# A Proof-Integrated Low-Level Programming Language with Local, Operational, and Extensible Reasoning

ANONYMOUS AUTHOR(S)

Verifying the functional correctness of safety-critical software remains a grand challenge. Existing verifiers for low-level pointer-manipulating programs fall into two camps: proof frameworks embedded in interactive theorem provers offer powerful proving facilities but separate verification from programming; while assertion-based verifiers enable verification in place but lack proof support.

We present C*, a proof-integrated language and verifier design that addresses this dichotomy between programming and proving by bringing local, operational, and extensible reasoning into a C programming-integrated interface. Specifically, C* introduces a *capability-passing* design where proof states are localized as ghost variables called capabilities, enabling users to perform local reasoning steps by transforming capabilities through proof code. C* provides a *sound LCF-style proof interface in C* for programming proof code and building reusable proof libraries in the same language as for writing implementation code.

We implement C* as a prototype verifier for a subset of C and evaluate it on a suite of C programs from existing benchmark. Our evaluation demonstrates that C* handles typical low-level program verification tasks while effectively supporting local and operational reasoning and enabling users to extend proof support in C.

## 1 Introduction

Verifying the functional correctness of safety-critical software is a long-standing challenge [2, 4, 24, 26, 30, 32, 33, 53]. Over the past few decades, built upon the foundations of separation logics and their variants [42, 46], a rich body of verifiers for low-level pointer-manipulating programs has emerged [6, 9, 11, 12, 16, 18, 23, 28, 29, 34, 39, 41, 44, 49, 52]. These verifiers primarily follow one of two designs, creating a dichotomy between *proof-centric* and *program-centric* verification.

- *Proof-Centric Verifiers.* These verifiers (e.g., VST [9] and Iris [29]) are program-logic frameworks embedded in interactive theorem provers (e.g., Rocq [7]), inheriting their highly expressive specification logics and comprehensive proving facilities. They provide users with complete control over the proof state and are extensible in terms of proof support via tactic languages [15] or programming upon a proof interface [20].
- *Program-Centric Verifiers.* These verifiers (e.g., VeriFast [28] and CN [41]) follow the Hoare tradition [27] of decorating programs with intermediate assertions as a proof outline. A verification algorithm [17] reconstructs full program-logic derivations from decorated programs, generates missing proof steps as verification conditions (VCs), and attempts to discharge them automatically through heuristics, proof search, and SMT solving.

We argue that this design dichotomy between programming-integration and powerful proving is artificial and hinders the development of verified software. Program-centric verifiers are more developer-friendly, but they lack two key features of their proof-centric counterparts:

- *Lack of Local and Operational Reasoning.* Current program-centric verifiers treat the proof state as an implicit *global* context. Users perform *declarative* reasoning steps by annotating assertions, hoping the verifier can prove an entailment from the preceding proof state to the assertion. They cannot perform *local and operational reasoning* by giving proofs directly, such as "rewrite a specific part of the proof state using this equational assumption." By not requiring the resulting assertions upfront, this local and operational style of reasoning is more programmatic and robust to changes in the implementation code [23].
- *Lack of Extensible Proof Support.* The inherent incompleteness of automatic VC discharging leads to the need for users to perform manual reasoning steps. Some program-centric verifiers (e.g., VeriFast and CN) support a fixed set of *proof directives*, such as `fold` or

`unfold` of logical predicates or `apply` lemmas. However, they do not provide users with an easy way to extend this support. In contrast, proof-centric verifiers allow users to package high-level reasoning patterns as reusable proof procedures.

To bridge this gap, we propose C*, a proof-integrated language design and verifier for (a subset of) C. Our goal is to bring the local, operational, and extensible reasoning of proof-centric frameworks into a C programming-integrated interface. There are two major challenges:

- *Localizing and Manifesting the Proof States.* How do we break up the implicit, global proof state of current program-centric verifiers into explicit, localized fragments that allow operational updates by proof? We need a language design that reifies the proof state locally as resources, and soundly tracks these resources and involved logical variables (e.g., universal and existential quantifiers) across control-flow paths, while also allowing users to write proof code to update them.
- *Taming Unsafe C Code in Proof Programming.* How do we provide extensible proof support in a C programming-integrated design? The natural answer is to let users write proof code in C itself. The challenge is that the established LCF architecture [22] for sound proof programming relies on enforcing *type abstraction* [21] to protect the proof kernel from arbitrary user proof code. But the low-level nature of C programming (e.g., direct memory manipulation and unchecked type casts) offers no such type safety guarantees.

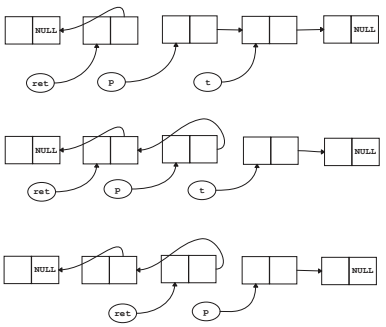
To address the first challenge, we propose **a capability-passing language design** (Section 3). We enrich C with two kinds of ghost entities: capabilities and reasoning blocks. *Capabilities* are separation-logic assertions explicitly tracked as ghost variables in a flow-sensitive manner. Our language design ensures that at a program point, each available capability is a fragment of the global proof state, and the separating conjunction of all available capabilities forms the whole proof state. These capabilities can be passed around separately, clearly manifesting local reasoning dependencies. *Reasoning Blocks* are ghost constructs that allow users to transform capabilities by providing proof code. These blocks explicitly consume capabilities, run the provided proof code to perform logical deductions, and produce new capabilities. These reasoning blocks can be either *declarative* (stating the produced assertions upfront) or *operational* (programmatically calculating the produced assertions from the consumed ones).

To enable users to write proof code in C within reasoning blocks—as well as address the second challenge—we devise **a sound mechanism for C-programmable proof support** (Section 4). We provide an LCF-style proof interface [20, 38] in C, with opaque types like `thm` (a proved theorem) and `term` (a logical term), allowing users to manipulate assertions and theorems as data in reasoning blocks. The interface provides term-manipulation utilities as well as proof functions that implement logical inference rules to compute theorems. To ensure soundness in the context of unsafe C programming, we adapt the longstanding LCF architecture by protecting the proof interface through *process isolation* (instead of type abstraction): user-written C proof code and the trusted proof kernel run in separate processes, communicating through opaque handles that are created and stored exclusively by the kernel. This design protects the proof kernel from arbitrary C code, ensuring theorems computed by user code are always valid.

We implement a prototype verifier for C* and evaluate it on a suite of C programs from an existing benchmark [23] (Section 5). Our evaluation demonstrates that C* successfully verifies all benchmark programs, which span simple algorithms, low-level memory manipulations, and linked data structures. The verification leverages C*'s support for local and operational reasoning: our metrics show that reasoning blocks typically operate on only a small fraction of the available proof state, and operational reasoning blocks are frequently employed throughout the verification process. These verifications are supported by user-extensible proof libraries written in C: we build

```
1  int32_t reverse(int32_t p) {
2      int32_t ret = 0;
3      while (p) {
4          int32_t t;
5          t = load32(p+4);
6          store32(p+4, ret);
7          ret = p; p = t;
8      }
9      return ret;
10 }
```

(a)




(b)

Fig. 1. The C function `reverse` and its intuitive correctness argument in a loop iteration.

general-purpose libraries for common reasoning patterns (e.g., rewriting strategies and backward reasoning tactics) and domain-specific libraries for linked data structures and array reasoning, demonstrating the practical benefits of C-programmable proof support.

To sum up, in this work, we make the following contributions:

- We propose C*, a proof-integrated language design that unifies program-centric verification with explicit, local, and operational reasoning by proof. We devise its syntax-guided typing rules and prove its soundness with respect to separation logic.
- We devise a sound mechanism for C-programmable proof support via process isolation and, based on it, implement a prototype verifier for C*.
- We evaluate C* on a suite of C programs, demonstrating its ability to handle typical low-level program verification tasks and its benefits in enabling local reasoning and building reusable proof libraries in C.

## 2 Overview of C*: A Guided Tour

This section provides an overview of C*, demonstrating its capability-passing verification style and its C-programmable proof support through a running example.

*Running Example: In-Place List Reversal.* We use the classic pointer-manipulating algorithm of reversing a singly linked list in place as our running example. Its C implementation is given in Figure 1(a). C* adopts a subset of C with a single integer type **`int32_t`** representing both integers and pointers (using a concrete memory model). A node is represented on the heap as two consecutive integers, storing the data and the pointer to the next node, respectively.

The loop maintains the invariant that the list is split into a processed part (the reversed prefix pointed to by `ret`) and a remaining part (the suffix pointed to by `p`). The loop body, illustrated in Figure 1(b), makes progress in three steps: saving the next node address in `t` (line 5), prepending the current node to the processed part (line 6), and updating `p` and `ret` to the new heads (line 7).

We present our verified C* version[1] of `reverse` in Figure 2 with slightly prettified syntax.[2] It manifests local reasoning dependencies as capability-passing and logical deductions as proof code, interspersing verification clauses with the original implementation code. We explain its verification in the following paragraphs.

[1]Full code and supporting proof libraries are available in the supplementary material.

[2]We show logical terms in sans-serif, italic font (e.g., *term*) and put capabilities in boxes (e.g., `own_list:`*sll_at p l* ).

```
1  forall(p: int, l: int list)  parameter(i32[p])  require( own_list:sll_at p l ) // input specification
2  exist(ret: int)  return(i32[ret])  ensure( own_rev_list:sll_at ret (rev l) ) // output specification
3  int32_t reverse( alloc( p_var ) int32_t p ) {
4      alloc( ret_var ) int32_t ret=0;
5      // establish invariant by producing required capabilities
6      produce( own_rev_list:sll_at 0 [] ) {|DC_REWRITE(ThmList(sll_at_def));|}
7      produce( inv:fact(l=app (rev []) l) ) {|DC_REWRITE(ThmList(rev_def, app_def));|}
8
9      exgiven(ret: int, p: int, l1: int list, l2: int list)
10     invariant( ret_var:i32_at &ret ret , p_var:i32_at &p p ,
11                own_rev_list:sll_at ret l1 , own_list:sll_at p l2 , inv:fact(l=app (rev l1) l2) )
12     while (p pathcond( cond_p )) {
13         alloc( t_var ) int32_t t=0;
14
15         consume( own_list , cond_p ) // split ownership of the non-empty, unprocessed part
16         given(next: int, data: int, tail: int list) produce( l2_is_cons , data_mem , next_mem , own_list )
17         {|OP_APPLY(sll_cons_destr, TermList(own_list, cond_p));|}
18
19         t = load32(p+4 check( next_mem ));
20         consume( next_mem ) produce( next_mem ) store32(p+4, ret);
21
22         // merge the new node into the reversed prefix
23         consume( data_mem , next_mem , own_rev_list ) produce( own_rev_list )
24         {|OP_APPLY(sll_cons_constr, TermList(data_mem, next_mem, own_rev_list));|}
25
26         ret = p; p = t; // re-establish invariant
27         consume( inv , l2_is_cons ) produce( inv:fact(l=app (rev (data::l1)) tail) )
28         {|DC_REWRITE(ThmList(..., gsym_rule(by_fact(l2_is_cons)), by_fact(inv)));|}
29     }
30     consume( cond_p , own_list ) produce( l2_is_nil ) // destruct empty ownership
31     {|OP_APPLY(sll_nil_destr, TermList(own_list, cond_p));|}
32     consume( l2_is_nil , inv ) produce( l1_is_rev:fact(l1=rev l) ) // list reasoning
33     {|DC_REWRITE(ThmList(by_fact(inv), by_fact(l2_is_nil), app_nil, rev_rev));|}
34     consume( own_rev_list , l1_is_rev ) produce( own_rev_list ) // produce output capability
35     {|OP_REWRITE(ThmList(by_fact(l1_is_rev)), TermList(own_rev_list));|}
36     return ret; // return result as well as the required capability
37 }
```

Fig. 2. Verified reverse in C*.

*Function Specification.* The function input and output specifications are given in lines 1 and 2, respectively. The `forall` clause introduces universally quantified logical variables: $p$ of type *int* (an unbounded mathematical integer) and $l$ of type *int list* (an inductive list of integers). The `parameter` clause assigns a refined singleton type `i32[`$p$`]` to the parameter `p`, relating its initial value to the logical variable $p$. The `require` clause specifies additional preconditions to invoke the function as ghost capability inputs. Capabilities are labeled separation-logic assertions that reify localized

fragments of the global proof state. Here, `own_list:sll_at p l` reifies that list *l* is stored at address *p* as a singly linked list; this capability will be available when verifying the function body and can be passed around explicitly or transformed in reasoning steps.

The output specification (line 2) must be satisfied on return. Symmetric to the input specification, `exist` introduces existential variables, `return` specifies the returned value, and `ensure` specifies additional guarantees as ghost capability outputs. Here, it returns to the caller with a heap location *ret* and a capability `own_rev_list:sll_at ret (rev l)`, stating the reversed list is stored at *ret*.

*User-Defined Specification Functions and Predicates.* Specification functions and predicates are defined in our specification logic, a polymorphic higher-order logic [3] with a bespoke separation logic theory. For example, *rev* is defined by equations *rev []=[]* and *rev (x::l)=app (rev l) [x]*. The predicate *sll_at* is defined by

$$\textit{sll\_at p []} = \textit{fact(p=0)} \quad \text{and} \quad \textit{sll\_at p (h::t)} = \exists q{:}\textit{int}.\ \textit{i32\_at p h} \star \textit{i32\_at (p+4) q} \star \textit{sll\_at q t}\,,$$

where ⋆ is separating conjunction, *i32_at addr val* is a basic points-to predicate that asserts *val* (within the range of **`int32_t`**) is stored at *addr*, and *fact(p)* embeds a pure proposition *p* into separation logic while asserting an empty heap. Users can extend data types, functions, and predicates on a per-module basis via definitional mechanisms in our C proof interface (see Section 4).

*Capability-Passing: Allocation and Assignment of Local Variables.* Each parameter and local variable declaration is accompanied by an `alloc` clause that associates a capability representing ownership of the variable.[3] For example, in line 3, `p_var` is associated with parameter `p`, initially set to *i32_at* `&p` *p*, where `&p` is the address of `p` and the initial value *p* is read from the parameter type `i32[p]`. On assignment, the type checker implicitly consumes and updates the variable's capability, tracking its current value. For example, assigning to `t` in line 19 transforms `t_var:i32_at &t 0` to `t_var:i32_at &t next`, reflecting that `t` now stores the next node pointer loaded from `load32(p+4)`.

*Capability-Passing: Path Conditions.* Conditional tests (e.g., in **`if`** or **`while`** statements) generate capabilities indicating the truth or falsity of the condition. Unlike traditional assertion-based verifiers where these *path conditions* are gathered implicitly in the internal proof state without explicit ways to refer to them, in C* they are explicitly bound to capabilities via a `pathcond` clause. The nullity test on `p` in line 12 creates `cond_p`, bound to *fact(p≠0)* inside the loop body and *fact(p=0)* at loop exit, allowing explicit reference in reasoning steps. For example, in line 15, `cond_p` serves as an explicit reasoning dependency to transform `own_list` via the unfolding lemma `sll_cons_destr` (stating that a linked list at *non-null* pointer address can be split into a head node and the remaining list); see the *Operational Reasoning Steps* paragraph below for details on this transformation.

*Capability-Passing: Function Calls.* Function calls pass required capabilities as ghost arguments with a `consume` clause and ensured capabilities as ghost returns with a `produce` clause. For example, the primitive function `store32` (with C prototype `void store32(int32_t,int32_t)`) is specified as:

`forall(p: int, w: int, v: int) parameter(i32[p],i32[v]) require( in:i32_at p w ) ensure( out:i32_at p v ).`

When `store32` is called (e.g., line 20), call sites provide a capability matching the assertion of `in` as input (universal quantifiers are automatically instantiated) and receive a capability instantiating the assertion of `out` as output. The general form of function calls also allows manual instantiation of universal quantifiers and binding of output existential quantifiers (see Section 3).

[3] Technically, for uniformity, we treat all program variables as addressable, i.e., they can be taken addresses and aliased.

```
void DC_REWRITE(thm* ths)
{   GN root = Init_goal(get_goal());
    GN g = FACTS_INTRO_TAC(root);
    g = REWRITE_LIST_TAC(g, ths);
    g = HCLEAN_TAC(g); AUTO_FRAME_TAC(g);
    by_using(Solve(root)); }
```

(a) Declarative rewrite.

```
void OP_REWRITE(thm* ths, term* tms) {
    term ant = list_mk_hsep(tms);
    thm eq = rewrite_list(ths, ant);
    thm trans = eq2ent(eq);
    by_using(trans);
}
```

(b) Operational rewrite.

Fig. 3. Example proof functions used in declarative and operational reasoning blocks.

Pure functions that do not modify the proof state can be called under expression contexts; we call these *operator functions*. For example, the primitive function `load32` is specified as:

`forall(`*p*: *int*, *v*: *int*`) parameter(i32[`*p*`]) check(` `in:`*i32_at p v* `) return(i32[`*v*`])`.

Note that the `require` and `ensure` clauses are replaced by a `check` clause, specifying the capabilities that are required and produced *as is*. It is invoked in line 19, where a `check` clause is put inside the parameter list.

*Loop Invariants.* The separate pieces of information in the informal loop invariant are formalized as capabilities in the `invariant` clause (lines 10-11): `own_rev_list` asserts ownership of the processed part *l1* at `ret`; `own_list` asserts ownership of the remaining part *l2* at `p`; `inv` relates *l1*, *l2*, and initial list *l*; `ret_var` and `p_var` represent ownership of the local variables `ret` and `p`. The `exgiven` clause (line 9) abstracts current values of `p`, `ret`, and lists *l1*, *l2* as existential variables shared among invariant capabilities, and immediately introduces them into scope for future reference.

*Reasoning Blocks: Transforming Capabilities by Proof.* Reasoning blocks perform local reasoning steps: explicit reasoning dependencies are specified in `consume` and `produce` clauses, and the rest of capabilities in scope remain unchanged, reflecting the framing principle of separation logic. The proof code that performs logical deductions on capabilities is wrapped in {| . . . |} delimiters. Reasoning blocks come in two styles: *declarative* ones (e.g., lines 6,7, 27-28) annotate produced capabilities with assertions explicitly, while *operational* ones (e.g., lines 15-16, 23-24) derive output assertions by proof code, requiring only names of the produced capabilities.

*Declarative Reasoning Steps: Establishing the Loop Invariant.* To see declarative reasoning steps in action, consider the initial establishment of the loop invariant. Initially, the processed part is empty and the remaining part is the entire input list (represented by the required capability `own_list`), hence the invariant is established by producing the empty list ownership capability `own_rev_list:`*sll_at 0 []* and the fact `inv:`*fact(l=app (rev []) l)*. The declarative reasoning blocks produce these capabilities in lines 6 and 7, respectively.

Take, for example, the reasoning block in line 6. Since no input capabilities are consumed, the proof obligation is *emp* ⊢ *sll_at 0 []*, where ⊢ stands for separation-logic entailment and *emp* asserts an empty heap with no additional information. This entailment can be directly solved by a proof function `DC_REWRITE` (abbreviation for "declarative rewrite"). Its definition is shown in Figure 3(a), which builds upon our library for goal-directed proof tactics (see Section 4). It encapsulates the automation process of (1) fetching the current proof obligation using `get_goal()` (which is possible because a declarative block requires all output capabilities to be annotated with assertions upfront) and initializing it as a goal (stored at `root` of type `GN` for *goal nodes*), (2) performing iterative rewriting using the input equational theorems (here, the definition of *sll_at*), clean-up of trivial

conjuncts (e.g., *emp*), and cancellation of matching separating conjuncts, until the goal is solved, and (3) returning the proven theorem using `by_using`.

*Operational Reasoning Steps: Splitting and Merging of Ownership of the Linked Lists.* To see operational reasoning steps in action, consider the splitting and merging of ownership of the linked lists in lines 15-17 and 23-24. These steps parallel the dynamic update in line 20, i.e., a node is picked from the remaining part and prepended to the processed part. For example, lines 15-17 perform the splitting of ownership of the remaining part by applying the unfolding lemma `sll_cons_destr` to `own_list`, which proves the following entailment (its proof is shown in Section 4):

$$\forall x, l.\ (\mathit{sll\_at}\ x\ l \star \mathit{fact}(x{\neq}0) \vdash \exists y, h, t.\ \mathit{fact}(l{=}h{::}t) \star \mathit{i32\_at}\ x\ h \star \mathit{i32\_at}\ (x{+}4)\ y \star \mathit{sll\_at}\ y\ t)\ .$$

The function `OP_APPLY` (abbreviation for "operational apply") performs automatic instantiation of the lemma by matching the separating conjuncts in the antecedent against the assertions recorded in the input capabilities and produces the resulting transformation entailment theorem:

$$\mathit{sll\_at}\ p\ l \star \mathit{fact}(p{\neq}0) \vdash \exists y, h, t.\ \mathit{fact}(l{=}h{::}t) \star \mathit{i32\_at}\ p\ h \star \mathit{i32\_at}\ (p{+}4)\ y \star \mathit{sll\_at}\ y\ t\ .$$

Our verifier will check the produced entailment theorem has an antecedent that matches the input capabilities; after that, the existential variables in the consequent are introduced as logical variables *next*, *data*, *tail* (as per the `given` clause) and the separating conjuncts in the consequent are bound to the capabilities `l2_is_cons`, `data_mem`, `next_mem`, and `own_list` (as per the `produce` clause). Note that no output assertions are stated in operational blocks—assertions recorded by output capabilities are "calculated" from the input capabilities by proving an entailment theorem.

*Reasoning Steps: Returning from the Function.* After the loop exits, we perform several more steps of operational destruction of empty list ownership and rewriting of the remaining capabilities using equational facts. For example, in lines 34-35, we rewrite `own_rev_list:`*sll_at ret l1* into the desired form `own_rev_list:`*sll_at ret (rev l)* specified in the `ensure` clause using the fact that `l1_is_rev:`*fact(l1=rev l)*. This operational rewriting step is performed by the function `OP_REWRITE` in Figure 3(b), which first constructs a symbolic heap from the input assertions `tms` as the antecedent (using `list_mk_hsep` to build up the term with separating conjunction in a right-associative manner), then performs rewriting on it using the interface `rewrite_list` to produce an equality `eq` between the antecedent and the rewritten consequent, and finally transforms this equation into an entailment theorem by calling `eq2ent` and returns it (i.e., *sll_at ret l1* $\vdash$ *sll_at ret (rev l)*) using `by_using`.

## 3 C* Language Design

This section presents the core design of C*. We introduce the overall structure and general concepts underlying our language design in Section 3.1. We then present the capability-passing language constructs of C* in Section 3.2. We describe the formalization of C* in Section 3.3. Full grammar definitions, typing rules, and proofs of meta-properties can be found in the supplementary material.

### 3.1 General Concepts

Our proof-integrated language has three layers: the implementation layer, the specification layer, and the proof layer. We describe each layer in the following.

*Implementation Layer: A Subset of C for Low-Level Programming.* Following Gruetter et al. [23], implementation code is written in a subset of C where all variables have type **`int32_t`**, representing both integers and pointers. Expressions must be pure (no side effects on memory).

We support: pointer arithmetic (i.e., calculated addresses), address-of (`&e`), dereferencing (`*e`), blocks (which introduce local scopes), and structured control-flow constructs (**`if`**, **`while`**, **`continue`**,

**break**, **return**). Fine-grained byte-level memory access is supported via primitive functions like `store8` and `load8`, with specifications matching the desired dynamic semantics. This subset is minimal yet practical enough for writing interesting low-level heap-manipulating programs.

*Specification Layer: Higher-Order Logic with Separation-Logic Theory.* Our design maintains a clear distinction between (static) logical terms and (dynamic) program expressions; hence, terms in the specification denote immutable, mathematical values, independent of the program state.

For this specification layer, we choose a standard higher-order logic (HOL) [3] for its simplicity and expressiveness, and axiomatize a separation-logic theory (with a concrete memory model) for heap-dependent assertions. Every logic term has a *sort*, which can be annotated explicitly by syntax *term:sort*. To list a few examples: integer terms have sort *int*, e.g., *42*, *x+y*; logical functions have arrow sorts, e.g., *rev: int list→int list*; boolean values and propositions have sort *bool*, e.g., *false*, a quantified proposition $\forall x{:}int.\ \exists y{:}int.\ y{>}x$, a separation-logic entailment $P \star Q \vdash Q \star P$; separation-logic assertions have sort *hprop* (roughly understood as *heap→bool*), e.g., an empty heap assertion *emp*, an embedded pure proposition *fact(p:bool)*, a separating conjunction $P \star Q$, a basic points-to predicate *i32_at addr val*.

*Proof Layer: Full-Featured C with an LCF Proof Interface.* Our proof layer shares the same programming language as the implementation layer, C. More precisely, the proof language is full-fledged C with an LCF-style programmable proof interface for HOL (detailed in Section 4). Theoretically, our type checker is agnostic of the concrete form of proof code (as long as it establishes the required entailment theorem; see Section 3.3); nevertheless, this choice allows users to compose proof code and encapsulate reasoning patterns in the same general-purpose programming language as for writing implementation code.

*Capabilities: Labeled Assertions as Linear Resources.* A *capability* is a separation-logic assertion[4] tracked by a label in a flow-sensitive manner, reifying information known to hold at a program point (e.g., ownership predicates, path conditions, or pure logical facts). Our capability-passing language design maintains the invariant that the proof state during verification is the (iterated) separating conjunction of all capabilities available at each program point. Capabilities thus represent localized fragments of the global proof state. Due to the resource-aware nature of separation-logic assertions, capabilities must be used *linearly*: a passed-in capability (via `consume`) is no longer available unless explicitly returned and updated (via `produce`). Capabilities cannot be duplicated or discarded at will except when recording a *pure assertion* [42], e.g., *fact(p=0)*. Capabilities can be transformed by sound logical deductions via reasoning blocks.

Our notion of capability is inspired by the $L^3$ language [1] (and related work [10, 47]), where capabilities are linear ghost variables reifying ownership for sound strong updates of reference types in a higher-order functional setting. In C*, capabilities track both ownership and content of memory locations for functional-correctness verification of first-order, low-level programs.

### 3.2 Language Constructs

We now describe the key constructs of C* with their concrete syntax and their static semantics in prose descriptions. Our syntax is a lightweight extension of C, where all verification clauses are interleaved in normal C language constructs as special comments wrapped in `/*@...@*/` delimiters. This design allows verified C* programs to be compiled directly using standard C compilers. To make the distinction between different language layers explicit, our syntax requires wrapping logical terms in backticks `` `...` `` and all proof code in `{|...|}` delimiters. We use the following

[4] Assertions recorded in capabilities need not be simple ownership predicates; they can have arbitrary internal structure (e.g., a disjunction or a separating implication).

meta-variable conventions in this subsection: $c, l$ for capability labels; $u, v$ for general HOL terms; $P, Q$ for separation-logic assertions (i.e., terms of sort *hprop*); and $s, t$ for arbitrary HOL sorts.

*Function Declarations.* A function declaration in C* starts with a `__NORMAL__` token (for parsing purposes) followed by six clauses specifying its behavior: The first three clauses specify function inputs: `forall` introduces logical-layer parameters (universal quantifiers); `parameter` assigns each implementation-layer parameter a singleton refined type relating its initial value to a logical term; and `require` specifies required capability parameters (preconditions). The last three clauses specify function outputs: `exist` introduces logical return variables (existential quantifiers); `return` specifies the computational return value; and `ensure` specifies guaranteed capability outputs (postconditions). A function definition is a declaration followed by a body; moreover, since parameters are treated as local variables, each computational parameter is accompanied by an `alloc` clause.

```
__NORMAL__
/*@ forall(`x1 : s1`, ...) @*/
/*@ parameter(i32[`u1`], ...) @*/
/*@ require(c1 : `P1`, ...) @*/
/*@ exist(`y1 : t1`, ...) @*/
/*@ return(i32[`v`]) @*/
/*@ ensure(l1 : `Q1`, ...) @*/
int32_t f(int32_t, ...);
```

Fig. 4. Function declaration.

*Function Calls and Assignments.* Function call statements use conventional C syntax $e_l = f(e_1, \ldots)$ for passing computational arguments and receiving return values. Corresponding to function specifications, function calls additionally pass logical terms via `instantiate` clauses (instantiating universal quantifiers; auto-inferred by the type checker if omitted) and required capabilities via `consume` clauses. They bind return logical variables via `given` clauses (eliminating existential quantifiers and introducing them into scope until the end of the enclosing block) and guaranteed capabilities via `produce` clauses. The assignment to $e_l$ implicitly consumes the capability for the l-value (i.e., the *i32_at* predicate for the designated address) and produces a new capability with updated content. An assignment statement is a degenerate case of a function call.

```
/*@ instantiate(`u1`, ...) @*/
/*@ consume(c1, ...) @*/
/*@ given(`y1`, ...) @*/
/*@ produce(l1, ...) @*/
el = f(e1, ...);
```

Fig. 5. Function call.

*Operator Functions. Operator functions* are a subset of functions that can be invoked directly in expression contexts. Their declaration syntax starts with a `__OPFUN__` token. Since C* requires pure expressions, operator functions omit `require` and `ensure` clauses; instead, a `check` clause specifies capabilities that must be available (but are not consumed or modified). We disallow the `exist` clause as well, yielding simpler call syntax $f(e_1, \ldots /*@\ \texttt{check}(c_1, \ldots) @*/)$. In fact, pointer dereferencing and all C arithmetic and comparison operators in C* are syntactic sugars for built-in operator functions with appropriate `check` clauses.[5] For example, $e_1 + e_2$ desugars to `op_add(`$e_1, e_2$`/*@ check(c) @*/)`, where `c` should record the fact that the result does not overflow. This capability is filled in by the type checker when desugaring the C operator syntax, and can be passed by users explicitly if desired.

```
__OPFUN__
/*@ forall(`x1 : s1`, ...) @*/
/*@ parameter(i32[`u1`], ...) @*/
/*@ check(c1 : `P1`, ...) @*/
/*@ return(i32[`v`]) @*/
int32_t f(int32_t, ...);
```

Fig. 6. Operator-function declaration.

*Reasoning Blocks.* The syntax of reasoning blocks resembles that of function calls, but instead of performing dynamic computation at the implementation level, it performs logical deductions on capabilities by proof code (wrapped inside `{|...|}`). Consumed capabilities are available in the proof code as `term` variables, recording assertions. The proof should establish a separation-logic entailment of

```
/*@ consume(c1, ...) @*/
/*@ given(`y1`, ...) @*/
/*@ produce(l1, ...) @*/
/*@ {| proof code |} @*/
```

Fig. 7. Reasoning block (operational).

[5] We do not use short-circuiting semantics for the && and || operators.

the following form and pass it to the type checker by `by_using`:

$$P_1 \star \cdots \star P_m \vdash \exists x_1 : s_1, \ldots, x_k : s_k . Q_1 \star \cdots \star Q_n \ .$$

The type checker receives the theorem and ensures that the conjuncts $P_1, \ldots, P_m$ in the antecedent match the capabilities provided by the `consume` clause, and for the consequent, binds existential variables $x_1 : s_1, \ldots, x_k : s_k$ to the `given` clause and assigns the resulting conjuncts $Q_1, \ldots, Q_n$ to the capabilities named in the `produce` clause (and checks conformance if assertions are already given).

There are two complementary styles of reasoning blocks. *Declarative* blocks state assertions in the resulting capabilities upfront and generate a proof obligation, which can be fetched using `get_goal()`. The proof code inside usually performs *backward reasoning* (also known as *goal-directed reasoning*), i.e., recursively decomposing a proof obligation into smaller ones until all sub-goals are trivially solvable. Alternatively, *operational* blocks derive output assertions by performing explicit logical deductions on input capabilities, without stating them upfront. The proof code inside usually performs *forward reasoning*, i.e., deriving a theorem from existing assumptions and theorems by sound proof rules. We describe our proof interface for backward and forward reasoning in Section 4.

*Local Variables.* Each local variable and function parameter declaration is associated with a capability representing its ownership via an `alloc` clause. If the initializer $e$ has type `i32[`v`]`, capability $c$ is initialized to `` `i32_at &x v` ``. The type checker ensures all local variable capabilities are deallocated when the variable's scope ends (block or function scope), preventing unsafe dereferencing of dangling pointers to deallocated stack frames.

```
/*@ alloc(c) @*/
int32_t x = e;
```

Fig. 8. Variable declaration.

*Jump Statements.* A `return` statement returns both a computational expression $e$ and the capabilities specified in the enclosing function's `ensure` clause. The type checker retrieves these capabilities from the current typing context and verifies that all remaining capabilities can be discarded (e.g., local variable capabilities or pure facts). Similar to the optional `instantiate` clause in function calls, a `witness` clause can explicitly provide witnesses for existential variables in the function specification (automatically inferred by the type checker if omitted). Other jump statements (**`break`** and **`continue`**) are handled similarly, with different expected available capabilities.

```
/*@ witness(`u1`, ...) @*/
return e;
```

Fig. 9. Return.

*Conditional Statements.* An **`if`** statement introduces flow-sensitive facts via a `pathcond` clause, recording that the condition holds (i.e., is non-zero) or does not hold (i.e., is zero) along each branch. At the control-flow merge point, a `join` clause explicitly specifies which capabilities from both branches are merged. The type checker ensures that at the end of both branches: (1) merged capabilities record the same assertion, (2) these assertions are well-formed in the outer scope, i.e., no escaping logical variables or local variable addresses, and (3) remaining capabilities in either branch are discardable. An additional `exgiven` clause may be used to abstract branch-specific details at the merge point as existential quantifiers (this typically requires annotating the capabilities in the `join` clause with explicit assertions to perform the correct abstraction), and immediately introduce them into scope until the end of the enclosing block (the same functionality as the `given` clause in function calls and reasoning blocks).

```
if (e /*@ pathcond(c) @*/)
  { stmt_t ... }
else
  { stmt_f ... }
/*@ exgiven(`x1 : s1`, ...) @*/
/*@ join(c1:`P1`, ...) @*/
```

Fig. 10. Two-branch conditional.

**Identifiers** $x, f$ **Operator Symbols** $op$ **Literals** $n \in [-2^{31}, 2^{31}-1]$

| | | | |
|---|---|---|---|
| **Expressions** | $e$ | ::= | $n \mid x \mid \&e \mid *e \mid op(\vec{e})$ |
| **Statements** | $s$ | ::= | skip $\mid s_1; s_2 \mid$ return $e \mid e_l = f(\vec{e})$ |
| | | $\mid$ | if $e$ then $s_1$ else $s_2 \mid$ while $e$ do $s$ |
| **Declarations** | $d$ | ::= | i32 $x = e$ |
| **Functions** | $fun$ | ::= | i32 $f\left(\overrightarrow{\text{i32 } x}\right)\ \vec{d}\ s$ |
| **Programs** | $p$ | ::= | $\overrightarrow{fun}$ |

Fig. 12. Abstract syntax of the C subset used by C*.

**Capability Variables** $l$ **Logic Variables** $\alpha, \beta$

| | | | |
|---|---|---|---|
| **Logic Sorts** | $\mathcal{S}$ | ::= | int $\mid$ bool $\mid$ hprop $\mid \ldots$ |
| **Logic Terms** | $w, P, Q$ | ::= | $\alpha \mid 42 \mid$ true $\mid$ emp $\mid$ fact$(w :$ bool$) \mid P \star Q \mid$ i32_at$(w_1, w_2) \mid \ldots$ |
| **Capabilities** | $\kappa$ | ::= | $l : P$ |
| **Expressions** | $\hat{e}$ | ::= | $n \mid x \mid \&\hat{e} \mid *\hat{e} \mid$ check$(\vec{\kappa})\ op(\vec{\hat{e}})$ |
| **Statements** | $\hat{s}$ | ::= | skip $\mid \hat{s}_1; \hat{s}_2 \mid$ witness$(\vec{w})$ return $\hat{e}$ |
| | | $\mid$ | instantiate$(\vec{w})$ consume$(\vec{\kappa_r})$ given$(\vec{\alpha})$ produce$(\vec{\kappa_e})\ \hat{e}_l = f(\vec{\hat{e}})$ |
| | | $\mid$ | if $\hat{e}$ pathcond$(l)$ then $\hat{s}_1$ else $\hat{s}_2$ exgiven$(\vec{\alpha})$ join$(\vec{\kappa})$ |
| | | $\mid$ | exgiven$(\vec{\alpha})$ invariant$(\vec{\kappa})$ while $\hat{e}$ do $\hat{s}$ |
| | | $\mid$ | consume$(\vec{\kappa_r})$ given$(\vec{\alpha})$ produce$(\vec{\kappa_e})$ $\{\!|proof|\!\}$ |
| **Declarations** | $\hat{d}$ | ::= | alloc$(\kappa)$ i32$[w]$ $x = \hat{e}$ |
| **Functions** | $\hat{fun}$ | ::= | forall$(\vec{\alpha})$ require$(\vec{\kappa_r})$ exist$(\vec{\beta})$ ensure$(\vec{\kappa_e})$ i32$[w_r]$ $f\left(\overrightarrow{\text{alloc}(\kappa)\ \text{i32}[w]\ x}\right)\ \vec{\hat{d}}\ \hat{s}$ |
| **Programs** | $\hat{p}$ | ::= | $\overrightarrow{\hat{fun}}$ |

Fig. 13. Abstract Syntax of C*.

*Iteration Statements.* A **while** statement requires an `invariant` clause specifying the loop invariant (essentially, a `join` clause specifically at the loop head); it may also use `exgiven` to introduce existential variables and bring them into scope. Similar to **if**, the loop condition uses a `pathcond` clause recording that the condition holds within the loop body and does not hold on exit. The type checker ensures the loop invariant is maintained when executing **continue** or falling through to the next iteration. The general form allowing **break** inside the loop body also requires a `join` clause at the loop exit point and is deferred to the supplementary material.

```
/*@ exgiven(`x1 : s1`, ...) @*/
/*@ invariant(c1:`P1`, ...) @*/
while (e /*@ pathcond(c) @*/)
{ stmt ... }
```

Fig. 11. While loop (with no break).

### 3.3 Formalization

This section formalizes the typing rules of C* and proves its meta-properties. For simplicity of presentation, we assume all functions have return values and all capabilities are assertion-annotated; we omit the treatment of blocks and **break** and **continue** statements.

$$\frac{\begin{array}{c}\vec{\kappa} = \overrightarrow{l : P} \qquad \Lambda_{\text{in}}; \Sigma_{\text{in}} \vdash_{\text{rexp}} \hat{e} : \text{i32}[w] \\ \Lambda_{\text{in}}; \Sigma_{\text{in}}, l_p : \text{fact}(w \neq 0) \mid K_{\text{ret}} \vdash_{\text{stmt}} \hat{s}_t : \Lambda_{\text{out}_t}; \Sigma_{\text{out}_t} \\ \Lambda_{\text{in}}; \Sigma_{\text{in}}, l_p : \text{fact}(w = 0) \mid K_{\text{ret}} \vdash_{\text{stmt}} \hat{s}_f : \Lambda_{\text{out}_f}; \Sigma_{\text{out}_f} \\ \exists \sigma_t.\ dom(\sigma_t) = \{\vec{\alpha}\} \wedge \Sigma_{\text{out}_t} = \overrightarrow{l : \sigma_t(P)}, \Sigma'_t \wedge \mathit{discardable}(\Sigma'_t) \\ \exists \sigma_f.\ dom(\sigma_f) = \{\vec{\alpha}\} \wedge \Sigma_{\text{out}_f} = \overrightarrow{l : \sigma_f(P)}, \Sigma'_f \wedge \mathit{discardable}(\Sigma'_f)\end{array}}{\Lambda_{\text{in}}; \Sigma_{\text{in}} \mid K_{\text{ret}} \vdash_{\text{stmt}} \text{if } \hat{e} \text{ pathcond}(l) \text{ then } \hat{s}_t \text{ else } \hat{s}_f \text{ exgiven}(\vec{\alpha}) \text{ join}(\vec{\kappa}) : \Lambda_{\text{in}}, \vec{\alpha}; \vec{\kappa}}$$

Fig. 14. Typing rule of if statements. Well-formedness checks are omitted for brevity.

*3.3.1 Syntax.* We present the abstract syntax of C* (Figure 13) and that of the corresponding C subset (Figure 12). We use the $\vec{\ }$ notation to denote a finite vector of items, e.g., $\vec{\alpha}$. For a C* program $\hat{p}$ (and similarly for a statement $\hat{s}$ or an expression $\hat{e}$), we write $p$ for its erased C counterpart obtained by removing all capability-passing clauses (e.g., check, pathcond), type refinements (e.g., $[w]$ in $\text{i32}[w]$), and proof blocks (i.e., $\{\!|\mathit{proof}|\!\}$; gets translated to a skip). We assume all variables are declared at the beginning of a function and all capabilities are annotated with assertions. We assume each logical variable and logic term inherently has a sort, and $P, Q$ are reserved for separation-logic assertions of sort hprop; the sort of a term can be explicitly annotated for clarity, e.g., $P : \text{hprop}$.

*3.3.2 Typing Judgments.* The typing judgment of C* statements has the form

$$\Lambda_{\text{in}}; \Sigma_{\text{in}} \mid K_{\text{ret}} \vdash_{\text{stmt}} \hat{s} : \Lambda_{\text{out}}; \Sigma_{\text{out}}$$

under an implicit context of user-defined and primitive function specifications. Logical variable contexts $\Lambda$, capability contexts $\Sigma$, and continuation specifications $K$ are defined as follows:

$$\Lambda ::= \cdot \mid \Lambda, \alpha \qquad \Sigma ::= \cdot \mid \Sigma, \kappa \qquad K ::= \text{exist}(\vec{\alpha})\ \text{return}(\text{i32}[w])\ \text{ensure}(\vec{\kappa_r}).$$

We treat typing contexts $\Lambda$ and $\Sigma$ as unordered and assume binders are distinct (by suitable $\alpha$-renaming). We denote context concatenation by juxtaposition, e.g., $\Lambda_1, \Lambda_2$.

In this typing judgment, $\hat{s}$, $\Lambda_{\text{in}}$, $\Sigma_{\text{in}}$, $K_{\text{ret}}$ are treated as inputs, whereas $\Lambda_{\text{out}}$ and $\Sigma_{\text{out}}$ are treated as output. $\Lambda_{\text{in}}$ and $\Sigma_{\text{in}}$ track the logic variables in scope and capabilities available at the beginning of the statement, whereas $\Lambda_{\text{out}}$ and $\Sigma_{\text{out}}$ are the updated contexts after the statement executes under normal exit (e.g., without calling return). The return continuation specification $K_{\text{ret}}$ corresponds to the output specification of the enclosing function. We also have typing judgments for l-value expressions $\hat{e_l}$ of the form $\Lambda_{\text{in}}; \Sigma_{\text{in}} \vdash_{\text{lexp}} \hat{e_l} : \text{i32}[\mathit{addr}]$, where the refinement term $\mathit{addr}$ represents an address to be accessed; and for r-value expressions $\hat{e}$ of the form $\Lambda_{\text{in}}; \Sigma_{\text{in}} \vdash_{\text{rexp}} \hat{e} : \text{i32}[\mathit{val}]$, where $\mathit{val}$ represents a value the expression evaluates to.

*3.3.3 Typing Rules.* The typing rules of C* are formulated in a syntax-guided manner, making them directly executable as a type-checking algorithm. For example, the rule for if statements is shown in Figure 14. This rule type-checks a conditional by ensuring that both branches unify with a common join specification. First, the rule type-checks the condition expression $\hat{e}$ to obtain its value $w$. It then type-checks both branches separately, each with an augmented capability context that includes the appropriate path condition. After type-checking both branches, the rule verifies that their output capability contexts $\Sigma_{\text{out}_t}$ and $\Sigma_{\text{out}_f}$ can be unified with the join specification $\vec{\kappa} = \overrightarrow{l : P}$. Specifically, the rule requires that there exist substitutions $\sigma_t$ and $\sigma_f$ mapping the existential variables $\vec{\alpha}$ (declared in $\text{exgiven}(\vec{\alpha})$) to concrete terms, such that $\Sigma_{\text{out}_t}$ contains $\overrightarrow{l : \sigma_t(P)}$ (i.e., an instantiation of the join capabilities) plus some additional capabilities $\Sigma'_t$ that are discardable (e.g.,

$$\textsc{SepComm}\ \frac{}{P_1 \star P_2 = P_2 \star P_1} \qquad \textsc{SepAssoc}\ \frac{}{(P_1 \star P_2) \star P_3 = P_1 \star (P_2 \star P_3)} \qquad \textsc{SepExist}\ \frac{}{(\exists \alpha.\ P_1) \star P_2 = \exists \alpha.\ (P_1 \star P_2)} \qquad \textsc{ExistForall}\ \frac{\forall \alpha.\ (P \vdash Q)}{(\exists \alpha.\ P) \vdash (\exists \alpha.\ Q)}$$

Fig. 15. Separation-logic structural rules used in the proof construction (an excerpt).

$$\frac{\begin{array}{l}[\![\vec{\kappa}_r]\!] \vdash \exists \vec{\alpha}.[\![\vec{\kappa}_e]\!] \quad \text{(established by running } \textit{proof}\text{)} \\ \Sigma_{\text{in}} = \vec{\kappa}_r, \Sigma_f \qquad \Lambda_{\text{out}} = \Lambda_{\text{in}}, \vec{\alpha} \qquad \Sigma_{\text{out}} = \vec{\kappa}_e, \Sigma_f\end{array}}{\Lambda_{\text{in}}; \Sigma_{\text{in}} \vdash_s \text{consume}(\vec{\kappa}_r)\ \text{given}(\vec{\alpha})\ \text{produce}(\vec{\kappa}_e)\ \{\!|\textit{proof}|\!\} : \Lambda_{\text{out}}; \Sigma_{\text{out}}}$$

Fig. 16. Typing rule of reasoning blocks. The exact form of proof code is made abstract.

$$\textsc{Conseq}\ \frac{\textsc{Frame}\ \dfrac{\textsc{Conseq-Skip}\ \dfrac{[\![\vec{\kappa}_r]\!] \vdash \exists \vec{\alpha}.[\![\vec{\kappa}_e]\!]}{\{[\![\vec{\kappa}_r]\!]\}\ \text{skip}\ \{\exists \vec{\alpha}.[\![\vec{\kappa}_e]\!]\}}}{\{[\![\vec{\kappa}_r]\!] \star [\![\Sigma_f]\!]\}\ \text{skip}\ \{(\exists \vec{\alpha}.[\![\vec{\kappa}_e]\!]) \star [\![\Sigma_f]\!]\}} \qquad (\exists \vec{\alpha}.[\![\vec{\kappa}_e]\!]) \star [\![\Sigma_f]\!] \vdash \exists \vec{\alpha}.[\![\vec{\kappa}_e]\!] \star [\![\Sigma_f]\!]}{\{[\![\vec{\kappa}_r]\!] \star [\![\Sigma_f]\!]\}\ \text{skip}\ \{\exists \vec{\alpha}.\ ([\![\vec{\kappa}_e]\!] \star [\![\Sigma_f]\!])\}}$$

Fig. 17. The corresponding separation-logic derivation for a reasoning block.

pure facts); similarly for $\Sigma_{\text{out}_f}$. If all checks pass, the rule concludes the if statement is well-typed, with updated logical variables $\Lambda_{\text{in}}, \vec{\alpha}$ (with newly introduced existentials) and join capabilities $\vec{\kappa}$.[6]

*3.3.4 Meta-Properties.* The typing rules of C* are closely related to separation-logic proof rules of the underlying C program. Our language design (with capability-passing and reasoning blocks) and typing rules ensure that a well-typed C* statement (and expression, program, etc.) corresponds to a valid separation-logic derivation of the underlying erased C statement (and expression, program, etc.). Given $\Lambda = \{\alpha_1, \ldots, \alpha_k\}$ and $\Sigma = \{l_1 : P_1, \ldots, l_m : P_m\}$, we define the following notations:

$$\forall \Lambda.P \stackrel{\text{def}}{=} \forall \alpha_1, \ldots, \alpha_k.\ P \qquad \exists \Lambda.P \stackrel{\text{def}}{=} \exists \alpha_1, \ldots, \alpha_k.\ P \qquad [\![\Sigma]\!] \stackrel{\text{def}}{=} \bigstar_{i=1}^{m} P_i\ .$$

We establish the following soundness result, stated for simplicity without concerning jump statements (hence ignoring the $K_{\text{ret}}$ continuation specification in the typing judgment).

Theorem 3.1 (Soundness). *For any well-typed C* statement $\hat{s}$, i.e., $\Lambda_{\text{in}}; \Sigma_{\text{in}} \vdash_{\text{stmt}} \hat{s} : \Lambda_{\text{out}}; \Sigma_{\text{out}}$, there exists a logical variable context $\Lambda$ such that: (1) $\Lambda_{\text{out}} = \Lambda_{\text{in}}, \Lambda$, and (2) the erased C statement $s$ satisfies the following separation-logic triple in a canonical form [9]:*

$$\forall \Lambda_{\text{in}}.\ \{[\![\Sigma_{\text{in}}]\!]\}\ s\ \{\exists \Lambda.[\![\Sigma_{\text{out}}]\!]\}\ .$$

Proof. The proof goes by straightforward induction on the typing derivation. It constructs a separation-logic derivation template for each case, relying on structural rules of separation logic (some are shown in Figure 15) to align the proof state to typing-context manipulations. □

As an example case, consider the typing rule for reasoning blocks, shown in Figure 16. It corresponds to the separation-logic derivation template for its erased C counterpart skip, shown in

[6] In our implementation, the join assertions $\vec{P}$ need not be provided upfront (when there is no need for abstraction) as they can be inferred from the output capability contexts of the two branches.

Term-Specific Utilities

```
/* constructor */
term conj = mk_binop(`**`, `P`, `Q`);

/* destructor */
term left = left_of_sep(conj);   // P
term right = right_of_sep(conj); // Q

/* discriminator */
if (is_sep(conj)) { ... }

/* equality checker */
if (equals_term(left, right)) { ... }
```

Theorem-Specific Utilities

```
/* assume th proves i < 2 |- i < 3 */
/* get the conclusion */
term concl = conclusion(th); // i < 3

/* get the n-th hypothesis */
term hyp = nth_hypth(th, 0); // i < 2
```

Definitional Mechanisms

```
/* inductive type */
indtype int_list =
  define_type("int_list =
               nil | cons integer int_list");

/* recursive function */
thm nth =
  define(`nth(cons(h,t)),0) = h &&
         nth(cons(h,t)),SUC(n)) = nth(t,n)`);
```

Reasoning Block Interface

```
/* get the goal of a declarative block */
term goal = get_goal();

/* give the proven entailment of this block */
by_using(axiom(`P |-- Q ** R`));

/* provide the produced capabilities */
returns(`Q`, `R`);
```

Inference Rules

```
axiom(`0 = 1`)                          // |- 0 = 1
assume(`0 = 1`)                         // 0 = 1 |- 0 = 1
disch(assume(`x > 0`), `x > 0`)         // |- x > 0 => x > 0
undisch(axiom(`p => q`))                // p |- q
mp(axiom(`p => q`), axiom(`p`))         // |- q
conjunct(axiom(`p`), axiom(`q`))        // |- p /\ q
conjunct1(axiom(`p /\ q`))              // |- p
disj_cases(axiom(`p \/ q`),
           undisch(axiom(`p => r`)),
           undisch(axiom(`q => r`)))    // |- r
refl(`x`)                               // |- x = x
trans(axiom(`x = y`), axiom(`y = z`))   // |- x = z
symm(axiom(`x = y`))                    // |- y = x
spec(`x`, axiom(`∀a. a = a`))           // |- x = x
arith_rule(`i < 2 => i < 3`)            // |- i < 2 => i < 3
hentail_refl(`i32_at p 0`)              // |- i32_at p 0 |-- i32_at p 0
eq2ent(axiom(`P = Q`))                  // |- P |-- Q
```

Backward Proof Tactics

```
GN Init_goal(term goal);  /* start backward proof */
void Proof_goal(GN g, thm th);  /* prove goal A ?- t with thm A |- t */
thm Solve(GN g); /* finish backward proof if subgoals in g are solved */
GN CONJ_TAC(GN g);  // [A ?- t1 /\ t2] => [A ?- t1; A ?- t2]
GN DISJ1_TAC(GN g); // [A ?- t1 \/ t2] => [A ?- t1]
GN MP_TAC(GN g, thm th);
                    // [A ?- t] => [A ?- s] where th: A |- s => t
GN DISCH_TAC(GN g); // [A ?- u => v] => [A,u ?- v]
GN UNDISCH_TAC(GN g, int n);
                    // [A,A_n ?- v] => [A ?- A_n => v]
GN CHOOSE_TAC(GN g, thm th);
                    // [A ?- u] => [A,t ?- u] where th: A |- ∃x.t
GN EXISTS_TAC(GN g, term u);
                    // [A ?- ∃x.t] => [A ?- t[u/x]]
GN GEN_TACS(GN g);  // [A ?- ∀x1 … xn. t] => [A ?- t]
GN REWRITE_TAC(GN g, thm th);
                    // [A ?- t] => [A ?- t[v/u]] where th: A |- u = v
GN ARITH_TAC(GN g); /* solve g by arith_rule */
GN FACTS_INTRO_TAC(GN g);
                    // [A ?- P ** fact(t) |-- Q] => [A,t ?- P |-- Q]
GN AUTO_FRAME_TAC(GN g);
                    // [A ?- h1 ** h2 |-- h1 ** h3] => [A ?- h2 |-- h3]
```

Note: `x` is syntax sugar for *parse_term*("x")

Fig. 18. Representative C proof functions in C*.

Figure 17 (outermost $\forall\Lambda_{\text{in}}$ quantifiers are implicit; associativity and commutativity of separating conjunction are implicitly applied). The entailments at leaf nodes are discharged by the entailment established by proof code and the structural rules shown in Figure 15.

## 4 Sound and C-Programmable Proof Interface

This section presents our proof interface in C. We describe our LCF-style interface in Section 4.1, showing how users of C* can construct theorems as abstract values and extend proof support by C programming. For ensuring soundness of proof programming in C, we introduce a variant of the LCF architecture in Section 4.2.

### 4.1 Programmable Proof Interface in C

*LCF-Style Theorem Proving.* For programmable proof support, we follow the long-standing LCF approach to theorem proving. Pioneered by Robin Milner and colleagues in the early work on the Edinburgh LCF theorem prover [21, 22], this approach and its descendants are still widely used today [25, 37, 45]. They use a general-purpose programming language as the meta-language to manipulate logical entities, such as terms, types, and theorems. The axioms of the logic are then represented as theorem constants, and the inference rules are implemented as interfaces that return theorems. This approach is programmable in that arbitrary proof functions (e.g., derived rules or search strategies) and proof styles (e.g., goal-directed) can be developed on top of this interface.

```
term list_mk_hsep(term* tms) {
  int len = tms_len(tms);
  if (len <= 0) return `emp`;
  term result = tms[len - 1];
  for (int i = len - 2; i >= 0; i--)
    result = mk_binop(`**`,tms[i],result);
  return result; }
```

(a)

```
thm sll_cons_destr_proof() {
  GN root = Init_goal(...);
  GN g = GEN_TACS(root);
  g = FACTS_INTRO_TAC(g);
  thm lem = match_mp(sll_not_zero, Get_asmp(g, 0));
  Proof_goal(g, spec(`l:int list`, lem));
  return Solve(root); }
```

(b)

Fig. 19. Example term manipulation and proof code in C*.

In C*, we provide an LCF-style proof interface in the C programming language, offering a variety of functions to manipulate HOL terms (of type `term`) and theorems (of type `thm`). Figure 18 showcases representative proof functions in C*, some of which are primitive in HOL (e.g., `assume` and `refl`) while others are extended in C (e.g., the backward proof tactics; see Section 5).

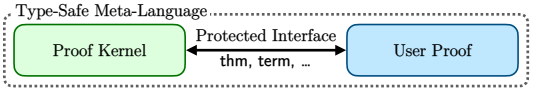
*Term Utilities.* The terms in C* are simply-typed lambda terms. Users can construct a term in conventional mathematical notation by calling `parse_term`, which parses a string representation into a `term` object (we use syntactic sugar `` `...` `` for this common operation). To facilitate programmatic manipulation of terms, term constructor functions allow building complex terms from simpler ones; for example, `mk_binop` constructs a binary application from an operator and two operands. These constructor functions check that the resulting terms are well-formed; for example, `mk_binop` verifies that the operator and operands have matching types. Terms can be inspected and dissected using discriminator functions (e.g., `is_sep`) and destructor functions (e.g., `left_of_sep`), and can be compared for ($\alpha$-)equality using `equals_term`.

As an example of extended term manipulation utilities, Figure 19(a) shows a function `list_mk_hsep` that constructs the iterated separating conjunction of an array of terms in a right-associative manner (returning `` `emp` `` if the array is empty).

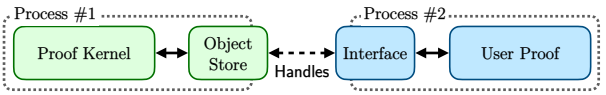
*Theorem Utilities and Inference Rules.* A theorem is a sequent consisting of a list of hypotheses and a conclusion (written as `A1,...,An |- A` in Figure 18), all of which must be propositions (i.e., terms of the *bool* sort). We provide destructors for theorems, which allow inspection of a theorem's conclusion and individual hypotheses as terms. Theorems are constructed using inference rules, such as `assume` (which deduces a proposition from itself as a hypothesis), `mp` (modus ponens), and `disch` (which discharges a hypothesis as an antecedent for the conclusion). For convenience, the interface also includes higher-level proof automation functions, such as `arith_rule`, which decides linear arithmetic propositions.

*Definitional Mechanisms.* Definitional mechanisms in the proof interface allow users to extend the logic with new types and constants. New inductive data types can be defined using `define_type`, which takes the clauses of the type definition as a string and returns an `indtype` (a struct containing the induction and recursion theorems for the new inductive type). The `define` function allows users to define new term constants by providing the defining equations of a total function.

*Backward Proof Tactics.* The native style of proof in the LCF approach is to invoke inference rules (and forward proof functions derived from them) such that new theorems are derived from existing ones, known as the *forward proof style*. It is sometimes more intuitive to work backwards when the goal is known upfront, recursively decomposing it into (hopefully simpler) subgoals that entail the validity of the original goal. This style of proof is known as the *backward proof style*.




(a) The traditional LCF architecture: protect proof interface by type abstraction.




(b) Our variant of the LCF architecture: protect proof interface by process isolation.

Fig. 20. Ensuring soundness of LCF-style proof programming.

We support this goal-directed proof style as a library extension, implementing tactics as functions that transform *goal trees*, which contain *goal nodes* representing subgoals. To distinguish from a proven theorem, we write a goal as a sequent `A1,...,An ?- A` in Figure 18, where the question mark indicates that the goal is yet to be proven. Users can start a backward proof using `Init_goal`, which initializes a goal tree with a single node representing the to-be-proven goal `?- A` with no hypotheses and the given term as the conclusion. Users can then apply tactics to the goal tree, extending it with new *goal nodes* that represent the subgoals of the original goal. For example, a conjunctive goal can be decomposed into two subgoals by `CONJ_TAC`. A subgoal can be solved directly by `Proof_goal` if a matching theorem is available, or by automation tactics like `ARITH_TAC`. Finally, after proving all subgoals, calling `Solve` on the root node of the goal tree automatically combines the proven theorems of the subgoals into the theorem corresponding to the original goal.

Figure 19(b) shows an example backward proof for the destruction theorem of non-empty lists `sll_cons_destr` (its statement is given in paragraph *Operational Reasoning Steps* in Section 2). It uses general tactics such as `GEN_TACS` and separation-logic-specific tactics such as `FACTS_INTRO_TAC` (we write `P|--Q` for a separation-logic entailment in Figure 18).

*Reasoning Block Interface.* Our C proof interface is designed to be used within *reasoning blocks* in C* programs. When type-checking a reasoning block, the consumed capabilities are automatically bound to term variables with the same names, providing the proof code with access to the input capabilities as logical terms. The proof code can then manipulate these terms using the proof functions described above to construct the desired entailment theorem. To interact with the type checker, the interface provides several specialized functions: `get_goal` retrieves the current goal to be proven (in a declarative reasoning block), and `by_using` submits the proven theorem back to the type checker to certify the reasoning step. By default, the produced capabilities are bound to the conjuncts of the consequent of the proven entailment theorem. We also provide a way to explicitly specify the resulting capabilities using the `returns` function (e.g., when we want to put two separating conjuncts into the same capability).

### 4.2 Ensuring Soundness

To ensure soundness, all proof code must ultimately reduce to calls to primitive inference rules (implemented by a small *proof kernel*), a property known as *full expansiveness* [20]. The original LCF architecture achieves this through *type abstraction* [21]: logical objects are implemented as abstract data types in the meta-language, ensuring all theorem values are constructed only through the proof interface (illustrated in Figure 20(a)).

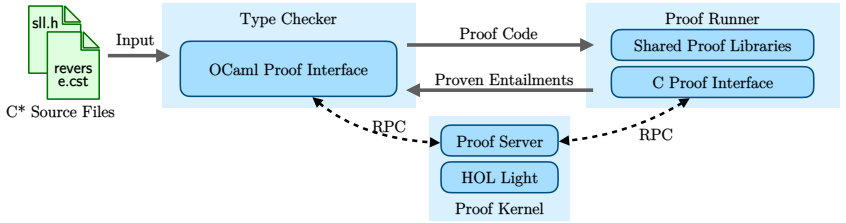



Fig. 21. Implementation architecture of the C* prototype.

However, C's weak type system does not provide such type abstraction as it allows arbitrary mutation and type casting. This allows user code to directly manipulate memory representations and forge invalid theorems. For example, if theorems are represented as a struct containing a conclusion and an array of hypotheses:

```
typedef struct thm_node { term concl; term* hyps } *thm;
```

A malicious user could directly mutate a theorem's conclusion, forging an invalid theorem:

```
thm taut = assume(`true`); // taut proves true |- true
taut->concl = `false`;     // taut proves true |- false, unsound!
```

Similarly, if goals and theorems share the same memory representation, a user could inadvertently pass an unproven goal where a proven theorem is expected, leading to unsound reasoning.

*Our Solution.* We address this challenge through a *process-isolated architecture* (illustrated in Figure 20(b)) that enforces full expansiveness without relying on language-level type abstraction. The key insight is to isolate user proof code from the proof kernel using separate processes:[7]

- The *proof kernel* runs in one process, maintaining the actual representations of logical objects (theorems, terms) and implementing the primitive proof interfaces.
- User-provided proof code runs in another separate process and manipulates logical objects only through *handles* (i.e., opaque identifiers) generated by the kernel.
- At the interprocess communication boundary, the kernel validates all operations by looking up handles in its object store, ensuring only valid logical objects are referenced.

Operating system memory isolation prevents user code from directly accessing or forging logical objects in the kernel's process. Consequently, all logical objects manipulated by user code must be constructed through the kernel's proof functions, satisfying the property of full expansiveness.

## 5 Implementation and Evaluation

*Implementation Notes.* We implement a prototype language and verifier for C*, whose architecture is illustrated in Figure 21. The system consists of three main components that interact during verification after the frontend compiler parses the input source files:

- The *type checker* implements the typing rules of C* (described in Section 3.3) and checks the input source files. When encountering a reasoning block, it delegates the C proof code to the proof runner for execution and certifies the resulting entailment theorem.
- The *proof runner* executes the C proof code within reasoning blocks. It receives the C proof code from the type checker, interacts with the proof kernel to resolve calls to the proof interface, and produces an entailment theorem that is fed back to the type checker.

[7] As seen shortly in Section 5, user proof and proof kernel need not be written in the same programming language.

Table 1. Benchmark statistics. The columns stand for: **#Fun**: number of functions; **#Impl**: lines of implementation code; **#Proof**: lines of proof code (not counting libraries); **#OP**: number of operational reasoning blocks; **#DC**: number of declarative reasoning blocks; **Locality%**: the arithmetic mean of the ratios of the number of capabilities passed in each reasoning block to the total number of capabilities available in typing context.

| Class | Name | #Fun | #Impl | #Proof | #OP | #DC | Locality% |
|---|---|---|---|---|---|---|---|
| LiveVerif | min | 2 | 18 | 36 | 2 | 2 | 32.5 |
| | fibonacci | 1 | 20 | 104 | 3 | 2 | 56.2 |
| | swap | 1 | 10 | 0 | 0 | 0 | - |
| | sort3 | 1 | 36 | 25 | 6 | 3 | 20.0 |
| | sort3_separate_args | 1 | 36 | 16 | 4 | 3 | 14.5 |
| | memset | 1 | 10 | 48 | 6 | 9 | 17.7 |
| | linked_list | 2 | 24 | 38 | 9 | 9 | 27.8 |
| | onesize_malloc | 3 | 37 | 57 | 15 | 11 | 15.7 |
| | tree_set | 4 | 97 | 100 | 34 | 17 | 18.8 |
| | nt_uint8_string | 1 | 17 | 83 | 22 | 26 | 11.5 |
| | critbit | 14 | 245 | 300 | 137 | 46 | 18.1 |
| Extended | clear | 1 | 15 | 31 | 4 | 2 | 52.3 |
| | abs | 1 | 13 | 4 | 4 | 2 | 41.7 |
| | factorial | 1 | 14 | 16 | 0 | 3 | 46.7 |

- The *proof kernel* provides core reasoning services in the LCF style, including term-manipulation utilities and basic proof functions (described in Section 4.1).

We reuse HOL Light [25], a theorem prover for higher-order logic implemented in OCaml, as our proof kernel. The type checker is implemented in OCaml, while the proof runner executes in C; both components communicate with the kernel through remote procedure calls (RPCs).

*Evaluation.* We evaluate C*'s effectiveness in supporting local, operational, and extensible reasoning using the benchmark from the Live Verification framework [23].[8] Our benchmark covers an interesting set of C programs, including: simple algorithms (e.g., `fibonacci`, `sort3`, `factorial`), low-level memory manipulations (e.g., `memset`, `nt_uint8_string`, `onesize_malloc`), and linked data structures (e.g., `linked_list`, `tree_set`, `critbit`). We verify the functional correctness of all benchmark programs entirely within C*, with verification results shown in Table 1.

*Locality of Reasoning.* The **Locality%** metric in Table 1 is calculated by taking the average of the ratios of the number of capabilities consumed by each reasoning block to the total number of capabilities available at that point in the typing context. Lower values of **Locality%** indicate better locality of the reasoning steps performed. The results demonstrate that C* allows users to effectively perform local reasoning steps, where the reasoning dependencies on the relevant pieces of information in the global proof state are made explicit as capability inputs.

*Prevalence of Operational Reasoning.* Table 1 shows the numbers of operational (**#OP**) and declarative (**#DC**) reasoning blocks for each program, which confirms that operational reasoning is frequently employed in verification. It is worth noting that these two styles have equivalent expressiveness *in theory* (i.e., any declarative block can be converted to an operational one as long as they prove the same entailment); nevertheless, all operational blocks in our verified programs

[8] We made some modifications to the benchmark programs (details are provided in the supplementary material): (1) We modify `tree_set` and `critbit` to a recursive style; (2) We omit several functions in `critbit`; (3) We remove the two variants of `swap`; (4) We use signed integers instead of unsigned; arithmetic overflow checks are ignored.

Table 2. Implemented proof libraries in C*. The columns stand for: **Lib Type**: type of the library (shared or domain-specific); **Lib Name**: name of the library; **#Func**: number of functions in the library; **File Size (KB)**: size of the library file in kilobytes; **Description**: brief description of the library's purpose.

| Lib Type | Lib Name | #Func | File Size (KB) | Description |
|---|---|---|---|---|
| Shared | proof_user | 91 | 21.8 | Term and theorem utilities and rewriting |
| | proof_conv | 61 | 31.4 | Separation logic reasoning |
| | proof_tactic | 77 | 61.8 | Backward reasoning support |
| | cstar_opfun_lib | 28 | 9.78 | Basic operational proof functions |
| | cstar_dcfun_lib | 4 | 1.83 | Basic declarative proof functions |
| | cstar_pure_solver | 16 | 8.18 | Automated solvers for pure facts |
| Domain-Specific | sll_lib | 4 | 0.63 | Singly linked lists |
| | array_at_lib | 14 | 4.40 | Arrays |
| | bst_lib | 10 | 5.86 | Binary search trees |

call the operational proof functions described later, meaning the operational style is *naturally applicable* in these reasoning steps as opposed to giving assertions upfront.

*C-Programmable Proof Support.* We implemented C proof libraries atop the C* proof interface and used them in reasoning blocks during the benchmark verification process. Table 2 summarizes the proof libraries we implemented, including their sizes and descriptions. We group these libraries into two categories: *shared* and *domain-specific*. The shared libraries provide general-purpose proof abstractions (C functions/macros) for common reasoning patterns, covering: term and theorem utilities; rewriting strategies (e.g., `rewrite_list`, `once_rewrite`); separation logic proof rules (e.g., `hent_refl`); backward reasoning tactics (e.g., `CONJ_TAC`); basic operational and declarative proof functions (e.g., `OP_REWRITE`, `DC_REWRITE`); and automated pure fact solving (e.g., `int_arith_solver`). The domain-specific libraries contain specialized predicates, lemmas, and proof functions tailored to particular domains and data structures.

## 6 Related Work

There are many verifiers, proof frameworks, and infrastructures targeting verification of heap-manipulating imperative programs [5, 6, 8, 9, 11–14, 18, 19, 23, 28, 31, 34, 39–41, 44, 49, 51, 52]. We compare C* with recent separation-logic-based verifiers for C [28, 41, 44] and Live Verification [23].

*Live Verification [23].* Live Verification [23] is a separation-logic proof framework embedded in the Rocq prover. Leveraging Rocq's support for existential meta-variables and custom tactic notations, it realizes a form of real-time verification: users can inspect the current proof state in the goal panel, use suitable customized tactics to synthesize the next line of implementation code along with its correctness proof derived.

C* adopts a similar C subset and reuses the Live Verification benchmark. C* does not support partial programs. C* takes a programming-integrated approach: it reifies the local proof state as capabilities and reasoning dependencies as explicit capability passing, visible in source programs without running proofs interactively. Additionally, whereas Live Verification uses Ltac/Ltac2 for composing proof scripts and automation procedures, C* offers a sound programmable proof interface in C for proof programming.

*CN [41].* CN is a separation-logic-based verifier for C with an SMT backend, targeting predictable automation for conventional systems software. It elaborates a large fragment of ISO C into a first-order functional language (Core) using the Cerberus semantics [35]. It defines a type system for Core,

which combines liquid-style refinement types [43] with linear resources to represent separation-logic assertions. By restricting the use of logical quantifiers in specifications, CN ensures decidable, quantifier-free VCs. It adds specialized automation for iterated separating conjunction [36]. When automation falls short, users can state lemmas and delegate proofs to an external prover.

CN's linear resources are akin to capabilities in C*: both represent separation-logic assertions. However, resources are mostly inferred and hidden in CN, whereas capabilities are passed explicitly in C*. To support non-backtracking inference, CN imposes syntactic restrictions on resource predicates (reflecting an input/output-mode distinction); C* allows capabilities to carry arbitrary assertions. CN provides a significantly higher level of verification automation and C feature coverage than C*; on the other hand, C* provides more expressive higher-order logic for specification and full control of the proof state, e.g., explicit local reasoning steps can be performed by user proof code, which is not supported in CN.

*VeriFast [28].* VeriFast is a state-of-the-art automated separation-logic verifier for C, Java, and (recently) Rust. Like CN, it covers a broad range of C features and achieves predictable automation via an SMT-backed first-order fragment. It uses a compositional symbolic execution approach [50], maintaining the symbolic heap and path conditions for each control flow branch, and generating VCs when an assertion is met along the way. When desired, users can manually unfold and fold predicates using the proof commands `open` and `close`, and invoke lemma functions.

The primary distinction between C* and VeriFast lies in the extensibility of their proof support. In VeriFast, proof support is limited to a fixed set of built-in ghost statements and built-in induction by writing recursive lemma functions. C* enables users to develop reusable, higher-level proof procedures in C by programming upon its sound proof interface.

*RefinedC [44].* RefinedC is a ownership-refinement type system for C programs, focusing on automated functional-correctness verification with foundational correctness guarantees. It adopts the semantic typing technique [48], interpreting its types and typing rules in Iris [29].

Compared with RefinedC, which binds ownership and invariants to program variables, C* treats these information as separate capabilities, which provides more flexibility of user-guided manual proof steps. On the other hand, the RefinedC approach supports syntax-guided proof automation by customizing typing rules for programming constructs in its separation-logic DSL. It is interesting to investigate whether the RefinedC style of ownership-refinement types and syntax-guided automation can be encoded in C*. We leave this as future work.

## 7 Conclusion

We have presented C*, a language and verifier design that integrates programming and proving in C. It requires users to pass static information as ghost variables called capabilities; together with the structure that separation logic provides, this explicitness enables in-place reasoning steps with clear local reasoning dependencies. Through our sound LCF-style proof interface in C, users can program proof code and proof libraries in the same language used for writing implementation code, enabling better integration of verification and development workflows.

Looking forward, several directions remain for future work, including (i) extending C*'s coverage of C features, (ii) improving proof automation for "trivial-but-frequent" steps, such as overflow checks, to reduce the need for explicit reasoning blocks for routine tasks, and (iii) exploring how we can support encoding rich typing features—such as polymorphism and Rust-style borrowing—as capability passing, potentially bringing additional safety guarantees to C programming at low cost.

Our ultimate goal is to make formal proving more accessible and practical for programmers. We envision that operational reasoning with capabilities should feel as natural as C programming at the static level. Achieving this vision requires larger case studies and more mature proof libraries.

## References


[1] Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007. $L^3$: A Linear Language with Locations. *Fundam. Informaticae* 77, 4 (2007), 397–449. http://content.iospress.com/articles/fundamenta-informaticae/fi77-4-06

[2] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby C. Murray, Gerwin Klein, and Gernot Heiser. 2016. CoGENT: Verifying High-Assurance File System Implementations. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*, Tom Conte and Yuanyuan Zhou (Eds.). ACM, 175–188. doi:10.1145/2872362.2872404

[3] Peter B. Andrews. 1986. *An introduction to mathematical logic and type theory - to truth through proof*. Academic Press.

[4] Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.* 37, 2 (2015), 7:1–7:31. doi:10.1145/2701415

[5] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures (Lecture Notes in Computer Science, Vol. 4111)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.). Springer, 364–387. doi:10.1007/11804192_17

[6] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures (Lecture Notes in Computer Science, Vol. 4111)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.). Springer, 115–137. doi:10.1007/11804192_6

[7] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer. doi:10.1007/978-3-662-07964-5

[8] Allan Blanchard, Frédéric Loulergue, and Nikolai Kosmatov. 2019. Towards Full Proof Automation in Frama-C Using Auto-active Verification. In *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11460)*, Julia M. Badger and Kristin Yvonne Rozier (Eds.). Springer, 88–105. doi:10.1007/978-3-030-20652-9_6

[9] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *J. Autom. Reason.* 61, 1-4 (2018), 367–422. doi:10.1007/S10817-018-9457-5

[10] Arthur Charguéraud and François Pottier. 2008. Functional translation of a calculus of capabilities. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 213–224. doi:10.1145/1411204.1411235

[11] Wei-Ngan Chin, Cristina David, and Cristian Gherghina. 2011. A HIP and SLEEK verification system. In *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 9–10. doi:10.1145/2048147.2048152

[12] Adam Chlipala. 2011. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 234–245. doi:10.1145/1993498.1993526

[13] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 23–42. doi:10.1007/978-3-642-03359-9_2

[14] Sa Cui, Kevin Donnelly, and Hongwei Xi. 2005. ATS: A Language That Combines Programming with Theorem Proving. In *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3717)*, Bernhard Gramlich (Ed.). Springer, 310–320. doi:10.1007/11559306_19

[15] David Delahaye. 2000. A Tactic Language for the System Coq. In *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1955)*, Michel Parigot and Andrei Voronkov (Eds.). Springer, 85–95. doi:10.1007/3-540-44404-1_7

[16] Gabriel Ebner, Guido Martínez, Aseem Rastogi, Thibault Dardinier, Megan Frisella, Tahina Ramananandro, and Nikhil Swamy. 2025. PulseCore: An Impredicative Concurrent Separation Logic for Dependently Typed Programs. *Proc. ACM Program. Lang.* 9, PLDI (2025), 1516–1539. doi:10.1145/3729311

[17] Marco Eilers, Malte Schwerhoff, and Peter Müller. 2024. Verification Algorithms for Automated Separation Logic Verifiers. In *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 14681)*, Arie Gurfinkel and Vijay Ganesh (Eds.). Springer,

362–386. doi:10.1007/978-3-031-65627-9_18

[18] Marco Eilers, Malte Schwerhoff, Alexander J. Summers, and Peter Müller. 2025. Fifteen Years of Viper. In *Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 15931)*, Ruzica Piskac and Zvonimir Rakamaric (Eds.). Springer, 107–123. doi:10.1007/978-3-031-98668-0_5

[19] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 125–128. doi:10.1007/978-3-642-37036-6_8

[20] Mike Gordon. 2000. From LCF to HOL: a short history. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, Gordon D. Plotkin, Colin Stirling, and Mads Tofte (Eds.). The MIT Press, 169–186.

[21] Michael J. C. Gordon, Robin Milner, F. Lockwood Morris, Malcolm C. Newey, and Christopher P. Wadsworth. 1978. A Metalanguage for Interactive Proof in LCF. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM Press, 119–130. doi:10.1145/512760.512773

[22] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. 1979. *Edinburgh LCF*. Lecture Notes in Computer Science, Vol. 78. Springer. doi:10.1007/3-540-09724-4

[23] Samuel Gruetter, Viktor Fukala, and Adam Chlipala. 2024. Live Verification in an Interactive Proof Assistant. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1535–1558. doi:10.1145/3656439

[24] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

[25] John Harrison. 2009. HOL Light: An Overview. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 60–66. doi:10.1007/978-3-642-03359-9_4

[26] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 165–181. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel

[27] C. A. R. Hoare. 1971. Proof of a Program: FIND. *Commun. ACM* 14, 1 (1971), 39–45. doi:10.1145/362452.362489

[28] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 41–55. doi:10.1007/978-3-642-20398-5_4

[29] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. doi:10.1017/S0956796818000151

[30] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel.. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, Jeanna Neefe Matthews and Thomas E. Anderson (Eds.). ACM, 207–220. doi:10.1145/1629575.1629596

[31] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 286–315. doi:10.1145/3586037

[32] Dirk Leinenbach and Thomas Santen. 2009. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5850)*, Ana Cavalcanti and Dennis Dams (Eds.). Springer, 806–809. doi:10.1007/978-3-642-05089-3_51

[33] Gregory Malecha, Gordon Stewart, Frantisek Farka, Jasper Haag, and Yoichi Hirai. 2022. Developing With Formal Methods at BedRock Systems, Inc. *IEEE Secur. Priv.* 20, 3 (2022), 33–42. doi:10.1109/MSEC.2022.3158196

[34] William Mansky and Ke Du. 2024. An Iris Instance for Verifying CompCert C Programs. *Proc. ACM Program. Lang.* 8, POPL (2024), 148–174. doi:10.1145/3632848

[35] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the depths of C: elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 1–15. doi:10.1145/2908080.2908081

[36] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 405–425. doi:10.1007/978-3-319-41528-4_22

[37] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer. doi:10.1007/3-540-45949-9

[38] Lawrence C. Paulson. 1987. *Logic and computation - interactive proof with Cambridge LCF*. Cambridge tracts in theoretical computer science, Vol. 2. Cambridge University Press.

[39] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. GRASShopper - Complete Heap Verification with Mixed Specifications. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8413)*, Erika Ábrahám and Klaus Havelund (Eds.). Springer, 124–139. doi:10.1007/978-3-642-54862-8_9

[40] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified low-level programming embedded in F*. *Proc. ACM Program. Lang.* 1, ICFP (2017), 17:1–17:29. doi:10.1145/3110261

[41] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 2023. CN: Verifying Systems C Code with Separation-Logic Refinement Types. *Proc. ACM Program. Lang.* 7, POPL (2023), 1–32. doi:10.1145/3571194

[42] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. doi:10.1109/LICS.2002.1029817

[43] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 159–169. doi:10.1145/1375581.1375602

[44] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 158–174. doi:10.1145/3453483.3454036

[45] Konrad Slind and Michael Norrish. 2008. A Brief Overview of HOL4. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5170)*, Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar (Eds.). Springer, 28–32. doi:10.1007/978-3-540-71067-7_6

[46] Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.* 34, 1 (2012), 2:1–2:58. doi:10.1145/2160910.2160911

[47] Frederick Smith, David Walker, and J. Gregory Morrisett. 2000. Alias Types. In *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1782)*, Gert Smolka (Ed.). Springer, 366–381. doi:10.1007/3-540-46425-5_24

[48] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *J. ACM* 71, 6 (2024), 40:1–40:75. doi:10.1145/3676954

[49] Harvey Tuch, Gerwin Klein, and Michael Norrish. 2007. Types, bytes, and separation logic. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 97–108. doi:10.1145/1190216.1190234

[50] Frédéric Vogels, Bart Jacobs, and Frank Piessens. 2015. Featherweight VeriFast. *Log. Methods Comput. Sci.* 11, 3 (2015). doi:10.2168/LMCS-11(3:19)2015

[51] Karen Zee, Viktor Kuncak, and Martin C. Rinard. 2009. An integrated proof language for imperative programs. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 338–351. doi:10.1145/1542476.1542514

[52] Litao Zhou, Jianxing Qin, Qinshi Wang, Andrew W. Appel, and Qinxiang Cao. 2024. VST-A: A Foundationally Sound Annotation Verifier. *Proc. ACM Program. Lang.* 8, POPL (2024), 2069–2098. doi:10.1145/3632911

[53] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1789–1806. doi:10.1145/3133956.3134043