preserves *presuffix_inv*. As a consequence, *partial_bound* and *no_occur* are invariants of the for-loop. These invariants collectively lead to the desired postconditions.

As shown in the example, a natural proof tends to incrementally assert and prove properties of the program according to their logical relevance and relations. In the process, the logical dependency between propositions (e.g. *no_occur* depends on *jrange*) is also naturally presented. This reveals the gap between the natural proof and the Hoare logic-based formal proof.

To address these challenges, we present a formal framework for naturally specifying and verifying sequential algorithms in Coq. Our approach introduces a state relation monad for algorithm specification based on the denotational semantics. The monad, defined over the ternary relation of initial state, return value, and resulting state, supports imperative constructs such as general recursion and loops break. It also integrates Coq's powerful type system with nondeterministic operations, enabling users to specify algorithms at an appropriate level of abstraction and customize program states as needed. We also provide stateless and errorful variants of the monad for different needs.

[2] More details of the prefix function can be found in classic algorithm textbooks [4].
[3] $array[s..t]$ refers to the 0-indexed segment of *array* ranging from s (included) to t (excluded).

For verification, we develop a proof framework for partial correctness tailored to our monad. It includes Hoare rules for various statements and introduces a novel approach to organizing proofs. This approach divides proofs into two parts: *essential proof* that captures the key logical implications in each basic block, and *mechanized proof* that combines propositions to establish end-to-end correctness. The latter can be highly automated. This results in proof that is more natural, modular, and readable.

We evaluate our framework by formalizing the DFS algorithm, and specifying and verifying the KMP algorithm. In another work [20], we further prove the correctness of a C program by proving it refines the KMP algorithm we specify in our framework. This highlights the real-world applicability and versatility of our framework.

The source code of our framework is available at https://github.com/Arcadia-Y/TASE25-Artifact.

Outline. The rest of the paper is organized as follows: Sect. 2 introduces the state relation monad used for defining algorithms and uses it to formalize the DFS algorithm. Section 3 presents the Hoare logic and our proof approach, with the KMP algorithm as a case study. In Sect. 4, we discuss related work, and in Sect. 5, we conclude with a summary and directions for future work.

2 State Relation Monad

2.1 Monad Design

Our framework is based on the monad, a well-established abstraction in functional programming for structuring computations with effects [10]. Its definition contains a type constructor `M: Type → Type` and two operators:

- `return: A → M A` (abbreviated as `ret`), which takes a value of type `A` and wraps it as a value of type `M A`.
- `bind: (M A) → (A → M B) → (M B)`, which takes a monadic value `m` of type `M A` and a function `f` of type `A → M B`. It tries to somehow unwrap `m`, applies `f` to it and returns the result as another monadic value.

In our framework, we model a program as a *state relation monad*, defined as a ternary relation over $\Sigma \times A \times \Sigma$, where Σ is the type of the program state and A is the type of the return value. This relation encodes the denotational semantics of a nondeterministic program c : $(s_1, r, s_2) \in c$ means that, starting from the state s_1, program c may terminate at s_2 and return r. We use the `unit` type with only one value `tt` to represent cases of no return value.

Our framework is based on a Coq library of sets [3], where sets and relations are represented as curried functions, such as `A → Prop` and `A → B → Prop`. The definition of a monadic program is as follows[4]:

[4] For simplicity and readability, some tedious portions of Coq code (e.g., implicit type parameters) are omitted in this paper.

```
Definition program (Σ A: Type): Type := Σ → A → Σ → Prop.
```

For any type `Σ`, `program Σ` is the type constructor for the state relation monad. The definitions of the two basic monad operators follow directly from the denotational semantics:

- For any a of type `A`, `ret`$(a)$is value of type `program Σ A` defined as

$$(s_1, r, s_2) \in \texttt{ret}(a) \iff r = a \ \wedge \ s_1 = s_2$$

- For any c of type `program Σ A` and f of type `A` $\to$ `program Σ B`, `bind`(c, f) is a value of type `program Σ B` defined as

$$(s_1, b, s_3) \in \texttt{bind}(c, f) \iff \exists\ a\ s_2,\ (s_1, a, s_2) \in c \ \wedge \ (s_2, b, s_3) \in f(a)$$

Intuitively, `ret` returns a value without changing the state, and `bind` composes two programs by passing the return value and terminal state of the first program to the second program. They satisfy the standard monad laws[5]:

1. `ret` is the left identity for `bind`: `bind`(`ret`$(x), f) = f(x)$.
2. `ret` is the right identity for `bind`: `bind`$(c,$ `ret`$) = c$.
3. `bind` is associative: `bind`(`bind`$(c, f), g) =$ `bind`$(c, \lambda x.$`bind`$(f(x), g))$.

To achieve an imperative-style syntax, we adopt a notation similar to Haskell's do-notation:

```
Notation "x ← c1 ;; c2" := (bind c1 (fun x ⇒ c2)) ...
Notation "e1 ;; e2" := (bind e1 (fun _: unit ⇒ e2) ...
```

The expressiveness of the state relation monad stems from the fact that defining a program is equivalent to defining a ternary relation in Coq's logical system. Combined with customizable states, this allows users to define program statements at an appropriate level of abstraction, providing high extensibility and flexibility.

In addition to user-defined statements, we provide several operators for convenient program construction:

- `choice` stands for nondeterministic choice between two programs.
- `assume` adds a logical proposition regarding the program state as an assumption. `assume'` is the notation for assumptions independent of the state.
- `any` returns an arbitrary value of a given type without changing the state.
- `update` modifies the state according to a binary relation over program states.

$$\begin{aligned}
\texttt{choice}(f, g) &:= f \cup g \\
\forall P : \Sigma \to \texttt{Prop},\ (s_1, \texttt{tt}, s_2) \in \texttt{assume}(P) &\iff P(s_1) \wedge s_1 = s_2 \\
\forall A : \texttt{Type},\ (s_1, a, s_2) \in \texttt{any}(A) &\iff s_1 = s_2 \\
\forall R : \Sigma \to \Sigma \to \texttt{Prop},\ (s_1, \texttt{tt}, s_2) \in \texttt{update}(R) &\iff R(s_1, s_2)
\end{aligned}$$

[5] Here, equality (=) denotes program equivalence, which corresponds to the equality of the underlying ternary relations, i.e. double inclusion.

By combining `choice` and `assume`, we can easily express common branching statements in imperative programs. For example, the following program computes the absolute value of an integer:

```
Example compute_abs: program unit Z :=
  choice (assume' (z >= 0);; ret z)
         (assume' (z < 0);; ret (-z)).
```

We can also use `any` and `assume` to define nondeterministic abstract operations. For instance, the following program returns an arbitrary prime number:

```
Example any_prime: program unit nat :=
  x ← any nat;;
  assume' (¬ exists (m n: nat), m > 1 ∧ n > 1 ∧ x = m * n);;
  ret x.
```

To express recursions and loops, we follow the standard approach in the denotational semantics by using the least fixed point in the Kleene fixed-point theorem [19]. For any directed-complete partial order `A` with a least element $\bot$ and any function $f : \texttt{A} \to \texttt{A}$, we define `Lfix`[6] as the supremum of the set produced by iterating f on $\bot$.

$$\texttt{Lfix}(f) := \sup(\{f^n(\bot) \mid n \in \mathbf{N}\})$$

Recursion can then be expressed directly using `Lfix`. For example, the following program computes the Fibonacci number:

```
Example Fibonacci: nat → program unit nat :=
  Lfix
  (fun (W: nat → program unit nat) (n: nat) ⇒
    choice
      (assume' (n <= 1);; ret n)
      (assume' (n > 1);;
        x ← W (n - 1);;
        y ← W (n - 2);;
        ret (x + y))).
```

We also define various loops using the least fixed point. For instance, we define loops with break to express flexible control flows. We first define an inductive type `ContinueOrBreak` similar to a sum type to represent results with control flow annotation, and then formalize loops using `Lfix`.

```
Inductive ContinueOrBreak (A B: Type): Type :=
| by_continue (a: A)
| by_break (b: B).
```

[6] It is defined for any f, although to apply the Kleene fixed-point theorem, f should be monotone and continuous.

```
Definition repeat_break_f
  (body: A → program Σ (ContinueOrBreak A B)) :=
  fun (W: A → program Σ B) (a: A) ⇒
      x ← body a;;
      match x with
      | by_continue a' ⇒ W a'
      | by_break b ⇒ ret b
      end.
Definition repeat_break
  (body: A → program Σ (ContinueOrBreak A B)): A → program Σ B :=
  Lfix (repeat_break_f body).
Definition continue (a: A): program Σ (ContinueOrBreak A B) :=
  ret (by_continue a).
Definition break (b: B): program Σ (ContinueOrBreak A B) :=
  ret (by_break b).
```

Using `continue` and `break`, we can construct loops with break. Below is an example of a loop that computes hailstone numbers.

```
Example hailstone: Z → program unit Z :=
  repeat_break
  (fun (x: Z) ⇒
    choice
      (assume' (x <= 1);; break x)
      (assume' (x > 1);;
       choice
        (assume' (exists k, x = 2 * k);;
        continue (x / 2))
        (assume' (exists k, x = 2 * k + 1);;
        continue (3 * x + 1)))).
```

For some algorithms, it is unnecessary to involve states; for some other algorithms, we need to model errorful computations. To address these cases, we provide stateless and errorful variants of our state relation monad: the set monad and state relation monad with error. Their syntax is very similar to the original monad. More details can be found in appendix A of the extended version [21].

2.2 Case Study: Formulating the DFS Algorithm

We formalize the Depth-First Search (DFS) algorithm in its imperative form using the state relation monad. Our definition of directed graphs and the step relation is based on a library of formalized graph theory [18].

```
Record PreGraph (Vertex Edge: Type) := {
  vvalid : Vertex → Prop; (* vertex set *)
  evalid : Edge → Prop; (* edge set*)
  src : Edge → Vertex; (* source of an edge *)
  dst : Edge → Vertex  (* destination of an edge *)
}.
```

```
Record step_aux (pg: PreGraph V E) (e: E) (x y: V): Prop := {
  step_evalid: pg.(evalid) e;
  step_src_valid: pg.(vvalid) x;
  step_dst_valid: pg.(vvalid) y;
  step_src: pg.(src) e = x;
  step_dst: pg.(dst) e = y;
}.
Definition step (pg: PreGraph V E) (x y: V): Prop :=
  exists e, step_aux pg e x y.
```

The program state for the DFS algorithm consists of a `visited` set that stores visited vertices and a `stack` that maintains the search path.

```
Record state (V: Type): Type := {
  stack: list V;
  visited: V → Prop;
}.
```

We also define several basic operations required for DFS:

$$\texttt{visit}(v) :=$$
$$\quad \texttt{update}(\lambda s_1 s_2.\ s_2.\texttt{visited} = s_1.\texttt{visited} \cup \{v\} \ \wedge \ s_2.\texttt{stack} = s_1.\texttt{stack})$$
$$\texttt{push\_stack}(v) :=$$
$$\quad \texttt{update}(\lambda s_1 s_2.\ s_2.\texttt{stack} = v :: s_1.\texttt{stack} \ \wedge \ s_2.\texttt{visited} = s_1.\texttt{visited})$$
$$\texttt{pop\_stack} :=$$
$$\quad \lambda s_1 v s_2.\ s_1.\texttt{stack} = v :: s_2.\texttt{stack} \ \wedge \ s_2.\texttt{visited} = s_1.\texttt{visited}$$
$$\texttt{if\_all\_neighbor\_visited}(pg, u) :=$$
$$\quad \texttt{assume}(\lambda s.\ \forall v,\ \texttt{step}(u, v) \to v \in s.\texttt{visited})$$

- $\texttt{visit}(v)$ marks vertex v as visited and leaves the stack unchanged.
- $\texttt{push\_stack}(v)$ pushes vertex v onto the stack without modifying the visited set.
- `pop_stack` pops the top vertex from the stack and returns it. Note how relation enables us to define the action of modifying the stack and returning the popped value concisely.
- $\texttt{if\_all\_neighbor\_visited}(pg, u)$ assumes that all neighbors of vertex u have been visited.

The DFS algorithm is then defined as follows:

```
Definition DFS_body (pg: PreGraph V E): V → program (state V) unit :=
  fun u ⇒
    choice
     (if_all_neighbor_visited pg u;;
      choice
       (assume (fun s ⇒ s.(stack) = nil);; break tt)
       (v ← pop_stack;; continue v))
```

```
    (v ← any V;;
     assume (fun s ⇒ ¬ v ∈ s.(visited));;
     assume' (step pg u v);;
     push_stack u;;
     visit v;;
     continue v).
Definition DFS (pg: PreGraph V E): V → program (state V) unit :=
  fun u ⇒
    visit u;; repeat_break (DFS_body pg) u.
```

The algorithm works as follows:

1. It visits the first vertex and begins to search from it.
2. If all neighbors of the current vertex `u` have been visited, it either terminates (if the stack is empty) or backtracks by popping the stack.
3. Otherwise, it nondeterministically selects an unvisited neighbor `v`, pushes `u` onto the stack, marks `v` as visited, and continues the search from `v`.

This formulation is concise and appropriately abstract, as it specifies neither the order in which vertices are visited, nor the concrete data structures used, aligning with the nondeterministic nature of DFS. We proved that a vertex is visited after the DFS if and only if it is reachable from the starting vertex based on the formulation.

3 Proof Framework

3.1 Hoare Logic

We develop a Hoare logic for our state relation monad to prove the partial correctness of algorithms. Drawing inspiration from Hoare Type Theory (HTT) [11], which integrates dependent types with Hoare triples, our logic adapts HTT's principles to a relational semantics while addressing imperative and nondeterministic constructs.

In our framework, a Hoare triple `Hoare P c Q` asserts that, for any initial state satisfying `P`, if the program `c` terminates, then resulting state and return value satisfy `Q`.

```
Definition Hoare (P: Σ → Prop) (c: program Σ A) (Q: A → Σ → Prop) :=
  forall s1 a s2, P s1 → (s1, a, s2) ∈ c → Q a s2.
```

We prove the following core Hoare rules[7]. Basic rules including bind, return and consequence are adapted from HTT's typing judgements. Choice rule enables compositional proof for branching programs. Assume, any and update rules provide strongest postcondition for our program constructs. For user-defined statements, step rule offers a general strongest postcondition. Pre-exist rule is useful

[7] We use the canonical notation $\{P\}c\{Q\}$ to denote a Hoare triple. Besides, for simplicity, some propositions may require lifting (e.g. $P \wedge Q$ may mean $\lambda s.P(s) \wedge Q(s)$).

for extracting existential variables in the precondition, which are often introduced by previous rules. We also adapt the conjunction rule into our logic to modularize proof for complicated postconditions, which is fundamental to our two-stage proof approach.

$$\textsc{Bind}\ \frac{\{P\}f\{Q\} \qquad \forall a,\ \{Q(a)\}g(a)\{R\}}{\{P\}\texttt{bind}(f,g)\{R\}} \qquad \textsc{Ret}\ \frac{P : \texttt{A} \to \Sigma \to \texttt{Prop}}{\{P(a)\}\texttt{ret}(a)\{P\}}$$

$$\textsc{Choice}\ \frac{\{P\}f\{Q\} \qquad \{P\}g\{Q\}}{\{P\}\texttt{choice}(f,g)\{Q\}} \qquad \textsc{Assume}\ \frac{}{\{P\}\texttt{assume}(Q)\{P \wedge Q\}}$$

$$\textsc{Any}\ \frac{}{\{P\}\texttt{any}(A)\{P\}} \qquad \textsc{Update}\ \frac{f : \Sigma \to \Sigma \to \texttt{Prop}}{\{P\}\texttt{update}(f)\{\lambda a s_2.\ \exists s_1, f(s_1, s_2) \wedge P(s_1)\}}$$

$$\textsc{Step}\ \frac{f : \Sigma \to \texttt{A} \to \Sigma \to \texttt{Prop}}{\{P\}f\{\lambda a s_2.\ \exists s_1, f(s_1, a, s_2) \wedge P(s_1)\}}$$

$$\textsc{Conseq}\ \frac{P_1 \to P_2 \qquad \{P_2\}f\{Q_2\} \qquad Q_2 \to Q_1}{\{P_1\}f\{Q_1\}}$$

$$\textsc{PreEx}\ \frac{\forall x,\ \{P(x)\}f\{Q\}}{\{\lambda s.\ \exists x, P(x,s)\}f\{Q\}} \qquad \textsc{Conj}\ \frac{\{P\}f\{Q_1\} \qquad \{P\}f\{Q_2\}}{\{P\}f\{Q_1 \wedge Q_2\}}$$

We also prove rules for recursions and loops with break. The fixed-point rule for recursions formalizes the induction principle for the iterated function in their denotational semantics.

$$\textsc{Fix}\ \frac{\forall W,\ (\forall a,\ \{P(a)\}W(a)\{Q\}) \to (\forall a,\ \{P(a)\}F(W,a)\{Q\})}{\forall a,\ \{P(a)\}\texttt{Lfix}(F,a)\{Q\}}$$

For loops with `break`, we introduce two auxiliary operators to modularize proof. One operator `continue_case` assumes that `x` has the form `by_continue a` and unwraps it to return `a`. The other one `break_case` is analogous. These lead to a more modular repeat-break rule.

$$\textsc{RepeatBreak}\ \frac{\begin{array}{c}\forall a,\ \{P(a)\}x \leftarrow f(a);;\texttt{continue\_case}(x)\{P\} \\ \forall a,\ \{P(a)\}x \leftarrow f(a);;\texttt{break\_case}(x)\{Q\}\end{array}}{\forall a,\ \{P(a)\}\texttt{repeat\_break}(f,a)\{Q\}}$$

For set monad and state relation monad with errors, we build a similar Hoare logic as well. See appendix A of the extended version [21] for more details.

3.2 The Two-Stage Proof Approach

Our framework introduces a two-stage proof approach to bridge the gap between natural reasoning and formal verification. This approach separates natural logical reasoning from mechanical composition, enhancing modularity and readability.

Consider for example, the *match* procedure in the KMP algorithm shown in Algorithm 1. We formalize it using the set monad as follows. `A` is the character type, and `range_iter_break`(l, h, f, j_0) is a for-loop with loop body being f, i ranging from l (included) to h (excluded) and j initialized as j_0.

```
Context {A: Type} (default: A)
        (patn text: list A) (next: list nat).
Definition inner_body(ch: A): nat → program (ContinueOrBreak nat nat) :=
  fun j ⇒
    choice
     (assume(ch = nth j patn default);; break (j+1))
     (assume(ch <> nth j patn default);;
      choice
       (assume(j = 0);; break 0)
       (assume(j <> 0);; continue (nth (j-1) next 0))).
Definition inner_loop(ch: A): nat → program nat :=
  repeat_break (inner_body ch).
Definition match_body:
  nat → nat → program (ContinueOrBreak nat nat) :=
  fun i j ⇒
      let ch := nth i text default in
      j' ← inner_loop ch j;;
      choice
        (assume (j' = length patn);;
         break (i - length patn + 1))
        (assume (j' < length patn);;
         continue (j')).
Definition match_loop: program (ContinueOrBreak nat nat) :=
  range_iter_break 0 (length text) match_body 0.
```

The correctness of *match* can be stated as follows: if *patn* is nonempty, *patn* and *next* has the same length and *next* is a prefix function of *patn*, then the return value r is either $by\_break(i)$ representing the first occurrence of *patn* in *text*, or $by\_continue(i)$ meaning there's no occurrence of *patn* in *text*. We formalize it as the following Hoare triple:

$$\{patn \neq nil \wedge patn.len = next.len \wedge prefix\_func(next)\}$$
$$\texttt{match\_loop}$$
$$\left\{\lambda r.\ \begin{cases} first\_occur(i) & \text{if } r = by\_break(i) \\ no\_occur(text.len) & \text{if } r = by\_continue(i) \end{cases}\right\}$$

The formal definition of these logical predicates and those introduced in Sect. 1 can be found in appendix C of the extended version [21].

The first stage of our two-stage proof is the **essential proof** that captures the logical structure of the algorithm proof. In this stage, the proof proceeds by

logical groups, each of which focuses on a logical topic and contains propositions relevant to the topic. For each group, we propose and prove **basic block propositions**, Hoare triples of relevant basic blocks. Tactics like `hoare_auto` could facilitate the basic block verification. Between adjacent basic blocks, we prove **logical implications** connecting their preconditions and postconditions. These also include implications between pre/post-conditions of the procedure and pre/post-condition of some basic blocks.

We provide tactical support for decomposing programs into individual basic blocks and verifying them conveniently (see appendix B of the extended version [21] for more details). Luckily, for the *match* example, the program is already well-structured so we do not need to transform it. Then following the natural proof, our proof proceeds as follows:

Group 1: range. In this group we prove *jrange* holds throughout the for-loop as a foundation for other propositions. Since *patn* is non-empty and *next* has the same length as *patn*, *jrange* holds when entering the match loop:

$$patn \neq nil \land patn.len = next.len \rightarrow jrange(0)$$

next is a prefix function implies that its elements are within certain range.

$$prefix\_func(next) \rightarrow \forall k : jrange(k), next[k] \in [0, k]$$

By definition, basic block is a program segment without control flow transfer like loops and branches [1]. However, in our framework, it can contain `choice` or even loops that have been verified previously, depending on the user's needs. Since *jrange* is a simple proposition, we treat `inner_body` as a basic block in this group. When it continues, *jrange* is preserved.

$$\{jrange(j) \land (\forall k : jrange(k), next[k] \in [0, k])\}$$
$$x \leftarrow \texttt{inner\_body}(ch, j);;\texttt{continue\_case}(x)$$
$$\{\lambda j'.\ jrange(j')\}$$

When it breaks, *jrange* no longer holds and the range of j is as follows.

$$\{jrange(j)\}\ x \leftarrow \texttt{inner\_body}(ch, j);;\texttt{break\_case}(x)\ \{\lambda j'.\ j' \in [0, patn.len]\}$$

After exiting the inner loop, when outer loop continues, *jrange* is back again.

$$\{j' \in [0, patn.len]\}\ \texttt{assume}(j' < patn.len);;\texttt{ret}(j')\ \{\lambda j''.\ jrange(j'')\}$$

Some reader may point out that in the original program it ends with $\texttt{continue}(j')$ instead of $\texttt{ret}(j')$. This does not matter since $x \leftarrow \texttt{continue}(j');;\texttt{continue\_case}(x)$ is equivalent to $\texttt{ret}(j')$. When applying the repeat-break rule in the second stage, the original basic block will become just like this.

Group 2: partial match. In this group we prove that $partial\_match(i, j)$ is an invariant of the for-loop, but we do not care about whether j is the best result. Obviously, it holds when the outer loop begins: $partial\_match(0, 0)$. Based on *jrange* and preconditions, $partial\_match(i, j)$ is preserved by the continue branch of the inner body.

$$\{next.len \leq patn.len \wedge prefix\_func(next) \wedge jrange(j) \wedge partial\_match(i, j)\}$$
$$\texttt{assume}(j \neq 0);;\texttt{ret}(next[j-1])$$
$$\{\lambda j'.\ partial\_match(i, j')\}$$

When inner loop breaks, we aim to prove the original $partial\_match(i, j)$ is extended to next i and current j, i.e. $partial\_match(i+1, j')$. The first case where $text[i] = patn[j]$ needs preconditions and the range of i in the for-loop.

$$\{next.len \leq patn.len \wedge i \in [0, text.len) \wedge jrange(j) \wedge partial\_match(i, j)\}$$
$$\texttt{assume}(text[i] = patn[j]);;\texttt{ret}(j+1)$$
$$\{\lambda j'.\ partial\_match(i+1, j')\}$$

The second case where $j = 0$ is trivial.

$$\{\}\ \texttt{assume}(j = 0);;\texttt{ret}(0)\ \{\lambda j'.\ partial\_match(i+1, j')\}$$

$partial\_match(i+1, j')$ is preserved when the for-loop continues because j' is not changed.

Group 3: partial bound. In this group, to prepare for the two *no_ occur* in the postcondition, we prove that *partial_ bound* is an invariant of the for-loop. It holds when the for-loop starts: $partial\_bound(0, 0)$. For inner loop, we propose a new invariant $presuffix\_inv(i, j)$. The two invariants of the for-loop, $partial\_match(i, j)$ and $partial\_bound(i, j)$, ensure $presuffix\_inv(i, j)$ when entering the inner loop.

$$next.len \leq patn.len \wedge i \in [0, text.len) \wedge partial\_match(i, j) \wedge$$
$$partial\_bound(i, j) \rightarrow presuffix\_inv(i, j)$$

When the inner loop continues, $presuffix\_inv(i, j)$ is preserved.

$$\{next.len \leq patn.len \wedge prefix\_func(next) \wedge jrange(j) \wedge$$
$$presuffix\_inv(i, j) \wedge text[i] \neq patn[j]\}$$
$$\texttt{assume}(j \neq 0);;\texttt{ret}(next[j-1])$$
$$\{\lambda j'.\ presuffix\_inv(i, j')\}$$

When inner loop breaks, we leverage $presuffix\_inv(i, j)$ to prove $partial\_bound(i+1, j')$. The first case is $text[i] = patn[j]$:

$$\{next.len \leq patn.len \wedge jrange(j) \wedge presuffix\_inv(i, j)\}$$
$$\texttt{assume}(text[i] = patn[j]);;\texttt{ret}(j+1)$$
$$\{\lambda j'.\ partial\_bound(i+1, j')\}$$

The second case is $text[i] \neq patn[j]$:

$$\{text[i] \neq patn[j] \wedge presuffix\_inv(i,j)\}$$
$$\texttt{assume}(j = 0);;\texttt{ret}(0)$$
$$\{\lambda j'.\ partial\_bound(i+1, j')\}$$

Similar to group 2, it is preserved when for-loop continues.

Group 4: post loop. In this group we prove the postcondition using the previous results. Firstly, we prove that $no\_occur(i)$ is an invariant of for-loop. Obviously $no\_occur(0)$. Based on $partial\_bound(i+1, j')$ and $jrange(j')$, we can prove it is preserved.

$$i \in [0, text.len) \wedge jrange(j') \wedge partial\_bound(i+1, j') \wedge no\_occur(i) \rightarrow$$
$$no\_occur(i+1)$$

If the for-loop terminates because i reaches the upperbound, then we have $no\_occur(text.len)$, which is half of our postcondition. On the other hand, if the for-loop terminates because it breaks, we can prove the *no_ occur* part of *first_ occur*.

$$\{no\_occur(i)\}$$
$$\texttt{assume}(j' = patn.len);;\texttt{ret}(i - patn.len + 1)$$
$$\{\lambda r.\ no\_occur(r + patn.len - 1)\}$$

Using $partial\_match(i+1, j')$, we can prove the other part of *first_ occur*.

$$\{partial\_match(i+1, j')\}$$
$$\texttt{assume}(j' = patn.len);;\texttt{ret}(i - patn.len + 1)$$
$$\{\lambda r.\ text[r..r + patn.len] = patn\}$$

Although the proof is complete from natural understanding, the formal proof needs to establish an end-to-end correctness theorem. Therefore, the second stage is the **mechanized proof**, which composes the essential proof's results into a complete formal argument using Hoare rules. In this stage, by repeatedly using conjunction rule and consequence rule, we can merge grouped propositions of the same basic block into a single Hoare triple. Then we can use choice rules to combine two branches' Hoare triples, use loop rules to transform the loop body's Hoare triple into the loop's Hoare triple, and use bind rule to combine them altogether.

In our example, combining results from the essential proof, the inner body satisfy following Hoare triples:

$$\{next.len \leq patn.len \wedge prefix\_func(next) \wedge (\forall k : jrange(k), next[k] \in [0, k]) \wedge$$
$$jrange(j) \wedge partial\_match(i, j) \wedge presuffix\_inv(i, j)\}$$
$$x \leftarrow \texttt{inner\_body}(text[i], j);;\texttt{continue\_case}(x)$$
$$\{\lambda j'.\ jrange(j') \wedge partial\_match(i+1, j') \wedge presuffix\_inv(i+1, j')\}$$

$$\{next.len \leq patn.len \wedge prefix\_func(next) \wedge i \in [0, text.len) \wedge$$
$$jrange(j) \wedge partial\_match(i, j) \wedge presuffix\_inv(i, j)\}$$
$$x \leftarrow \texttt{inner\_body}(text[i], j); ; \texttt{break\_case}(x)$$
$$\{\lambda j'.\ j' \in [0, patn.len] \wedge partial\_match(i+1, j') \wedge partial\_bound(i+1, j')\}$$

Using the repeat-break rule and the consequence rule, we further obtain the Hoare triple of the inner loop:

$$\{next.len \leq patn.len \wedge prefix\_func(next) \wedge i \in [0, text.len) \wedge$$
$$jrange(j) \wedge partial\_match(i, j) \wedge partial\_bound(i, j) \wedge no\_occur(i)\}$$
$$\texttt{inner\_loop}(text[i], j)$$
$$\{\lambda j'.\ j' \in [0, patn.len] \wedge partial\_match(i+1, j') \wedge$$
$$partial\_bound(i+1, j') \wedge no\_occur(i+1)\}$$

Then similarly we can obtain Hoare triples for the match body and the match loop, finishing the proof.

This two-stage proof approach improves modularity by splitting complicated propositions of complex programs into simple propositions of basic blocks. Besides, it enhances readability by ensuring that proofs reflect the natural logical structure of the correctness argument rather than the program's syntactic structure. Moreover, the approach achieves generality, as it can be applied to almost all formal frameworks based on Hoare logic.

In addition, we also formalize and prove the table-building procedure in the KMP algorithm, further showcasing the usability of our framework.

4 Related Work

The Isabelle Refinement Framework [8] provides a monadic approach to program verification, enabling users to specify and refine nondeterministic algorithms in a functional style. While it supports abstract specifications and stepwise refinement, it is primarily designed for functional and stateless programs, making it less suitable for imperative algorithms with complex state transitions. In contrast, our framework introduces customizable states and more flexible control flow constructs such as loops with break, which are convenient for naturally specifying algorithms like DFS and KMP. Additionally, it focuses on the data refinement and program refinement, but our work focuses more on specifying and verifying abstract algorithms.

Separation Logic [16] provides a foundation for reasoning about programs with mutable state and pointers, focusing on local reasoning about memory. Frameworks like the Iris [6] build upon Separation Logic to verify concrete programs, often involving complex memory manipulations. While these frameworks excel at verifying low-level implementation details, our approach focuses on specifying and verifying algorithms at a higher level of abstraction using a state relation monad, aiming to separate the core algorithmic logic from more concrete implementation concerns like memory management.

Nigron et al. also developed a framework with a Hoare logic for monadic programs in Coq [12]. While their work, like ours, utilizes monads and builds a corresponding Hoare logic, the motivations and resulting logics differ significantly. They aimed to reason about identifier freshness generated by monadic programs. To achieve this, they employed separation logic principles, tailoring their Hoare logic to enable local reasoning about freshness, notably featuring the frame rule. In contrast, our framework uses the state relation monad primarily to specify algorithms naturally and concisely. Consequently, our Hoare logic is designed to mirror the structure of natural proofs, featuring rules for various monadic operators, loop constructs, and recursion, alongside structural rules like the conjunction rule to facilitate modular natural reasoning about algorithm correctness.

Lammich and Neumann's framework for verifying DFS algorithms [9] provides a structured approach to specifying and proving the correctness of DFS using a combination of refinement techniques and modular proof components. Since it is built on the Isabelle Refinement Framework, it represents program states as explicit arguments. In contrast, our framework abstracts away the state using a state relation monad, enabling more concise and elegant formulations of algorithms. Besides, while the DFS framework features the design principle and technique of incrementally establishing invariants, our two-stage proof approach can not only achieve the same effect, but also handle more complex structures, such as multiple layers of loops and intricate branches. This makes our framework suitable for a wider range of algorithms.

There has been formal proof of the KMP algorithm, such as Paulson's [14]. It describes the table-building and matching procedures as single-layer loops and uses traditional Hoare logic for verification. In contrast, we model both procedures as two-layer loops that share the same inner loop, aligning more closely with common practices. Our proof framework allows us to provide a modular and readable proof for such formulation.

5 Conclusion

In this paper, we introduced a formal framework for naturally specifying and verifying sequential algorithms in Coq. Our approach leverages a state relation monad to integrate Coq's expressive type system with the flexible control flow of imperative languages. This allows for the specification of algorithms at an appropriate level of abstraction, supporting nondeterministic operations and customizable program states. We provided stateless and errorful variants of the monad. We also developed a proof framework for partial correctness tailored to our monad, which includes Hoare rules for various statements and a novel two-stage proof approach. This approach separates natural logical reasoning from mechanical composition, enhancing modularity and readability. We demonstrated the versatility and applicability of our framework through practical evaluations, including the formalization of the DFS algorithm and the verification of the KMP algorithm.

Acknowledgments. This material is based upon work supported by NSF China 62472274 and 92370201.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison-Wesley (2007)
2. Appel, A.W.: Verified Functional Algorithms, Software Foundations, vol. 3. Electronic textbook (2024), version 1.5.5. http://softwarefoundations.cis.upenn.edu
3. Cao, Q., Wu, X., Liang, Y.: A Coq library of sets for teaching denotational semantics. Electron. Proc. Theor. Comput. Sci. **400**, 79–95 (2024). https://doi.org/10.4204/EPTCS.400.6
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press (2009)
5. Hoare, C.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
6. Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: a modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, e20 (2018). https://doi.org/10.1017/S095679681800014X
7. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. **6**, 323–350 (1977). https://api.semanticscholar.org/CorpusID:11697579
8. Lammich, P.: Refinement for monadic programs. Archive of Formal Proofs (2012). https://isa-afp.org/entries/Refine_Monadic.html, Formal proof development
9. Lammich, P., Neumann, R.: A framework for verifying depth-first search algorithms. In: CPP 2015, pp. 137–146. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2676724.2693165
10. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991). https://doi.org/10.1016/0890-5401(91)90052-4
11. Nanevski, A., Morrisett, J., Birkedal, L.: Hoare type theory, polymorphism and separation. J. Funct. Program. **18**, 865–911 (2008). https://doi.org/10.1017/S0956796808006953
12. Nigron, P., Dagand, P.E.: Reaching for the star: tale of a monad in Coq. In: Cohen, L., Kaliszyk, C. (eds.) 12th International Conference on Interactive Theorem Proving (ITP 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 193, pp. 29:1–29:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.ITP.2021.29
13. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer, Heidelberg (2002)
14. Paulson, L.C.: Knuth–Morris–Pratt string search. Archive of Formal Proofs (2023). https://isa-afp.org/entries/KnuthMorrisPratt.html, Formal proof development
15. Pierce, B.C., et al.: Programming Language Foundations, Software Foundations, vol. 2. Electronic Textbook (2024)
16. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002), pp. 59–78. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029817

17. The Coq Development Team: The Coq reference manual – release 8.19.0 (2024). https://coq.inria.fr/doc/V8.19.0/refman
18. Wang, S., Cao, Q., Mohan, A., Hobor, A.: Certifying graph-manipulating C programs via localizations within data structures. Proc. ACM Program. Lang. **3**(OOPSLA) (2019). https://doi.org/10.1145/3360597
19. Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. MIT Press, Cambridge (1993)
20. Wu, S., Wu, X., Cao, Q.: Encode the ∀∃ relational Hoare logic into standard Hoare logic (2025). https://arxiv.org/abs/2504.17444
21. Yang, C., Wu, S., Cao, Q.: A formal framework for naturally specifying and verifying sequential algorithms (2025). https://arxiv.org/abs/2504.19852