

QCP: A Practical Separation Logic-based C Program Verification Tool

Xiwei Wu¹, Yueyang Feng^{1,*}, Xiaoyang Lu^{1,*}, Tianchuan Lin¹, Kan Liu¹,
Zhiyi Wang², Shushu Wu¹, Lihan Xie¹, Chengxi Yang¹, Hongyi Zhong¹, Zihan
Zhang¹, Juanru Li¹, Naijun Zhan², Zhenjiang Hu², and Qinxiang Cao^{1,†}

¹ Shanghai Jiao Tong University

² Peking University

Abstract. As software systems increase in size and complexity dramatically, ensuring their correctness, security, and reliability becomes an increasingly formidable challenge. Despite significant advancements in verification techniques and tools, their practical application to complex, real-world systems is often hindered by critical gaps in both automation and expressiveness. To address these difficulties, this paper presents **Qualified C Programming Verifier (QCP)**, a novel verification tool that integrates annotation-based automatic verification with interactive proving using Rocq. QCP employs symbolic execution and a separation logic entailment solver to automatically discharge many verification obligations, while deferring more complex obligations to Rocq for manual proof. Furthermore, QCP includes a VS Code extension designed to enhance proof efficiency and support a deeper understanding of both the program behavior and verification outcomes.

Keywords: Program Verification, Programming Languages, Separation Logic

1 Introduction

Software verification tools have made significant advancements, providing robust frameworks to ensure program correctness. Existing verification tools can be primarily classified into three predominant categories: (1) fully automated systems (e.g. Infer [4]) that focus on shape properties, such as memory safety and the absence of specific faults like null pointer dereferences, using built-in heuristics and algorithms for predefined predicates; (2) annotation-based systems (e.g., VeriFast [11], Viper [16], Hip/Sleek [18] and Smallfoot [2]) which verify not only shape properties but also some functional correctness properties by leveraging built-in SMT solvers and user-provided annotations; (3) interactive systems (e.g., VST [5], Iris [12] using Rocq) capable of verifying complex functional correctness but requiring substantial manual effort to write proof code in proof assistants.

*These authors contributed equally to this work.

†Corresponding Author

Ideally, a verification tool should achieve the following:

- (A) Support the verification of complex functional correctness properties, comparable to interactive proof assistants.
- (B) Minimize human intervention to the greatest extent possible, similar to fully automated and annotation-based tools. For simple programs and safety properties, verification should be achievable solely by writing annotations, without requiring manual proof construction in a proof assistant.
- (C) Provide real-time feedback during development. For example, immediately following the input of a precondition and a line of code, the tool should run symbolic execution and show the results before further code is written. If the execution fails, the tool should highlight the problematic line and report the error.

Existing tools either only support (A) or (B); quite few of them support (A) and (B) simultaneously, let alone (C) (to the best of our knowledge, none of them support (C)). This paper introduces **Qualified C Programming (QCP)**, a C program verification tool that targets all three requirements above.

For (A) and (B), QCP adopts well-established design elements from existing tools. Specifically, (1) QCP handles the memory manipulation of the C language using separation logic, which introduces an additional connective logic separating conjunction ($*$). $P * Q$ asserts that P and Q hold for the disjoint heap regions. (2) It allows users to annotate programs with assertions outlining a proof skeleton, enabling separation-logic-based symbolic execution to generate verification conditions automatically. (3) For C function calls, QCP requires users to provide function specifications that describe the functions' behavior. During symbolic execution, QCP verifies that the preconditions of callee functions are satisfied and derives the postconditions for the calling context. (4) To automate the process of checking verification conditions, users can employ QCP's built-in SMT solver or add customized separation logic heuristics. (5) QCP allows users to manually write Rocq proof code to fill proof gaps, which are about user-defined predicates that cannot be verified automatically by an SMT solver. Therefore, QCP can combine the advantages of annotation-based and interactive-based verifiers.

However, regarding requirement (C), conventional verification tools typically require users to provide a fully annotated program before initiating the verification process. This approach presupposes that developers possess thorough familiarity with both the program code and the tool's internal verification mechanisms—specifically, the ability to anticipate whether a given program can be automatically verified under the current assertions. In contrast, QCP introduces a Visual Studio Code extension and a web interface that visualize intermediate verification states and partial results during the verification process. This functionality provides users with insight into program behavior and the underlying verification process, enabling them to incrementally develop and refine annotations, which is especially useful when working with complex or unfamiliar code.

Outline. This paper makes two primary contributions: (1) a novel integration of annotation-based and interactive verification methodologies (Section 2), (2) an incremental verification support via a Visual Studio Code (VS Code) extension or a web interface (Section 3). Additionally, Section 4 presents the implementation of QCP. Section 5 presents comprehensive evaluation results of QCP across diverse sample programs. Section 6 discusses related work, including comparative analysis with existing verification tools to demonstrate QCP’s efficacy and practical utility, while Section 7 concludes with findings and suggests directions for future research.

2 Overview of QCP

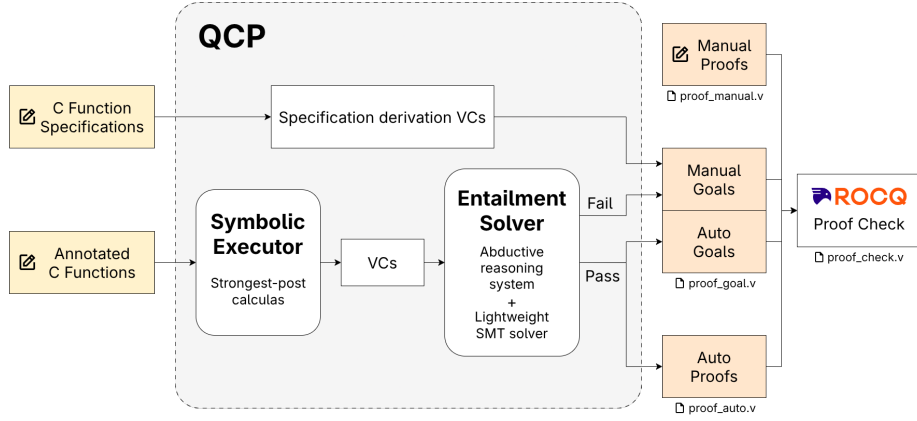


Figure 1: The components of QCP.

QCP takes annotated C code as input and generates verification conditions. The straightforward conditions are discharged automatically, as shown in Figure 1, while the complex ones are left for manual proof within the expressive logic of the Rocq proof assistant.

2.1 C module verification

QCP supports C module verification. In other words, we can verify all functions defined in a C file against their specifications, assuming that any function merely declared in that file — but implemented elsewhere — satisfies its own specification. In such cases, QCP requires that each function is accompanied by at least one function specification. Figure 2 is a minimum example of C module verification in QCP: `free_node_list` is only declared in this module and probably implemented elsewhere, and `sll_free` is implemented in this module. Both functions are attached with their specifications. The predicate `listrep`, conventionally used to describe the structural properties of singly-linked lists, is defined as follows:

```
listrep(x) = x == NULL && emp ||
  ∃ v y, x != NULL && x → next == y && x → data == v && listrep(y)
```

The `listrep` predicate specifically characterizes that pointer `x` references a singly-linked list, without concerning itself with the list's contents or other attributes. Here, the expression `x→next` indicates that we hold both the permission for `x` and the access permission to its `next` field. The syntax of QCP is designed to make sure that the `.c` file is still a legal C program that can pass typical compiler checks like `gcc`. Indeed, this file says, if `free_node_list(x)` correctly frees the memory permission of `x → data` and `x → next`, then the implementation of `sll_free(x)` is also correct w.r.t. its own specification, i.e. `sll_free(x)` frees all elements in the linked list with head point `x`.

```

1 // The function free_list_node and the definition of struct list
2 // are imported via the header file (.h).
3 struct list {
4     int data;
5     struct list *next;
6 };
7
8 void free_list_node(struct list * x)
9 /*@ With d n
10    Require x -> data == d && x -> next == n
11    Ensure emp */;
12
13 void sll_free(struct list * x)
14 /*@ Require listrep(x)
15    Ensure emp */
16 {
17     /*@ Inv listrep(x) */
18     while (x != NULL) {
19         /*@ exists v n, x -> data == v && x -> next == n && listrep(n) */
20         struct list *y = x -> next;
21         free_list_node(x);
22         x = y;
23     }
24 }

```

Figure 2: example of `sll_free`. The predicate `listrep` describes the structural properties of singly-linked lists.

In practical software verification, it is often necessary to reason about functions at different levels of abstraction. A function may be equipped with both a concrete, implementation-oriented specification and more abstract specifications that capture its high-level behavior. To handle this, QCP mandates that each function be assigned one primitive function specification, which is used directly in verifying the function itself. Any other function specifications associated

with it can be derived from this primitive specification and utilized in verifying functions that call it. QCP automatically generates verification conditions for function specification derivation, guaranteeing that all secondary specifications logically follow from the primary one, thereby maintaining consistency and completeness at the module level.

```

1  typedef struct LOS_DL_LIST {
2      int pstData;
3      struct LOS_DL_LIST * pstPrev, * pstNext;
4  } LOS_DL_LIST;
5
6  void LOS_ListAdd(LOS_DL_LIST *list, LOS_DL_LIST *node)
7  /*@ primitive_spec
8      With (x: Z) (a: Z)
9      Require list -> pstNext == x && x -> pstPrev == list &&
10         node -> pstData == a &&
11         has_permission(&(node -> pstPrev)) * has_permission(&(node -> pstNext))
12      Ensure  x -> pstPrev == node && node -> pstNext == x &&
13         list -> pstNext == node && node -> pstPrev == list &&
14         node -> pstData == a */;
15
16  void LOS_ListAdd(LOS_DL_LIST *list, LOS_DL_LIST *node)
17  /*@ head_insert_spec <= primitive_spec
18      With (a: Z)
19      Require node -> pstData == a && store_dll(list) *
20         has_permission(&(node -> pstPrev)) * has_permission(&(node -> pstNext))
21      Ensure store_dll(list) */
22  {
23      node->pstNext = list->pstNext;
24      node->pstPrev = list;
25      list->pstNext->pstPrev = node;
26      list->pstNext = node;
27  }
28
29  void LOS_ListTailInsert(LOS_DL_LIST *list, LOS_DL_LIST *node)
30  /*@ With (a: Z)
31      Require node -> pstData == a && store_dll(list) *
32         has_permission(&(node -> pstPrev)) * has_permission(&(node -> pstNext))
33      Ensure store_dll(list)
34  */
35  { LOS_ListAdd(list->pstPrev, node); }

```

Figure 3: Specification of `LOS_ListAdd` and `LOS_ListTailInsert` from LiteOS. The predicate `store_dll(x)` represents a circular doubly-linked list storage structure, where `x` denotes the sentinel node.

For example, Figure 3 demonstrates the multiple specification case of the `LOS_ListTailInsert` function from LiteOS [9], a lightweight operating system based on a real-time kernel. To simplify the description, we have reduced the

original polymorphic doubly-linked list to a circular doubly-linked list where each node stores an integer. Figure 4 illustrates the simplified definition of `LOS_DL_LIST`, which consists of only a previous pointer (`pstPrev`), a next pointer (`pstNext`), and a data pointer (`pstData`). The predicate `store_dll(x)` represents a circular doubly-linked list storage structure, where `x` denotes the sentinel node. In this context, our focus is solely on the overall shape property, while the actual data list is disregarded.

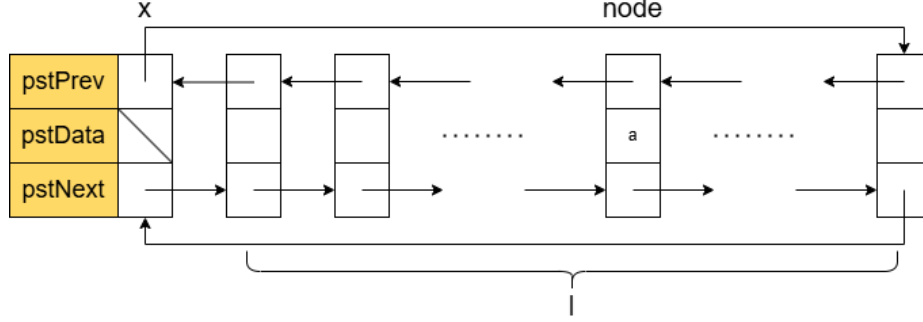


Figure 4: The definition of `LOS_DL_LIST` in LiteOS. The node `x` serves as a sentinel node, and its `pstData` field remains unused.

In this case, the `LOS_ListAdd` function is equipped with two specifications. The `primitive_spec` is used for the verification of `LOS_ListAdd` itself as well as the `LOS_ListTailInsert` function. Meanwhile, since `LOS_ListAdd` can also function as a routine for inserting a node at the head, an additional `head_insert_spec` is provided to describe this high-level behavior. QCP then requires the user to prove that `head_insert_spec` can be derived from `primitive_spec`, thereby eliminating the need for a separate, redundant verification of `LOS_ListAdd` against `head_insert_spec`.

2.2 C function verification

As shown in previous examples in Figure 2 and 3, users can use assertion annotations to verify a C function against its primitive specification. Given an input annotated C function, QCP performs symbolic execution to compute the strongest postcondition and generates corresponding verification conditions (VCs). These VCs are then processed by an entailment solver, which automatically discharges some verification conditions. The entailment solver integrates a rule-based abductive reasoning system (similar to the one used in VeriFast) and a lightweight SMT solver. Any remaining VCs are exported to Rocq for manual proof construction. This design combines the strengths of existing verification tools: it retains Rocq’s expressiveness and proving power through manual proof engineering, while enhancing automation and efficiency through the entailment solver.

We now proceed to demonstrate the QCP verification process using the function in Figure 2. Throughout the verification, it must be guaranteed that `x→next`

remains properly accessible for the store operation at line 14, which requires solving the following entailment:

$$x \neq \text{NULL} \ \&\& \ \text{listrep}(x) \vdash \exists v \ R, \ x \rightarrow \text{next} == v \ \&\& \ R$$

At line 19, the user introduces a new assertion annotation to unfold the `listrep` (x) predicate, ensuring that the symbolic execution of the read operation on $x \rightarrow \text{next}$ at line 19 can proceed correctly. For this user-introduced annotation, QCP generates the following verification condition:

$$\begin{aligned} & x \neq \text{NULL} \ \&\& \ \text{listrep}(x) \\ \vdash & \exists v \ n, \ x \rightarrow \text{next} == n \ \&\& \ x \rightarrow \text{data} == v \ \&\& \ \text{listrep}(n) \end{aligned}$$

This verification condition can be automatically discharged by our entailment solver. As symbolic execution progresses, the verification process generates multiple VCs, comprising both invariant validity checks and function call verifications. These VCs are systematically organized into four distinct output files:

- `proof_goal.v`: Contains statements of all generated VCs, including the derivation of function specifications and VCs generated during the symbolic execution of function bodies.
- `proof_auto.v`: Contains proof of all automatically verified VCs
- `proof_manual.v`: Contains proof goals of VCs requiring manual verification.
- `proof_check.v`: Ensuring all VCs present in `proof_goal.v` are properly accounted for in either `proof_auto.v` or `proof_manual.v`

The files `proof_goal.v`, `proof_auto.v`, and `proof_check.v` are automatically generated and require no user modification. The file `proof_manual.v` contains only the VCs to be proved, which require manual completion of the proofs. In practice, we achieve full automatic verification for this example in Figure 2, which can refer to Appendix B.

3 Integration of development and incremental verification

Notably, our tool supports incremental verification and simultaneous development of a program and its correctness proof—it operates not only on fully annotated code but also presents intermediate verification results and program states through its VS Code IDE integration and web interface QIDE. The interested readers could access <https://ide.qua.codes> to test it. Although the two interfaces differ slightly in layout, they present identical verification information. The following explanation will be based on the VS Code IDE version.

As shown in the screenshot of Figure 5, which corresponds to the `sll_free` function in Figure 2, the current symbolic state is displayed on the right-hand side. To enhance usability for developers with varying levels of expertise, we provide multiple assertion formats that allow users to customize how the current state is visualized and interpreted. Such detailed feedback helps users better understanding program behavior and the verification process, thereby assisting

them in completing annotations—or refining unfinished programs—more effectively. Green highlighting indicates code segments that have passed preliminary checks and are ready for symbolic execution to generate verification conditions. This demonstrates how our VS Code plugin allows users to seamlessly incorporate verification into their development workflow, offering real-time validation during code writing.

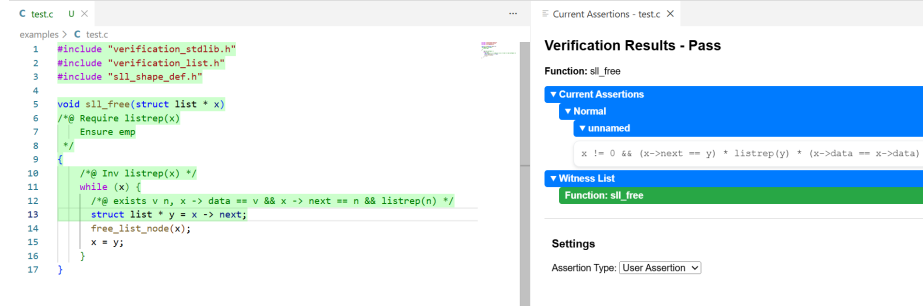


Figure 5: Live verification during code editing in VS Code.

As for failure scenarios, Figure 6 illustrates a case in which an incorrect implementation of the `sll_free` function leads to a use-after-free error. In this situation, symbolic execution fails due to the inability to retain permission for `x→next` after the memory has been freed. Within the VS Code interface, the affected code segment is highlighted in red on the left, while the right-hand panel displays the error message: “Cannot derive the precondition of Memory Read.”. It is worth noting that the code in Figure 6 is not yet complete. Even in such cases, QCP is able to provide intuitive and actionable error feedback, assisting users in diagnosing and correcting such faults.

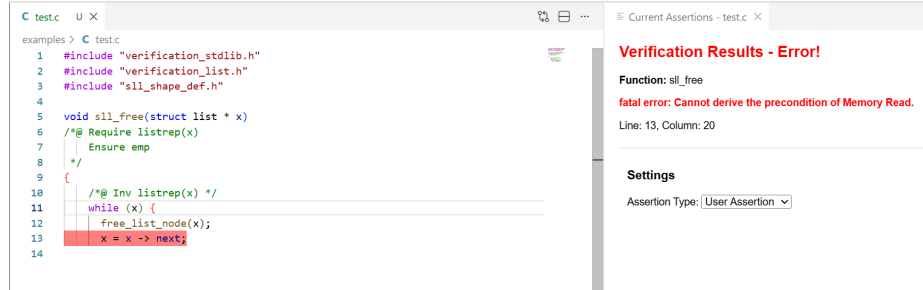


Figure 6: Failure case for live verification during code editing in VS Code.

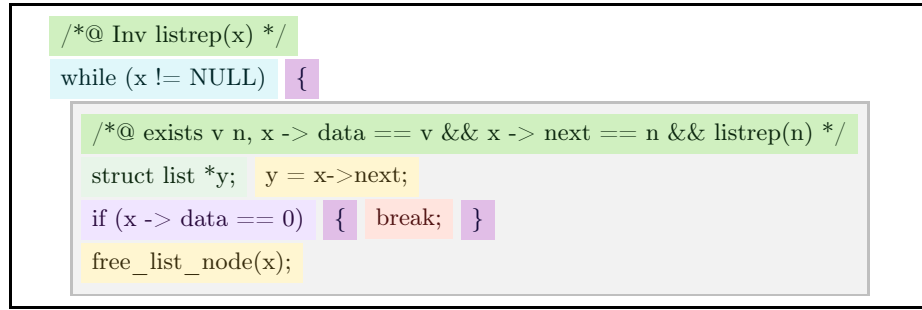
For those cases that pass symbolic execution but cannot be fully solved automatically, we use yellow highlighting to distinguish them from failure cases. The corresponding code lines will be marked in yellow.

In summary, QCP features an integrated development interface that provides immediate visual feedback during the verification process. This significantly enhances the user’s understanding of program behavior and proof status, thereby facilitating more efficient annotation writing and error debugging.

4 Implementation

4.1 Symbolic execution

QCP’s frontend decomposes complete or incomplete C programs into what we term partial statements, which subsequently serve as the input to QCP’s symbolic executor. Figure 7 illustrates the decomposition of an incomplete function into its constituent partial statements, with different types of partial statements distinguished by color. These include: one program variable declaration (`struct list *y`); two singleton statements (specifically, the assignments `y = x->next` and the function call `free_list_node(x)`); two annotation statements; one control-flow statement (`break`); one conditional statement (`if (x -> data == 0)`); one iteration statement (`while`); two block begin and a block end. The complete grammar for partial statements is provided in Figure 8.



```

/*@ Inv listrep(x) */
while (x != NULL) {
    /*@ exists v n, x -> data == v && x -> next == n && listrep(n) */
    struct list *y; y = x->next;
    if (x -> data == 0) { break; }
    free_list_node(x);
}

```

Figure 7: Example illustrating symbolic state during partial program execution, with different types of partial statements distinguished by color.

Partial Statement $ps ::= s$		singleton statement
<i>asrt</i>		annotation statement
<code>break</code> <code>continue</code>		control-flow statement
<code>return</code> <code>return e</code>		return statement
<code>if (e)</code> <code>else</code>		conditional-if/else
<code>switch(e)</code> <code>case e:</code> <code>default:</code>		switch-case-default
<code>while (e)</code> <code>do</code> <code>for(s;e;s)</code>		iteration
<code>{</code> <code>}</code>		block begin/end
<i>ctype</i> <code>var_name</code>		program variable declaration

Figure 8: Syntax of C partial statements. Singleton statements encompass assignments, increment/decrement operations, and function calls.

The symbolic execution over partial statements follows conventional methodologies. During execution, QCP maintains a symbolic state stack. Figure 7 shows

a partial program example: when execution reaches point `free_list_node(x);`, the stack contains two nodes as shown by two rectangular blocks. Within the inner block, QCP maintains both a **Normal** assertion (when $x \rightarrow \text{data}$ does not store 0) and a **Break** assertion (when $x \rightarrow \text{data}$ stores 0). Similarly, `continue` and `return` statements generate **Continue** and **Return** assertions, respectively. These states are visualized in real-time within the QIDE environment (see Figure 5). Upon loop exit, the four types of inner assertions—**Normal**, **Break**, **Continue**, and **Return**—are merged with the outer assertion before symbolic execution continues with subsequent instructions.

4.2 Entailment solver

To efficiently discharge verification conditions, QCP employs a custom entailment solver at its core. This solver is specifically designed to automate the reasoning process within our verification framework. As shown in Figure 1, the solver integrates an abductive reasoning system with a lightweight SMT solver.

The reasoning system is a rule-based separation logic engine responsible for eliminating spatial constraints in separation logic formulas. It supports user-defined predicates and corresponding elimination rules while simultaneously generating soundness proofs for each deduction step. QCP’s implementation builds upon the Stellis framework [1], and the details are provided in Appendix A.

Our SMT solver implements decision procedures for two fundamental theories: **Linear Integer Arithmetic (LIA)** [7] and **Uninterpreted Functions (UF)** [19]. The solver architecture follows the Nelson-Oppen framework [15] for theory combination, ensuring soundness when processing formulas containing mixed LIA and UF constraints. Currently, the solver does not support quantified formulas; such cases require manual user intervention.

4.3 Rocq library

QCP formalizes a C memory model and develops a separation logic library upon it. This library includes a C-specific assertion language, support for relational reasoning, and a suite of specialized proof tactics. The library employs C-style notations that closely resemble front-end C assertions, while being expressed in standard separation logic form. For example, the verification condition discussed in Section 2.2 is presented as follows:

```
forall (x: Z),
  [| (x <> 0) |] && (listrep x)
⊢ EX (v: Z) (n: Z),
  [| (x <> 0) |] && ((&((x) # "list" → "next")) # Ptr ↦ n)
  ** (listrep n) ** ((&((x) # "list" → "data")) # Int ↦ v)
```

To efficiently manage the proof burden associated with separation logic, QCP provides a suite of specialized proof tactics. These tactics handle the tedious aspects of reasoning about spatial assertions, such as:

- **Intros/Exists**: Introduces or instantiates existentially quantified variables, and improves the organization of canonical preconditions.
- **entailer!**: Automatically cancels common spatial predicates on both sides of the entailment by leveraging the associativity and commutativity of the separating conjunction.
- **sep_apply**: Performs separation logic transformations by applying lemmas; it automatically collects the left spatial predicates of a lemma against those in the entailment’s left side, replacing them with the lemma’s right side.

Furthermore, QCP incorporates relational verification capabilities through Shushu Wu’s new discovery that can encode forall-exists relational Hoare logic into standard Hoare logic [22]. This design allows users to write relational specifications for conducting refinement proofs and algorithm consistency verification within the same framework. The system smoothly supports standard Hoare logic, relational Hoare logic, and even hybrid reasoning styles, providing flexibility for diverse verification requirements. Several verification examples demonstrating these capabilities will be presented in Appendix D.

5 Evaluation

5.1 Performance of QCP

We evaluated QCP across 155 functions spanning five domains: arithmetic (Arith), typical data structures (Typical DS, covering sll/dll/trees/arrays), typical algorithm implementations (Typical alg, covering merge sort, Knuth-Morris-Pratt algorithm, etc.), LiteOS microkernel components (mainly for doubly-linked lists), QCP implementation components (QCP Impl, covering Fourier–Motzkin elimination algorithm [7], Typeinfer algorithm, etc.).

Our evaluation focuses on QCP’s ability to verify imperative implementations that exhibit typical programming patterns rather than verification-oriented implementations. The supported C features (see Section 5.4) are representative of these patterns and are exercised across all benchmark examples. The functions in QCP implementation components represent a codebase that evolved organically over a year of QCP’s development, providing a case study of verifying non-verification-oriented software; details are provided in Appendix E.

Table 1 reveals two key trends: (1) The code-to-annotation ratio ranges from 1:0.6 for arithmetic programs (159 code vs. 98 annotations) to 1:2.3 for QCP implementation components (QCP Impl), reflecting the intrinsic specification needs of imperative programming; (2) QCP automates 74.3% of verification conditions (1964 auto vs. 679 manual VCs), achieving high automation in arithmetic checks while reserving expert effort in LiteOS (60.3% automation).

Several examples in our evaluation are verified fully automatically, and the results for these cases are presented in Table 2. On these cases, QCP achieves automated verification with an average code-to-annotation ratio of 1:0.5.

Table 1: Evaluation results for QCP, including annotation lines, the number of verification conditions (Auto VCs and Manual VCs), symbolic-execution time per function (SE Time), and Rocq checking time per function (Rocq Time).

Type	Functions	Annotations (Lines)	Total Code (Lines)	Auto VCs	Manual VCs	SE Time (s)	Rocq Time (s)
Arithm	12	98	159	84	39	0.07	1.43
Typical DS	67	954	1135	716	192	0.40	2.38
Typical alg	19	574	339	214	101	0.06	3.23
QCP Impl	36	1757	755	859	287	2.01	16.48
LiteOS DLL	21	493	584	91	60	0.02	11.00
Total	155	3876	2972	1964	679	0.65	6.85

Table 2: Automatically verified results for QCP

File	Type	Functions	Annotations (Lines)	Total Code (Lines)	Auto VCs	SE Time (s)
add	Arith	5	27	27	18	< 0.01
max3	Arith	1	9	14	4	0.01
sll_auto	Typical DS	6	45	110	48	0.02
dll_auto	Typical DS	9	66	187	81	0.16
avl_insert	Typical DS	8	99	116	250	2.84
array_auto	Typical DS	9	68	85	81	0.03

5.2 Comparison with separation-logic-based annotation verifiers

Real-world programs often involve complex memory structures that necessitate the use of separation logic. However, some annotation-based verification tools, such as Frama-C [13], do not support separation logic. For this reason, our evaluation is limited to a comparison with other separation logic-based verifiers. Table 3 and Table 4 present a comparative evaluation of QCP against VeriFast and Hip/Sleek on a suite of fully automatic verification cases, measuring the number of required annotations and verification time. The data, drawn from their respective benchmarks, show that QCP achieves verification times comparable to VeriFast and Hip/Sleek in the fully automatic mode. Our evaluation can be found at <https://anonymous.4open.science/r/Qualified-C-Programming-B44E/>.

Table 3: Comparison of verification effort and verification checking time (Annotation Lines, Proof Lines, and Time in s) between QCP and VeriFast.

			QCP		VeriFast	
Program	Functions	LoC	Anno.	Time(s)	Anno.	Time(s)
arraylist	7	74	33	0.37	31	0.10
strlib	4	38	31	0.14	17	0.10
sll_stack	9	77	40	0.09	46	0.17

5.3 Comparison with interactive verifiers

To the best of our knowledge, limited prior research has integrated annotation-based and interactive verification methodologies such as QCP. Among exist-

Table 4: Comparison of verification checking time between QCP and Hip/Sleek.

Program	Functions	LoC	Anno.	Hip/Sleek Time(s)	QCP Time(s)
Singly linked list	16	139	32	0.93	0.05
Doubly linked list	15	181	34	0.90	0.14
Sort algorithms	10	110	20	1.52	0.05

ing approaches, VST-A is the most relevant. We evaluate the performance of QCP against both VST-A and VST using the VST-A benchmark (see Table 5). The benchmark data are sourced directly from the original publication [23]. We recorded the number of annotations, proof codes, and verification checking time for each tool, including both the time required to generate Rocq code and the time to execute the Rocq proof.

In terms of proof effort, QCP also significantly reduces the required proof lines by 52.8% compared to VST-A (457 vs. 969) and 62.3% compared to VST (457 vs. 1212), which is attributed to its use of an entailment solver that automates significant portions of the proof process. As observed in Table 1, the symbolic execution time in QCP is notably lower than the corresponding Rocq verification time. Therefore, reducing proof length directly contributes to shorter Rocq execution time. For example, in the SLL category, QCP achieves a 95.4% reduction in execution time compared to VST-A (16.11s vs. 349.75s) and an 83.3% reduction compared to VST (16.11s vs. 96.39s).

It should be noted that QCP requires a larger number of annotations than VST-A. This is because QCP utilizes annotations to drive assertion transformations—effectively shifting the separation logic transformations that would traditionally be done within proofs into the annotation phase. Despite this increase in annotations, the combined total of annotation lines and proof lines in QCP remains lower than that of VST-A. This result demonstrates that QCP not only provides a more intuitive proving experience by allowing users to perform assertion transformations through manual annotations, but also achieves an overall reduction in the total verification effort.

Table 5: Comparison of verification effort and verification checking time (Annotation Lines, Proof Lines, and Time in s) among QCP, VST-A, and VST.

			QCP			VST-A			VST	
Program	Function	Codes	Anno.	Proofs	Time(s)	Anno.	Proofs	Time(s)	Proofs	Time(s)
Basics	8	78	59	8	0.65	61	84	84.36	156	15.90
SLL	18	350	209	457	16.11	140	969	349.75	1212	96.39
DLL	4	95	36	70	4.33	32	171	155.25	213	29.91
BST	4	115	93	169	13.62	67	202	190.35	364	33.83

5.4 Supported C features

QCP provides comprehensive support for standard C types, encompassing integers, pointers, arrays, enumerated types, `struct`, `union`, and `typedef`. The unsupported features consist of floating-point types, function pointers, and bit-fields.

Regarding expression handling, QCP accommodates most C expressions, including: postfix operators (like array subscripting and structure member selection), arithmetic operators (both value and pointer arithmetic), bitwise and logical operators, conditional operator, and assignment operators. The system strictly enforces C-standard implicit type conversions and implements short-circuit evaluation for all expressions. Notable unsupported features include string literals, the comma operator, and compound operators (e.g., `(int []){2, 4}`).

For control flow, QCP supports all standard C control flow statements including conditional branches (`if-else`), iteration constructs (`while`, `for`, and `do-while`), and selection statements (`switch-case`). The only non-supported control flow feature is the `goto` statement.

6 Related work

C module verification: multiple specifications in other tools Multiple specification verification is valuable in real-world program verification, and many verifiers have recognized this need and introduced relevant support. For instance, Frama-C [13] and VeriFast [11] allow multiple specifications to be verified simultaneously within a single function, without requiring these specifications to be derivationally related—an approach that is particularly suitable for independent multiple specifications. Although similar effects can be encoded through specification derivation (Appendix F will show an example), QCP will also incorporate such functionality in future development. Similar to specification derivation, VSU [3] enables specification multiplicity between modules via subsumption, yet each module internally maintains only one specification. In this regard, Hip/Sleek [17] goes further: it supports simultaneous symbolic execution for multiple specifications and specification derivation [6]. But it requires all derivations to be automatically verified—a constraint that can become a bottleneck in complex correctness proofs. In contrast, QCP offers more flexible support for specification derivation by allowing manual proof of the derivation verification conditions.

C function verification: fully automated systems Tools like CBMC [14] employ bounded model checking to verify memory safety and undefined behavior, yet their reliance on solvers like Z3 leads to non-negligible false positives. Similarly, abstract interpretation tools such as SPARTA [8] and MemCAD [10] utilize various abstract domains to verify numerical and basic memory properties efficiently. While these fully automated tools excel at proving a wide range of simple properties, they generally fall short in establishing the full functional correctness of real-world programs.

C function verification: annotation-based system VeriFast [11] and Hip/Sleek [17] are separation logic-based annotation verifiers, and CN [20] is a separation logic-based refinement type annotation verifier. All three tools incorporate specialized annotations designed to assist in the symbolic execution process. They rely on SMT solvers or type checkers to discharge verification conditions, and for moderately complex cases, VeriFast and Hip/Sleek further allow users to supply lemma

functions to aid the SMT solver. While this approach can support automated proofs of functional correctness to a certain extent, it remains insufficient for more complex verification scenarios. As demonstrated in Tables 3 and 4, QCP achieves verification automation comparable to that of VeriFast and Hip/Sleek. Moreover, by enabling manual proofs for verification conditions that cannot be resolved automatically, QCP can address more complex verification tasks.

As for Frama-C [13], it is built upon first-order logic and does not support separation logic. While Frama-C excels at scalable, automated analysis of runtime errors and functional properties within its supported logic fragment, it is not convenient for verifying complex data structures and algorithms.

C function verification: interactive systems VST-A [23] decomposes the entire annotated program into multiple straight-line programs in Rocq, requiring users to perform manual symbolic execution proofs using tactics. The entire system is built on CompCert, with corresponding tactics and decomposition operations formally verified for soundness in Rocq. In contrast, QCP only requires users to prove specific VCs while leveraging entailment solvers to automatically resolve others. This approach significantly reduces the user’s workload and enhances verification efficiency.

RefinedC [21] translates annotated C programs directly into Rocq. It employs typing rules to drive execution and generate typing derivation VCs, which are mostly discharged through a library of Rocq tactics—though unresolved cases still require manual intervention. However, its lack of SMT-based automation limits proof power, while its full implementation within Rocq results in lower verification efficiency and higher usability overhead compared to QCP.

7 Conclusion

In this work, we presented QCP, a verification tool designed to efficiently verify complex program properties. By leveraging separation logic, QCP provides a flexible and intuitive framework for symbolic execution and verification condition generation. QCP integrates both entailment solvers and Rocq, balancing automation with the ability to handle intricate proofs manually when necessary. Our evaluation demonstrates QCP’s effectiveness in verifying programs with common data structures and highlights its ability to reduce manual effort.

Looking ahead, we aim to extend QCP to support advanced language features such as function pointers and goto statements. We also plan to enhance its usability for developers—for instance, by enabling integrated verification of multiple specifications. QCP represents a significant step forward in making program verification more accessible and practical for real-world software development.

References

1. This paper is now under review

2. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: modular automatic assertion checking with separation logic. In: Proceedings of the 4th International Conference on Formal Methods for Components and Objects. p. 115–137. FMCO'05, Springer-Verlag, Berlin, Heidelberg (2005). https://doi.org/10.1007/11804192_6, https://doi.org/10.1007/11804192_6
3. Beringer, L.: Verified software units. In: Programming Languages and Systems: 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021, Proceedings. p. 118–147. Springer-Verlag, Berlin, Heidelberg (2021). https://doi.org/10.1007/978-3-030-72019-3_5, https://doi.org/10.1007/978-3-030-72019-3_5
4. Calcagno, C., Distefano, D.: Infer: an automatic program verifier for memory safety of c programs. In: Proceedings of the Third International Conference on NASA Formal Methods. p. 459–465. NFM'11, Springer-Verlag, Berlin, Heidelberg (2011)
5. Cao, Q., Beringer, L., Gruetter, S., Dodds, J., Appel, A.W.: Vst-floyd: A separation logic tool to verify correctness of c programs. *Journal of Automated Reasoning* **61**(1), 367–422 (Jun 2018). <https://doi.org/10.1007/s10817-018-9457-5>, <https://doi.org/10.1007/s10817-018-9457-5>
6. Chin, W.N., David, C., Nguyen, H.H., Qin, S.: Multiple pre/post specifications for heap-manipulating methods. In: 10th IEEE High Assurance Systems Engineering Symposium (HASE'07). pp. 357–364 (2007). <https://doi.org/10.1109/HASE.2007.19>
7. Dantzig, G.B., Curtis Eaves, B.: Fourier-motzkin elimination and its dual. *Journal of Combinatorial Theory, Series A* **14**(3), 288–297 (1973). [https://doi.org/https://doi.org/10.1016/0097-3165\(73\)90004-6](https://doi.org/https://doi.org/10.1016/0097-3165(73)90004-6), <https://www.sciencedirect.com/science/article/pii/0097316573900046>
8. facebook: <https://github.com/facebook/SPARTA>, gitHub repository
9. Huawei LiteOS: Liteos kernel. <https://github.com/LiteOS/LiteOS>
10. Illous, H., Lemerre, M., Rival, X.: A relational shape abstract domain. In: Barrett, C., Davies, M., Kahsai, T. (eds.) *NASA Formal Methods*. pp. 212–229. Springer International Publishing, Cham (2017)
11. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: Verifast: A powerful, sound, predictable, fast verifier for c and java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) *NASA Formal Methods*. pp. 41–55. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
12. Jung, R., Krebbers, R., Birkedal, L., Dreyer, D.: Higher-order ghost state. *SIGPLAN Not.* **51**(9), 256–269 (Sep 2016). <https://doi.org/10.1145/3022670.2951943>, <https://doi.org/10.1145/3022670.2951943>
13. Kirchner, F., Cuoq, P., Correnson, L., Prevosto, V., Signoles, J.: Framac: A software analysis perspective. *Formal Aspects of Computing* **27**(3), 573–609 (2015). <https://doi.org/10.1007/s00165-014-0326-7>, <https://doi.org/10.1007/s00165-014-0326-7>
14. Kroening, D., Schrammel, P., Tautschnig, M.: Cbmc: The c bounded model checker (2023), <https://arxiv.org/abs/2302.02384>
15. Kroening, D., Strichman, O.: *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edn. (2008)
16. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. LNCS, vol. 9583, pp. 41–62. Springer-Verlag (2016), https://doi.org/10.1007/978-3-662-49122-5_2

17. Nguyen, H.H., David, C., Qin, S., Chin, W.N.: Automated verification of shape and size properties via separation logic. In: Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation. p. 251–266. VMCAI’07, Springer-Verlag, Berlin, Heidelberg (2007)
18. Nguyen, Q.L., David, C., Chin, W.N.: Hip/sleek: Verification system for heap-manipulating programs (splice example). <https://github.com/hipsleek/hipsleek> (2023), https://github.com/hipsleek/hipsleek/blob/master/benchmark/SV-COMP/list_properties/splice.ss, gitHub repository, benchmark example: `splice.ss`
19. Nieuwenhuis, R., Oliveras, A.: Proof-producing congruence closure. In: Proceedings of the 16th International Conference on Term Rewriting and Applications. p. 453–468. RTA’05, Springer-Verlag, Berlin, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32033-3_33, https://doi.org/10.1007/978-3-540-32033-3_33
20. Pulte, C., Makwana, D.C., Sewell, T., Memarian, K., Sewell, P., Krishnaswami, N.: Cn: Verifying systems c code with separation-logic refinement types. Proc. ACM Program. Lang. **7**(POPL) (Jan 2023). <https://doi.org/10.1145/3571194>, <https://doi.org/10.1145/3571194>
21. Sammler, M., Lepigre, R., Krebbers, R., Memarian, K., Dreyer, D., Garg, D.: Refinedc: Automating the foundational verification of c code with refined ownership types. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021). pp. 158–174. ACM, New York, NY, USA (2021). <https://doi.org/10.1145/3453483.3454060>, <https://dl.acm.org/doi/10.1145/3453483.3454060>
22. Wu, S., Wu, X., Cao, Q.: Encode the $\forall\exists$ relational hoare logic into standard hoare logic. Proc. ACM Program. Lang. **9**(OOPSLA2) (Oct 2025). <https://doi.org/10.1145/3763138>, <https://doi.org/10.1145/3763138>
23. Zhou, L., Qin, J., Wang, Q., Appel, A.W., Cao, Q.: Vst-a: A foundationally sound annotation verifier. Proc. ACM Program. Lang. **8**(POPL) (Jan 2024). <https://doi.org/10.1145/3632911>, <https://doi.org/10.1145/3632911>

A Overview of Stellis

Stellis is a domain-specific language (DSL) designed to streamline the automation of separation logic entailment proofs. By enabling users to specify verification strategies, Stellis systematically reduces complex separation logic formulas (combining spatial and pure constraints) into pure logical forms that can be directly processed by constraint solvers. This transformation eliminates manual reasoning about heap-allocated structures while preserving soundness.

To ensure correctness, Stellis integrates a novel algorithm based on the Ramify Rule. For each user-defined strategy, the framework automatically generates a corresponding soundness lemma. The validity of the strategy is thereby reduced to proving this lemma, decoupling high-level strategy design from low-level proof obligations. This approach not only guarantees formal soundness but also empowers users to extend verification capabilities without requiring deep expertise in separation logic metatheory.

Figure 9 presents the formal syntax of Stellis. The figure and its accompanying description are from the original Stellis paper [1]. An Stellis program

Priority	r	$::= n$
Pattern term	\hat{t}	$::= n \mid ?x \mid x \mid \mathbf{field_addr}(\hat{t}, field) \mid f(\hat{t}_1, \hat{t}_2, \dots)$
Pattern pure formula	\hat{p}	$::= \hat{t}_1 == \hat{t}_2 \mid \sim \hat{p} \mid \hat{p}_1 \oplus \hat{p}_2 \mid P(\hat{t}_1, \hat{t}_2, \dots)$
Pattern spatial formula	\hat{s}	$::= \mathbf{emp} \mid \mathbf{data_at}(\hat{t}_1, \hat{t}_2) \mid A(\hat{t}_1, \hat{t}_2, \dots)$
Pattern formula	\hat{f}	$::= \hat{p} \mid \hat{s}$
Left pattern	q_l	$::= \hat{f} \text{ at } n$
Right pattern	q_r	$::= \text{exists } x, q_r \mid \hat{f} \text{ at } n$
Pattern	q	$::= \text{left: } q_l \mid \text{right: } q_r$
Check	c	$::= \mathbf{left_absent}(p) \mid \mathbf{right_absent}(p) \mid \mathbf{infer}(p)$
Operation	o	$::= \mathbf{left_add}(f) \mid \mathbf{right_add}(f) \mid \mathbf{left_erase}(n) \mid \mathbf{right_erase}(n) \mid \mathbf{forall_add}(x) \mid \mathbf{right_exist_add}(x)$
Action	a	$::= o \mid \mathbf{instantiate}(x \rightarrow t)$
Strategy	S	$::= \text{priority: } r$
		q
		check: c
		action: a
Program	$Prog$	$::= S$

Figure 9: Syntax of Stellis

$Prog$ consists of a sequence of strategies S . A strategy S has the following four elements:

1. A priority r , specified using the **priority** label.
2. A sequence of patterns q .
3. A sequence of checks c , denoted by the **check** label.
4. An action a , indicated by the **action** label.

The pattern part is used to identify specific formulas on both sides of the entailment. For right patterns, a pattern variable x may be constrained to bind an existential variable in the entailment via the syntax “exists x, q_r ”. The pattern formula \hat{f} follows the same syntactic structure as the formula f in the entailment, except that \hat{f} may contain $?x$, which introduce new pattern variables to bind to terms t in the entailment.

The check part ensures the entailment satisfies specific constraints. For example, **left_absent**(p) confirms the absence of a pure formula p in the antecedent, while **infer**(p) invokes an SMT solver to determine whether a pure fact p can be inferred from the antecedent.

The action part consists of two types: a sequence of operations o that manipulates the entailment by adding, removing, or introducing fresh variables, and **instantiate**($x \rightarrow t$), which instantiates an existential variable x with a term t .

We will present specific strategy examples in Appendix B.

B Examples: Full verification process for `s11_free` in QCP

In the actual verification setup, as shown in Figure 10, the process involves three header files (.h) and one implementation file (.c). The `free_node_list` function is imported via `s11_shape_def.h`. Upon completion, four output files

are generated: `sll_free_goal.v`, `sll_free_proof_auto.v`, `sll_free_proof_manual.v`, and `sll_free_goal_check.v`. It should be noted that `sll_free_proof_manual.v` remains an empty file in this process, as it can be automatically verified.

```

1 #include "verification_stdlib.h"
2 #include "verification_list.h"
3 #include "sll_shape_def.h"
4
5 void sll_free(struct list * x)
6 /*@ Require listrep(x)
7     Ensure emp
8 */
9 {
10     /*@ Inv listrep(x) */
11     while (x) {
12         /*@ exists v n, x -> data == v && x -> next == n && listrep(n) */
13         struct list * y = x -> next;
14         free_list_node(x);
15         x = y;
16     }
17 }

```

Figure 10: The real verification file of `sll_free`.

Regarding proof checking, we leverage Rocq's Module Type mechanism. In `sll_free_goal.v`, we generate a Module Type named `VC_Correct` that contains all verification goals (see Figure 11), including the correctness proof of the strategies and the proof of the verification conditions. Subsequently, in `sll_free_goal_check.v`, QCP automatically create a module `VC_Correctness` of type `VC_Correct` (see Figure 12). This module includes all contents from both `sll_free_proof_auto.v` and `sll_free_proof_manual.v`, thereby utilizing Rocq to ensure that every goal has a corresponding proof.

```

Module Type VC_Correct.
  Include common_Strategy_Correct.
  Include sll_shape_Strategy_Correct.

  Axiom proof_of_sll_free_entail_wit_1 : sll_free_entail_wit_1.
  Axiom proof_of_sll_free_entail_wit_2 : sll_free_entail_wit_2.
  Axiom proof_of_sll_free_entail_wit_3 : sll_free_entail_wit_3.
  Axiom proof_of_sll_free_return_wit_1 : sll_free_return_wit_1.
  Axiom proof_of_sll_free_partial_solve_wit_1 :
    sll_free_partial_solve_wit_1.
End VC_Correct.

```

Figure 11: `VC_Correct` Module Type for `sll_free` in `sll_free_goal.v`.

Although this example can be fully automatically solved by our solver, here we present the Rocq proof for the verification condition discussed in Section 2.2, which is `sll_free_entail_wit_2`. QCP provides a `pre_process` command for pre-processing verification conditions. After preprocessing, the proof becomes :

```

Module VC_Correctness : VC_Correct.
  Include common_strategy_proof.
  Include sll_shape_strategy_proof.
  Include sll_free_proof_auto.
  Include sll_free_proof_manual.
End VC_Correctness.

```

Figure 12: VC_Correctness Module for sll_free in sll_free_proof_check.v.

```

x : Z
H : x <> 0
-----
listrep x
⊢ EX v : Z, EX n : Z,
  [| x <> 0 |] &&
  &( x # "list" → "next") # Ptr |→ n ** listrep n **
  &( x # "list" → "data") # Int |→ v

```

Next, given that $x \neq 0$, which means $\text{listrep}(x)$ is non-empty, we can use `sep_apply listrep_nonzero` to unfold $\text{listrep}(x)$ into its non-empty form. The proof then becomes:

```

x :Z
H: x <> 0
-----
EX y : Z, EX a : Z,
&( x # "list" → "data") # Int |→ a **
&( x # "list" → "next") # Ptr |→ y ** listrep y
⊢ EX v: Z, EX n: Z,
  [|x <> 0|] &&
  &( x # "list" → "next") # Ptr |→ n ** listrep n **
  &( x # "list" → "data") # Int |→ v

```

Subsequently, by using `Intros y a` and `Exists a y`, we introduce the existential quantifier on the left-hand side and instantiate the one on the right-hand side. Finally, applying `entailer!` eliminates the separation logic components and completes the proof. Total proof scripts are shown below :

```

Lemma proof_of_sll_free_entail_wit_2 : sll_free_entail_wit_2.
Proof.
  pre_process.
  sep_apply listrep_nonzero.
  Intros y a. Exists a y.
  entailer!.
Qed.

```

At the end of this section, we present the Stellis strategy we introduced to solve this verification condition automatically. This strategy is applied only

when the left-hand side of the verification condition contains `listrep(p)` and `p != NULL`, and the right-hand side involves `p→data` or `p→next`. It works by expanding the `listrep(p)` on the left-hand side. Here, the notation `?p` indicates that the matched expression is bound to `p`, and this is not limited to variable scenarios. In the `left` and `right` patterns, we use `at` notation to simplify the matched expressions. This approach ensures that the `listrep` predicate is expanded only when load/store is required, avoiding unnecessary operations on unrelated predicates, thereby simplifying the assertion.

```

id: 8
priority: core(6)
left: listrep(?p : Z) at 0
      (p != NULL || NULL != p) at 1
right: (data_at(field_addr(p, list, data), I32, ?q : Z) || data_at(
        field_addr(p, list, next), PTR(struct list), ?q : Z)) at 2
action: left_erase(0);
        left_exist_add(x : Z);
        left_exist_add(y : Z);
        left_add(data_at(field_addr(p, list, data), I32, x));
        left_add(data_at(field_addr(p, list, next), PTR(struct list), y))
;
        left_add(listrep(y));

```

C Example: language design comparison between QCP and RefinedC

This example in Fig 13 is adapted from page 5 of the RefinedC paper [21]. In comparison, QCP’s annotation language is more intuitive and easier to read.

D Examples: Relational Proof in QCP

We decompose the proof of functional correctness into two parts: the correctness of the algorithm itself and the consistency of its implementation. The former can be established directly in Rocq, while the latter can be naturally formulated and verified using relational Hoare logic—a proof system for reasoning about the equivalence or refinement between two programs. Within this framework, we verify a merge sort function by implementing an abstract program using monads in Rocq. This approach employs an imperative-style formulation, which more effectively captures the algorithmic behavior to be verified. Figure 14 illustrates this abstract program formulation for the merge sort function.

By applying the encoding introduced by Shushu Wu, which transforms $\forall - \exists$ relational Hoare logic into standard Hoare logic, QCP is employed to prove that the implemented code refines the algorithm described by the abstract program. We obtain the encoded specification shown in Figure 15. Here, `safeExec(True, mergesortrec(1), X)` expresses that there exists an initial state satisfying `True`

```

1 typedef struct
2 [[rc::refined_by("s: {multiset nat}")]]
3 [[rc::ptr_type("chunks_t:"
4   "s ≠ 0) @ optional<down<...>, null>")]]
5 [[rc::exists ("n: nat", "tail: {multiset nat}")]]
6 [[rc::size ("n")]]
7 [[rc::constraints("s = {[n]} ∪ tail)",
8   "{V k, k ∈ tail → n ≤ k}")]
9 chunk {
10  [[rc::field("n @ int<size_t>")] size_t size;
11  [[rc::field("tail @ chunks_t")] struct chunk* next;
12 }* chunks_t;
13
14 [[rc::parameters("s:{multiset nat}", "p:loc", "n:nat")]]
15 [[rc::args("p @ &own<s @ chunks_t>", "&own<uninit<n>>",
16   "n @ int<size_t>")]]
17 [[rc::requires("sizeof(struct_chunk) ≤ n")]]
18 [[rc::ensures ("own p: {[n]} ∪ s @ chunks_t")]]
19 [[rc::tactics ("all: multiset_solver.")]]
20 void free(chunks_t* list, void* data, size_t sz) {
21  chunks_t* cur = list;
22  [[rc::exists ("cp: loc", "cs: {multiset nat}")]]
23  [[rc::inv_vars("cur: cp @ &own<cs @ chunks_t>")]]
24  [[rc::inv_vars("list:"
25   "p @ &own<wand<cp ↪y ([n]) ∪ cs @ chunks_t>,"
26   "{[n]} ∪ s @ chunks_t>>")]]
27  while(*cur != NULL) {
28   if(sz <= (*cur)->size) break;
29   cur = &(*cur)->next;
30  }
31  chunks_t entry = data;
32  entry->size = sz; entry->next = *cur;
33  *cur = entry;
34 }

```

```

1 typedef struct chunk {
2   size_t size;
3   struct chunk* next;
4 }*chunks_t;
5
6 /*@ Extern Ceq
7   (chunk_listrep : Z -> list Z -> Assertion)
8   (chunk_listseg : Z -> Z -> list Z -> Assertion)
9   (insert : Z -> list Z -> list Z -> Assertion)
10 */
11
12 void free(chunks_t* list, void* data, size_t sz)
13 /*@ With l
14   Require chunk_listrep(list, l) *
15   undef_data_at(&(data->size)) *
16   undef_data_at(&(data->next))
17   Ensure chunk_listrep(list, insert(l,sz))
18 */
19 {
20  chunks_t* cur = list;
21  /*@ Inv exists l1 l2,
22   l == l1 ++ l2 && sz == sz@pre &&
23   chunk_listseg(list@pre, *cur, l1) *
24   chunk_listrep(*cur, l2) *
25   undef_data_at(&(data->size)) *
26   undef_data_at(&(data->next))
27 */
28  while (*cur != NULL) {
29   if (sz <= (*cur)->size) break;
30   cur = &(*cur)->next;
31  }
32  chunks_t* entry = data;
33  entry->size = sz; entry->next = *cur;
34  *cur = entry;
35 }

```

(a) RefinedC example

(b) Version in QCP

Figure 13: Language Design comparison between RefinedC and QCP.

such that any final state after executing `mergesortrec(1)` satisfies **X**. In essence, this specification can be interpreted as follows: given a list `l` stored in a singly linked list at address `x`, upon termination of the function, we have effectively executed the abstract program `mergesortrec(1)` and returned 10, which is then stored in the linked list at `x`.

E Verification results for programs in QCP implementation components

Table 6: Verification results for QCP implementation components

File	Functions	Anno. (Lines)	Codes (Lines)	Auto VCs	Manual VCs	Manual Proof (Lines)	SE Time (s)	Rocq Time (s)
gmp_add	14	315	128	110	77	790	0.97	4.58
fme	9	253	165	142	38	522	3.59	9.78
typeinfer	3	366	109	110	35	400	0.10	11.76
alpha_equiv	1	105	33	35	22	228	1.01	45.58
subst	2	147	61	42	32	194	0.12	14.36
thm_apply	4	192	64	62	34	323	0.06	10.84
cnf_trans	3	379	195	358	49	1550	8.27	96.10

Table 6 presents the verification results for the QCP implementation components and Table 7 presents the verification properties for each file. In the following, we describe each algorithm individually and summarize the corresponding

```

Definition merge_body:
  list Z * list Z * list Z →
  MONAD (CntOrBrk (list Z * list Z * list Z) (list Z)) :=
  fun '(l1, l2, l3) ⇒
    match l1, l2 with
    | nil, _ ⇒ return (by_break (l3 ++ l2))
    | _, nil ⇒ return (by_break (l3 ++ l1))
    | x :: l1', y :: l2' ⇒
      choice
        (test' (x <= y);; return (by_continue (l1', l2, l3 ++ x :: nil)))
        (test' (y <= x);; return (by_continue (l1, l2', l3 ++ y :: nil)))
    end.

Definition merge_rel l l0 :=
  repeat_break merge_body (l, l0, nil).

Definition mergesortrec_f (W : (list Z) → MONAD (list Z) )
  : ((list Z) → MONAD (list Z)) :=
  fun x ⇒ '(p1, q1) <- split_rel x ;;
    match q1 with
    | nil ⇒ return p1
    | _ :: _ ⇒ p2 <- W (p1) ;;
      q2 <- W (q1) ;;
      merge_rel p2 q2
    end.

Definition mergesortrec := Lfix (mergesortrec_f).

```

Figure 14: Definition of mergesortrec.

```

struct list * merge_sort(struct list * x)
/*@ With l X
  Require safeExec(ATrue, mergesortrec(l), X) && sll(x, l)
  Ensure exists l0,
    safeExec(ATrue, return(l0), X) &&
    sll(_return, l0)
*/;

```

Figure 15: Relational specification of merge sort example.

verification outcomes. We omit the detailed Rocq definitions and present only the specifications of the principal functions in each file.

Table 7: Verification Properties for QCP Implementation Components

File	Verification Property
<code>gmp_add</code>	The <code>mpn_add</code> correctly performs high-precision addition calculations.
<code>fme</code>	If <code>lia_deduction</code> says a linear arithmetic entailment is correct, it is indeed correct.
<code>typeinfer</code>	The <code>atype_unify</code> always finds the best unification solution.
<code>alpha_equiv</code>	The <code>alpha_equiv</code> correctly determines alpha-equivalence.
<code>subst</code>	The <code>subst</code> correctly computes symbolic substitution.
<code>thm_apply</code>	The <code>thm_apply</code> correctly preforms theorem application.
<code>cnf_trans</code>	The <code>cnf_trans</code> correctly performs Tseitin transformation.

gmp_add: The file `gmp_add`, taken from the `minigmp` library, implements high-precision addition. We have formally verified the functional correctness of this algorithm.

```

unsigned int mpn_add (unsigned int *rp, unsigned int *ap,
                     int an, unsigned int *bp, int bn)
/*@ With val_a val_b
  Require an >= bn && an > 0 && bn > 0 &&
    mpd_store_Z(UINT_MOD, ap, val_a, an) *
    mpd_store_Z(UINT_MOD, bp, val_b, bn) *
    UIntArray::undef_full(rp, an)
  Ensure exists val_r_out,
    (val_r_out + __return * Z::pow(UINT_MOD, an) == val_a + val_b) &&
    mpd_store_Z(UINT_MOD, ap, val_a, an) *
    mpd_store_Z(UINT_MOD, bp, val_b, bn) *
    mpd_store_Z(UINT_MOD, rp, val_r_out, an)
*/

```

The store predicate for mini-gmp is `mpd_store_Z`. The specification states that if `val_a` (a large integer) is stored at address `ap` with length `an`, `val_b` (a large integer) is stored at address `bp` with length `bn`, and `an` is greater than `bn`, then upon function completion, the result of the addition will be stored starting at address `rp`, and the carry-out will be returned as the function's return value.

fme: The `fme` module implements the Fourier–Motzkin elimination algorithm, a quantifier elimination procedure used to solve systems of linear inequalities by iteratively removing variables and projecting the solution space onto remaining dimensions.

```

int lia_deduction(struct InequList** pr, int n)
/*@ With p1 l1

```



```

Require BP0 != 0 && pr != 0 &&
  n >= 1 && n <= INT_MAX - 1 &&
  BP0->upper == 0 &&
  BP0->lower == 0 &&
  BP0->remain == 0 &&
  LP_abs_in_int_range(n+1, l1) &&
  data_at(pr, p1) *
  InequList(p1, n + 1, l1)
Ensure ((__return == -1 &&
  data_at(pr, p1) *
  undef_data_at(&(BP0->upper)) *
  undef_data_at(&(BP0->lower)) *
  undef_data_at(&(BP0->remain)))
  ||
  (exists p2, __return == 1 && UNSAT(l1) &&
    data_at(pr, p2) *
    undef_data_at(&(BP0->upper)) *
    undef_data_at(&(BP0->lower)) *
    undef_data_at(&(BP0->remain)))
  ||
  (exists p2 l2, __return == 0 &&
    LP_implies(l1, l2) && InequList_Zeros(l2, 1, n + 1) &&
    data_at(pr, p2) *
    InequList(p2, n + 1, l2) *
    undef_data_at(&(BP0->upper)) *
    undef_data_at(&(BP0->lower)) *
    undef_data_at(&(BP0->remain))))
*/

```

Acting as the entry point for Fourier-Motzkin Elimination, the `lia_deduction` function's return value has specific meanings: a value of -1 signals a failure to complete the reduction; a value of 0 means the reduction finished but did not produce a contradiction (i.e., cannot prove UNSAT); and a value of 1 signifies a successful reduction to a contradictory system, conclusively proving the original problem is UNSAT.

The store predicate for inequations is `InequList`. The specification states that if a system of inequalities `l1` is stored at address `p1` with `n` variables, and if every coefficient in `l1` falls within the range of the `int` type, then this function will attempt to reduce the original inequality system according to the Fourier-Motzkin Elimination (FME) algorithm and return the corresponding result.

typeinfer: The `typeinfer` file implements a type inference algorithm, in which `atype_unify` performs type unification—a core subroutine that resolves equality constraints between type expressions by computing a most general unifier or detecting inconsistency.

```

int atype_unify(struct atype *t1, struct atype *t2)
/*@ With (s_pre : sol) (tr1 : TypeTree) (tr2 : TypeTree)
  Require

```

```

    store_compressed_solution(res, s_pre) *
    store_type(t1, tr1) *
    store_type(t2, tr2)
  Ensure
  (
    (
      exists s_post,
        __return == 0 &&
        sol_correct_iter(tr1, tr2, s_pre, s_post) &&
        store_compressed_solution(res, s_post) *
        store_type(t1, tr1) *
        store_type(t2, tr2)
    )
    ||
    (
      exists s_post,
        __return != 0 &&
        store_solution(res, s_post) *
        store_type(t1, tr1) *
        store_type(t2, tr2)
    )
  )
*/

```

Below are the Rocq definitions of `store_compressed_solution` and `sol_correct_iter`. The `store_compressed_solution` predicate indicates that a solution s' is stored in memory, where s is the compressed representation of s' and s is guaranteed to be finite and acyclic. The `sol_correct_iter` predicate expresses that the solution after unification is equivalent to the previous solution.

```

Definition sol_correct_iter (t1 t2: TypeTree)
  (s_pre s_post: sol): Prop :=
forall sf: sol,
  sol_compressed sf →
  sol_valid_eq t1 t2 sf /\ sol_refine s_pre sf ↔ sol_refine s_post sf.

Definition store_compressed_solution (x : addr) (s : sol) : Assertion :=
EX (s' : sol),
  [| sol_compress_to s' s |] && [| sol_finite s |] &&
  [| sol_no_loop s' |] &&
  store_solution x s'.

```

For the `atype_unify` function, a return value of 0 indicates that unification has completed successfully, and the unified solution is stored in the global variable `res`. Any non-zero return value indicates that unification has failed.

alpha_equiv: The `alpha_equiv` file implements an algorithm for determining α -equivalence between two terms—a fundamental operation in syntactic equality modulo bound variable renaming. The implementation of `term_alpha_eqn` is similar to `alpha_equiv`, so we have omitted the Rocq definitions here.

```

bool alpha_equiv(term *t1, term *t2)
/*@ With term1 term2
    Require store_term(t1, term1) *
           store_term(t2, term2)
    Ensure __return == term_alpha_eqn(term1, term2) &&
           t1 == t1@pre && t2 == t2@pre &&
           store_term(t1, term1) * store_term(t2, term2)
*/

```

The predicate `store_term(t1, term1)` indicates that the term `term1` is stored at address `t1`. The specification states that if addresses `t1` and `t2` store terms `term1` and `term2` respectively, then the function will return whether the two terms are alpha-equivalent.

subst: The `subst` file provides a substitution algorithm that systematically replaces free variables within a term while preserving binding structure and avoiding variable capture. The implementation of `term_subst_t` is similar to `subst_term`, so we have omitted the Rocq definitions here.

```

term *subst_term(term *den, char *src, term *t)
/*@ With trm src_str den_term
    Require den != 0 && src != 0 && t != 0 &&
           store_term(t, trm) *
           store_string(src, src_str) *
           store_term(den, den_term)
    Ensure den == den@pre && src == src@pre &&
           store_term(__return, term_subst_t(den_term, src_str, trm)) *
           store_term(den, den_term) *
           store_string(src, src_str)
*/

```

thm_apply: The `thm_apply` component implements theorem application, a core routine that matches the conclusion of a given theorem against a target goal and appropriately instantiates its premises to construct a derivation.

```

solve_res* thm_apply(term* thm, var_sub_list* lis, term* goal)
/*@ With t l g X
    Require thm != 0 &&
           safeExec(ATrue, thm_app_rel(t, l, g), X) &&
           store_term(thm, t) * store_term(goal, g) *
           sll_var_sub_list(lis, l)
    Ensure exists sr t,
           thm == thm@pre &&
           safeExec(ATrue, return(sr), X) &&
           store_term(thm, t) * store_term(goal, g) *
           sll_var_sub_list(lis, l) *
           store_solve_res(__return, sr)
*/

```

In this example, we apply the method outlined in Appendix D to construct an abstract program, `thm_app_rel`, and subsequently prove the consistency of the algorithmic implementation.

```

Definition thm_app:
  term * var_sub_list * term →
  MONAD (solve_res) :=
  fun '(t, l, g) ⇒
  match thm_subst_allres t l with
  | None ⇒ ret (SRBool 0)
  | Some (_, thm_ins) ⇒
    if (term_alpha_eq thm_ins g) then ret (SRBool 1)
    else x <- (check_rel thm_ins g) ;; get_list x
  end.

Definition thm_app_rel (thm : term) (l : var_sub_list) (goal : term) :=
  thm_app (thm, l, goal).

```

cnf_trans: The `cnf_trans` file implements the Tseitin transformation algorithm, which converts arbitrary propositional formulas into conjunctive normal form by introducing auxiliary variables and clauses while preserving satisfiability. The implementation of `prop2cnf_logic` is similar to `prop2cnf`, so we have omitted the Rocq definitions here.

```

int prop2cnf(SmtProp *p, PreData *data)
/*@ With prop clist pcnt ccnt
  Require prop_cnt_inf_SmtProp(prop) <= pcnt &&
    SmtProp_size(prop) <= 10000 &&
    Zlength(clist) <= 40000 - 4 * SmtProp_size(prop) &&
    pcnt <= 40000 - SmtProp_size(prop) &&
    store_SmtProp(p, prop) *
    store_predata(data, clist, pcnt, ccnt)
  Ensure exists clist' pcnt' ccnt' res,
    make_prop2cnf_ret(make_predata(clist', pcnt', ccnt'), res) ==
    prop2cnf_logic(prop, make_predata(clist, pcnt, ccnt)) &&
    __return == res && res != 0 && res <= pcnt' && -res <= pcnt' &&
    Zlength(clist') <= Zlength(clist) + 4 * SmtProp_size(prop) &&
    pcnt' <= pcnt + SmtProp_size(prop) &&
    store_SmtProp(p, prop) *
    store_predata(data, clist', pcnt', ccnt')
*/

```

F Example: multiple specifications and specification derivation

We use the `swap` function as an example to illustrate the different approaches of multiple specifications and specification derivation. As is well known, the `swap` function has two specifications in verification scenarios: one where the input

parameters occupy the same address and another where they occupy different addresses. Using the multiple specifications method, this can be expressed as follows:

```
void swap(int * px, int * py)
/*@ requires integer(px, ?x) && integer(py, ?y);
    @@ ensures integer(px, y) && integer(py, x);
    @@ requires px == py && integer(px, ?x);
    @@ ensures px == py && integer(px, x);
```

Based on specification derivation, we can express it as follows:

```
void swap(int * px, int * py)
/*@ all
    With para
    Require swap_pre(px, py, para)
    Ensure swap_post(px, py, para)
*/;

void swap(int * px, int * py)
/*@ neq <= all
    With x y
    Require x == *px && y == *py
    Ensure y == *px && x == *py
*/;

void swap(int * px, int * py)
/*@ eq <= all
    With x
    Require px == py && x == *px
    Ensure x == *py
*/;
```

Here, we introduce `para` to distinguish between the cases of same address and different addresses. In the actual proof of `all` specification, we manually perform case analysis on `para` to expand it into the two scenarios, and then complete subsequent symbolic execution and verification condition solving within the multiple branches. In this example, since the different specifications are entirely independent and cannot be derived from one another, using multiple specifications is more convenient. Although we have demonstrated how to achieve a similar effect using specification derivation, QCP will also support multiple specifications in future versions.