C*：面向开发与证明一体化的 C 语言增强

曹奕远　北京大学 · 程序设计语言研究室

CCF 程序设计语言会议 · 2026 年 8 月 5 日

背景

大型证明实践：可行、有益，但**代价不菲**

seL4

微内核的功能正确性证明

Klein et al., SOSP'09

CertiKOS

并发操作系统内核的分层精化证明

Gu et al., OSDI'16

μC/OS-II

实时操作系统内核的关键组件证明

Xu et al., CAV'16

大型证明实践：可行、有益，但**代价不菲**

seL4
微内核的功能正确性证明
Klein et al., SOSP'09

CertiKOS
并发操作系统内核的分层精化证明
Gu et al., OSDI'16

µC/OS-II
实时操作系统内核的关键组件证明
Xu et al., CAV'16

seL4 的代价 Klein et al., CACM'10；Heiser, LCA'15

≈ 2.5 人年
抽象行为规约与具体 C 实现

≈ 20 人年
Isabelle 框架构建 + 证明开发

开发成本 **400$** 每行代码

背景

大型证明实践：可行、有益，但**代价不菲**

seL4
微内核的功能正确性证明
Klein et al., SOSP'09

CertiKOS
并发操作系统内核的分层精化证明
Gu et al., OSDI'16

μC/OS-II
实时操作系统内核的关键组件证明
Xu et al., CAV'16

seL4 的代价 Klein et al., CACM'10；Heiser, LCA'15

≈ 2.5 人年
抽象行为规约与具体 C 实现

≈ 20 人年
Isabelle 框架构建 + 证明开发

开发成本 **400$** 每行代码

如何进一步降低**程序形式化证明**的代价与门槛?

背景

大型证明实践：可行、有益，但**代价不菲**

seL4
微内核的功能正确性证明
Klein et al., SOSP'09

CertiKOS
并发操作系统内核的分层精化证明
Gu et al., OSDI'16

μC/OS-II
实时操作系统内核的关键组件证明
Xu et al., CAV'16

seL4 的代价 Klein et al., CACM'10；Heiser, LCA'15

≈ 2.5 人年
抽象行为规约与具体 C 实现

≈ 20 人年
Isabelle 框架构建 + 证明开发

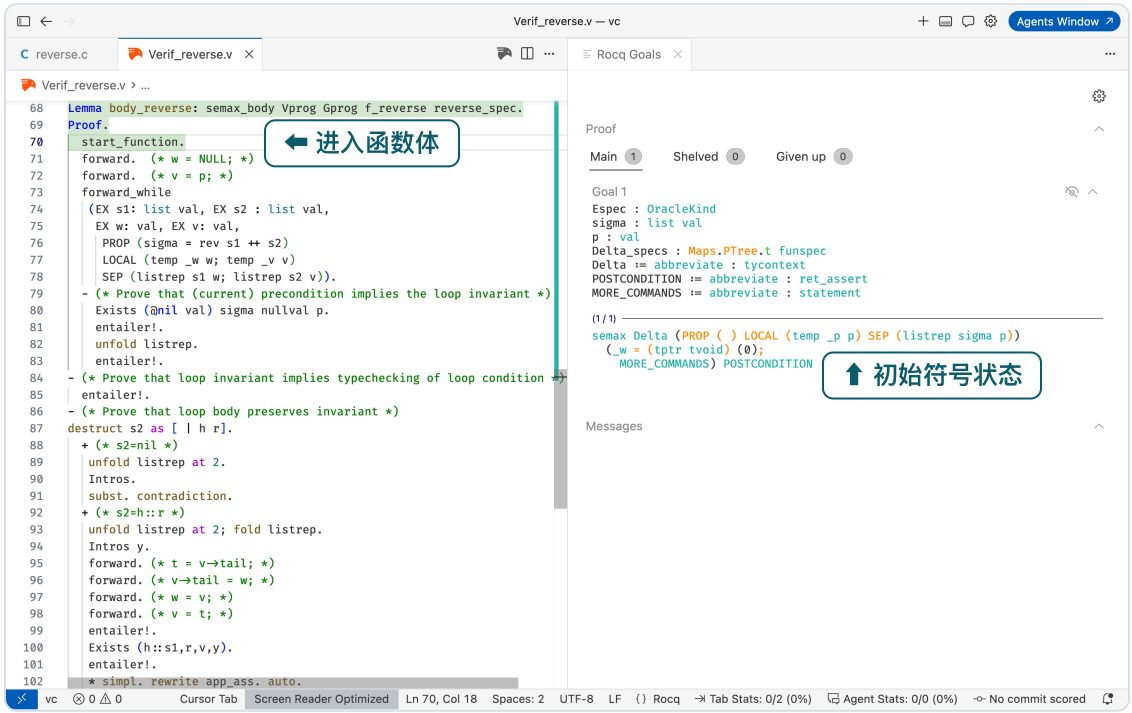
开发成本 **400$** 每行代码

如何进一步降低**程序形式化证明**的代价与门槛?

我们以 VST 为代表，展示现有范式如何验证链表原地反转算法的正确性。[1]

1. 证明来自 Software Foundations · Verifiable C · `Verif_reverse`

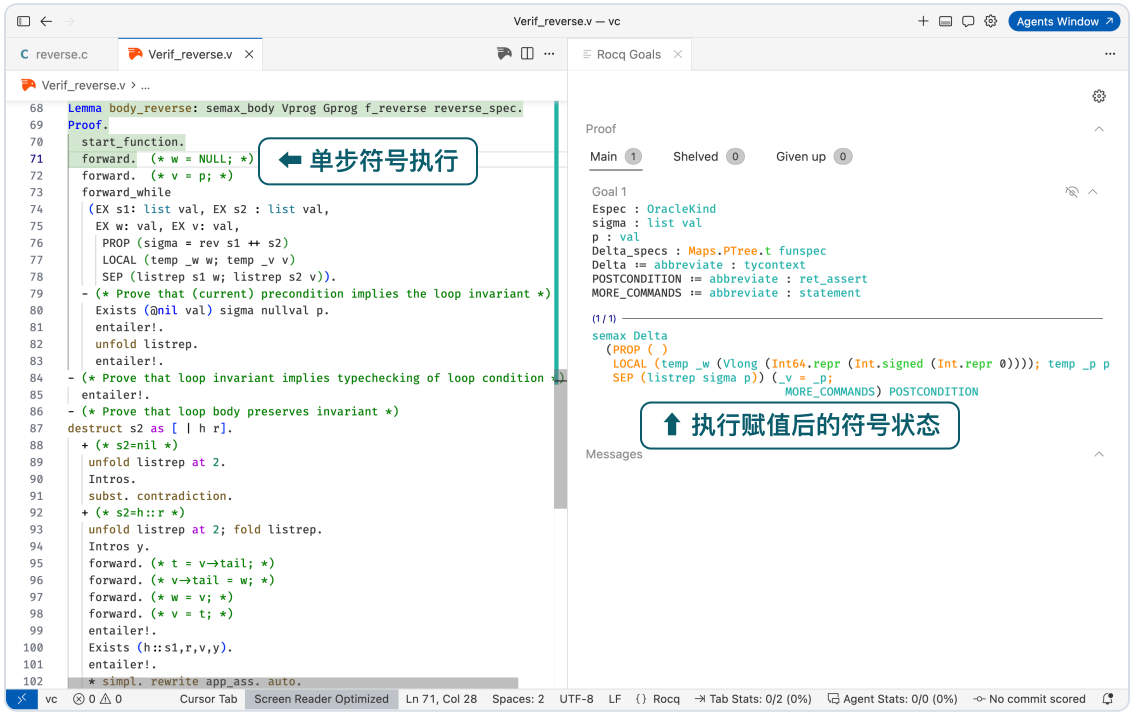
验证状态实时展示




```
struct list *reverse(struct list *p) {
  struct list *w = NULL;
  struct list *v = p;
  while (v) {
    struct list *t = v->tail;
    v->tail = w;
    w = v;
    v = t;
  }
  return w;
}
```

```
WITH sigma : list val, p: val
PRE  [ tptr t_list ]
   PROP ()  PARAMS (p)  SEP (listrep sigma p)
POST [ (tptr t_list) ]
  EX q:val,
   PROP ()  RETURN (q)  SEP (listrep(rev sigma) q)
```

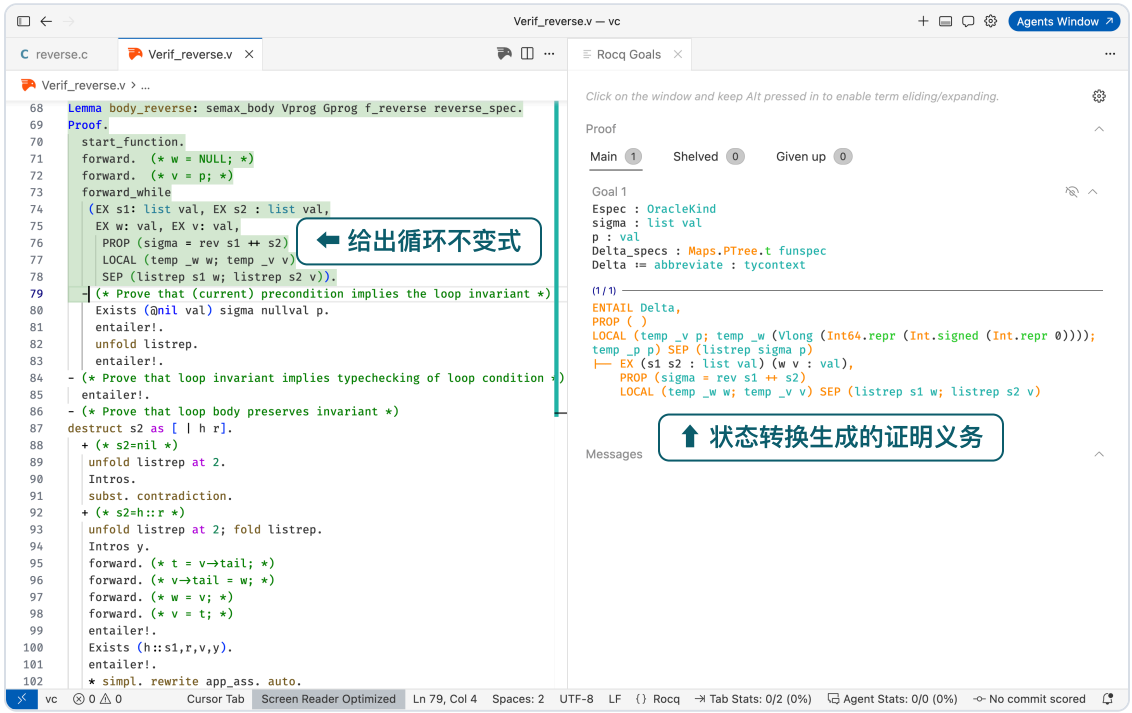
验证状态实时展示



`forward` 前推一条 C 基本语句

符号状态的改变，反映这条指令执行后程序状态的局部变化。

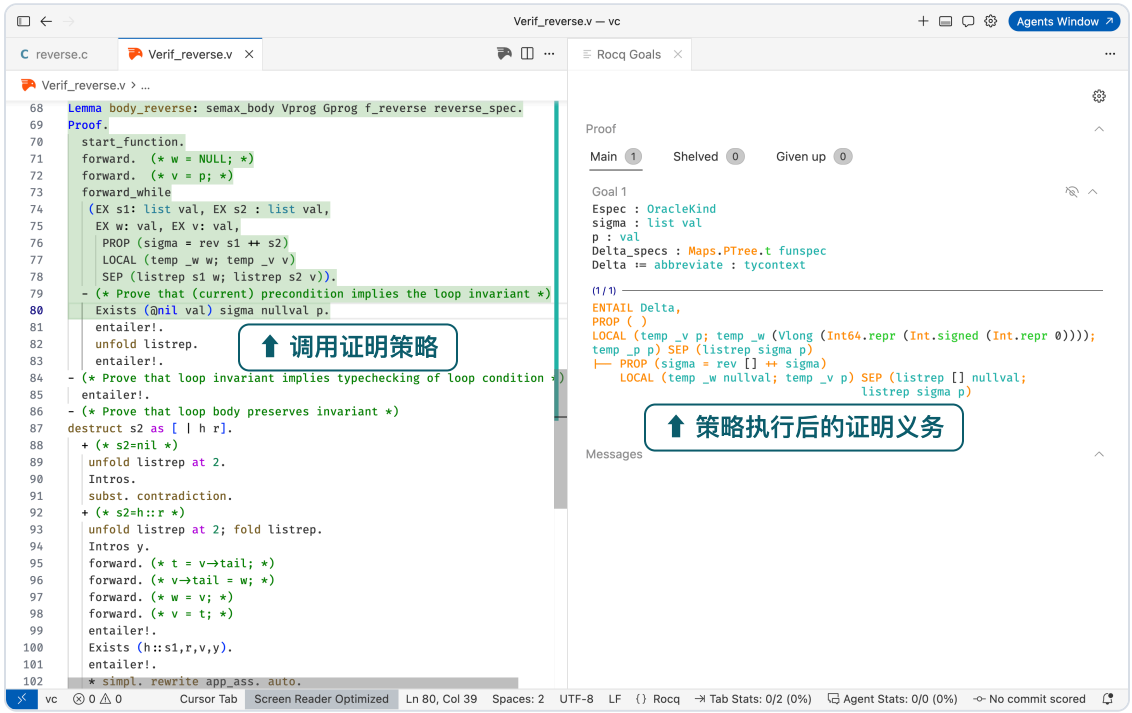
交互式证明



证明义务

循环前的符号状态与循环不变式不一致，产生一条待证的分离逻辑蕴含式。

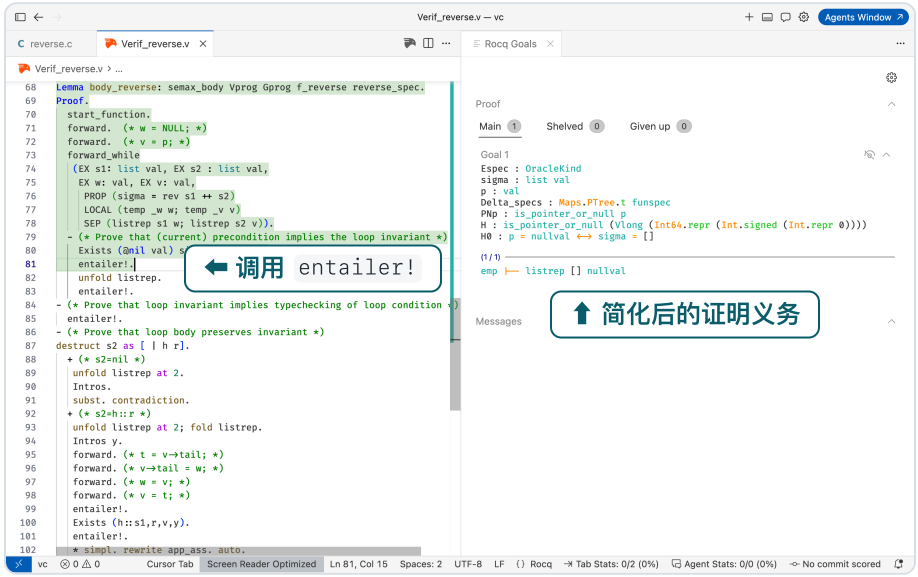
交互式证明

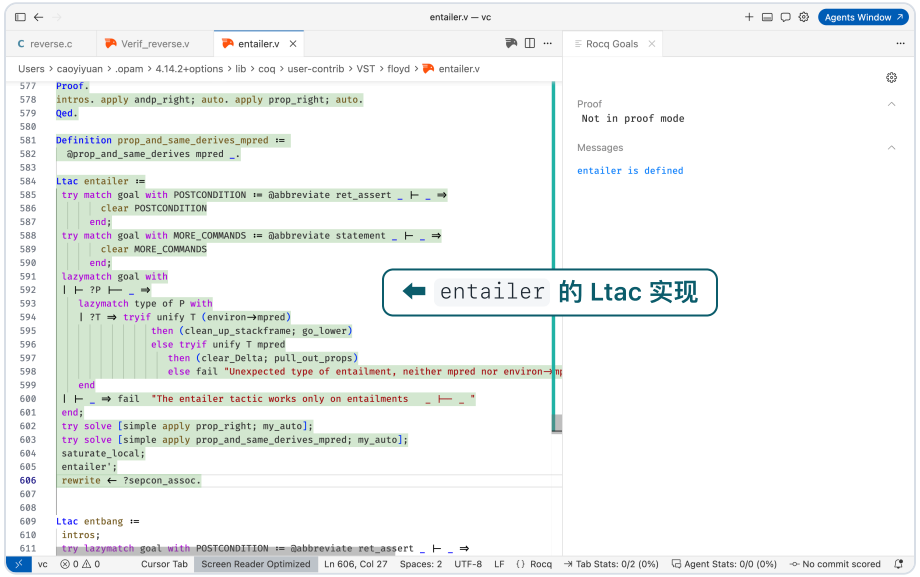


交互式证明

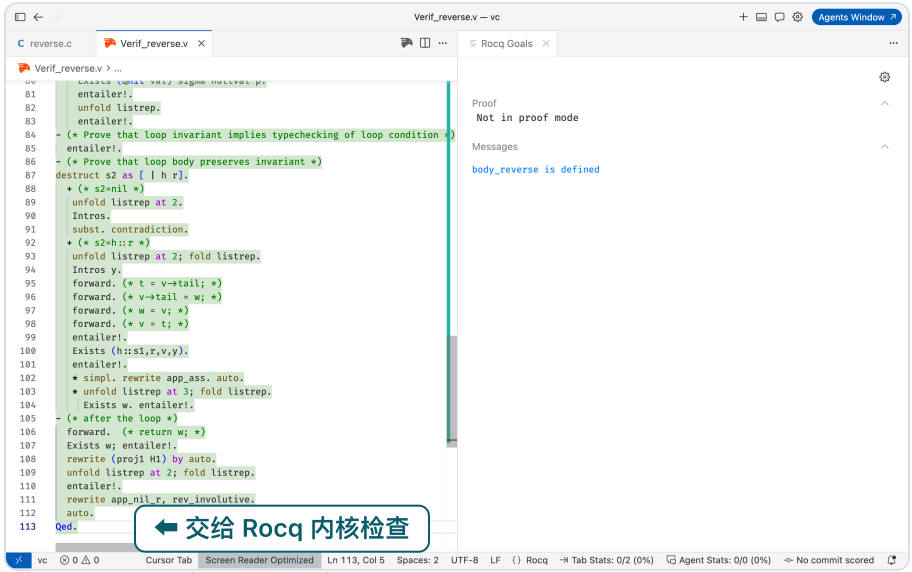
观察证明状态，调用合适的证明策略，逐步推进。

可扩展的证明能力


Verif_reverse.v — vc
Agents Window
reverse.c
Verif_reverse.v
Rocq Goals
Lemma body_reverse: semax_body Vprog Gprog f_reverse reverse_spec.
Proof.
start_function.
forward. (* w = NULL; *)
forward. (* v = p; *)
forward_while
(EX s1: list val, EX s2 : list val,
EX w: val, EX v: val,
PROP (sigma = rev s1 ++ s2)
LOCAL (temp _w w; temp _v v)
SEP (listrep s1 w; listrep s2 v)).
- (* Prove that (current) precondition implies the loop invariant *)
Exists (@nil val) s
entailer!.
unfold listrep.
entailer!.
- (* Prove that loop invariant implies typechecking of loop condition *)
entailer!.
- (* Prove that loop body preserves invariant *)
destruct s2 as [| h r].
+ (* s2=nil *)
unfold listrep at 2.
Intros.
subst. contradiction.
+ (* s2=h::r *)
unfold listrep at 2; fold listrep.
Intros y.
forward. (* t = v→tail; *)
forward. (* v→tail = w; *)
forward. (* w = v; *)
forward. (* v = t; *)
entailer!.
Exists (h::s1,r,v,y).
entailer!.
* simpl. rewrite app_ass. auto.
Proof
Main 1
Shelved 0
Given up 0
Goal 1
Espec : OracleKind
sigma : list val
p : val
Delta_specs : Maps.PTree.t funspec
PNp : is_pointer_or_null p
H : is_pointer_or_null (Vlong (Int64.repr (Int.signed (Int.repr 0))))
H0 : p = nullval ↔ sigma = []
(1/1)
emp ⊢ listrep [] nullval
Messages
⬅ 调用 entailer!
⬆ 简化后的证明义务


entailer.v — vc
Agents Window
reverse.c
Verif_reverse.v
entailer.v
Rocq Goals
Users › caoyiyuan › .opam › 4.14.2+options › lib › coq › user-contrib › VST › floyd › entailer.v
Proof.
intros. apply andp_right; auto. apply prop_right; auto.
Qed.
Definition prop_and_same_derives_mpred :=
@prop_and_same_derives mpred _.
Ltac entailer :=
try match goal with POSTCONDITION := @abbreviate ret_assert _ ⊢ _ ⇒
clear POSTCONDITION
end;
try match goal with MORE_COMMANDS := @abbreviate statement _ ⊢ _ ⇒
clear MORE_COMMANDS
end;
lazymatch goal with
| ⊢ ?P ⊢ _ ⇒
lazymatch type of P with
| ?T ⇒ tryif unify T (environ→mpred)
then (clean_up_stackframe; go_lower)
else tryif unify T mpred
then (clear_Delta; pull_out_props)
else fail "Unexpected type of entailment, neither mpred nor environ→mp
end
| ⊢ _ ⇒ fail "The entailer tactic works only on entailments _ ⊢ _ "
end;
try solve [simple apply prop_right; my_auto];
try solve [simple apply prop_and_same_derives_mpred; my_auto];
saturate_local;
entailer';
rewrite ← ?sepcon_assoc.
Ltac entbang :=
intros;
try lazymatch goal with POSTCONDITION := @abbreviate ret_assert _ ⊢ _ ⇒
Proof
Not in proof mode
Messages
entailer is defined
⬅ entailer 的 Ltac 实现

总结：交互式程序证明框架



三个不可或缺的要素

1. **验证状态可观测**——符号状态与证明目标在边栏实时更新。
2. **证明能力可扩展**——自动化求解策略可由用户定制扩展。
3. **验证结论高可信**——由可信证明内核检查整个证明项。

关键问题

开发与证明的割裂

开发

程序文本	**reverse.c**
编程语言	**C**
开发环境	**C 集成开发环境**

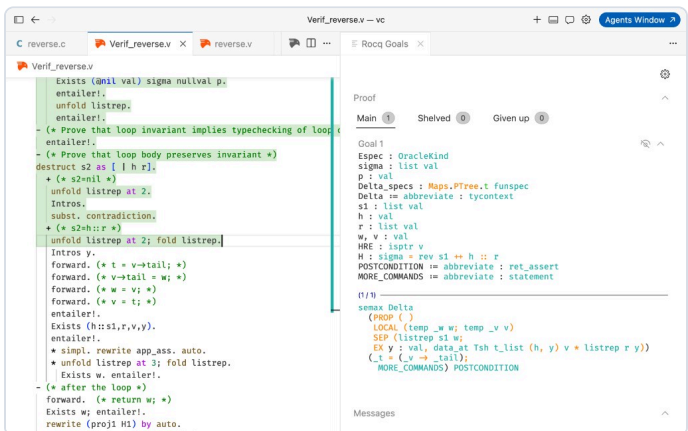
```
struct list *reverse(struct list *p) {
  struct list *w = NULL, *v = p;
  while (v) {
    struct list *t = v->tail;
    v->tail = w;
    w = v;
    v = t;
  }
  return w;
}
```

切换

切换

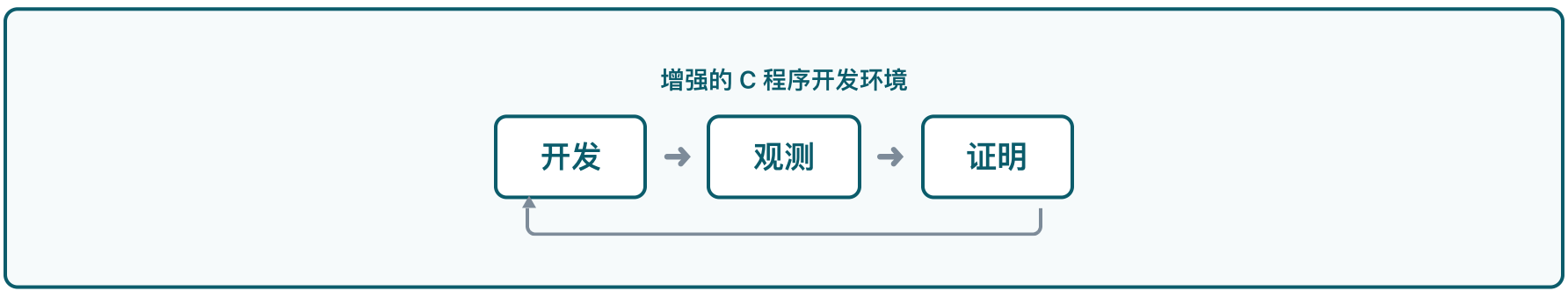
证明

程序文本	**Verif_reverse.v**
编程语言	**Ltac**
开发环境	**Rocq**



解决方案

C*：开发与证明的一体化



验证状态可观测

符号状态与证明状态随光标实时更新

证明能力可扩展

证明函数与证明库用 C 编写和扩展

验证结论高可信

符号状态的每次转换都由定理背书

解决方案

C* 中的链表反转

```
struct list *reverse(struct list *p)
  PARAM(`l:(int)list`)
  REQUIRE(`sll p l`)
  ENSURE(`sll __return (REVERSE l)`)
{
  struct list *w = NULL, *v = p;
  PROOF fold_sll_null();
  PROOF prove_fact_by(`l == (REVERSE []) ++ l`, simp_conv, …);
  while (v)
    INV(`exists l1 l2 w_v v_v. fact(l == (REVERSE l1) ++ l2) **
      sll w_v l1 ** sll v_v l2 ** data_at v__addr Tptr v_v **
      data_at w__addr Tptr w_v ** data_at p__addr Tptr p__pre`)
  {
    PROOF unfold_sll_not_null(`v_v`);
    struct list *t = v->tail;
    v->tail = w;  w = v;  v = t;
    PROOF {
      fold_sll_not_null(`v_v`);
      prove_fact_by(`l == (REVERSE (hd :: l1)) ++ tl`, simp_conv, …);
    }
  }
  PROOF { /* … 出口：展开空链表，化简出 l1 == REVERSE l … */ }
  return w;
}
```

解决方案

C* 中的链表反转

```
struct list *reverse(struct list *p)
  PARAM(`l:(int)list`)
  REQUIRE(`sll p l`)
  ENSURE(`sll __return (REVERSE l)`)
{
  struct list *w = NULL, *v = p;
  PROOF fold_sll_null();
  PROOF prove_fact_by(`l == (REVERSE []) ++ l`, simp_conv, …);
  while (v)
    INV(`exists l1 l2 w_v v_v. fact(l == (REVERSE l1) ++ l2) **
      sll w_v l1 ** sll v_v l2 ** data_at v__addr Tptr v_v **
      data_at w__addr Tptr w_v ** data_at p__addr Tptr p__pre`)
  {
    PROOF unfold_sll_not_null(`v_v`);
    struct list *t = v->tail;
    v->tail = w;  w = v;  v = t;
    PROOF {
      fold_sll_not_null(`v_v`);
      prove_fact_by(`l == (REVERSE (hd :: l1)) ++ tl`, simp_conv, …);
    }
  }
  PROOF { /* … 出口：展开空链表，化简出 l1 == REVERSE l … */ }
  return w;
}
```

实现

与原来一样的 C 实现代码

解决方案

C* 中的链表反转

```
struct list *reverse(struct list *p)
  PARAM(`l:(int)list`)
  REQUIRE(`sll p l`)
  ENSURE(`sll __return (REVERSE l)`)
{
  struct list *w = NULL, *v = p;
  PROOF fold_sll_null();
  PROOF prove_fact_by(`l == (REVERSE []) ++ l`, simp_conv, …);
  while (v)
    INV(`exists l1 l2 w_v v_v. fact(l == (REVERSE l1) ++ l2) **
      sll w_v l1 ** sll v_v l2 ** data_at v__addr Tptr v_v **
      data_at w__addr Tptr w_v ** data_at p__addr Tptr p__pre`)
  {
    PROOF unfold_sll_not_null(`v_v`);
    struct list *t = v->tail;
    v->tail = w;  w = v;  v = t;
    PROOF {
      fold_sll_not_null(`v_v`);
      prove_fact_by(`l == (REVERSE (hd :: l1)) ++ tl`, simp_conv, …);
    }
  }
  PROOF { /* … 出口：展开空链表，化简出 l1 == REVERSE l … */ }
  return w;
}
```

实现

与原来一样的 C 实现代码

规约

断言写在反引号里，用高阶逻辑与分离逻辑联结词表达

解决方案

C* 中的链表反转

```
struct list *reverse(struct list *p)
  PARAM(`l:(int)list`)
  REQUIRE(`sll p l`)
  ENSURE(`sll __return (REVERSE l)`)
{
  struct list *w = NULL, *v = p;
  PROOF fold_sll_null();
  PROOF prove_fact_by(`l == (REVERSE []) ++ l`, simp_conv, …);
  while (v)
    INV(`exists l1 l2 w_v v_v. fact(l == (REVERSE l1) ++ l2) **
      sll w_v l1 ** sll v_v l2 ** data_at v__addr Tptr v_v **
      data_at w__addr Tptr w_v ** data_at p__addr Tptr p__pre`)
  {
    PROOF unfold_sll_not_null(`v_v`);
    struct list *t = v->tail;
    v->tail = w;  w = v;  v = t;
    PROOF {
      fold_sll_not_null(`v_v`);
      prove_fact_by(`l == (REVERSE (hd :: l1)) ++ tl`, simp_conv, …);
    }
  }
  PROOF { /* … 出口：展开空链表，化简出 l1 == REVERSE l … */ }
  return w;
}
```

实现

与原来一样的 C 实现代码

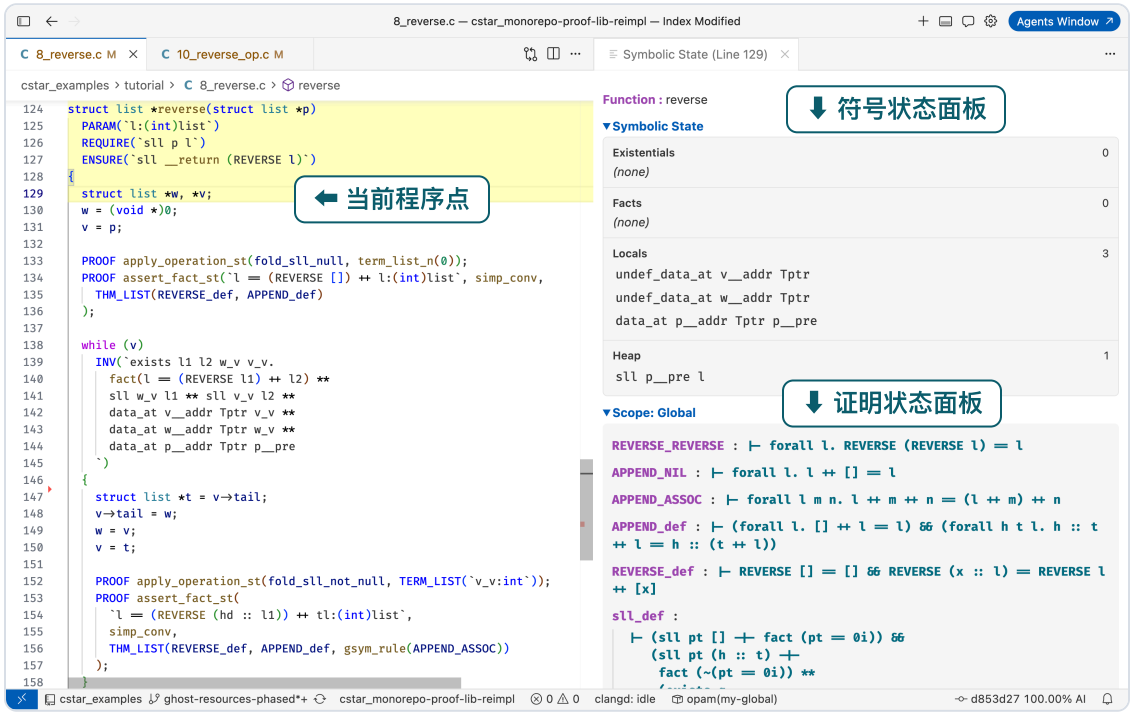
规约

断言写在反引号里，用高阶逻辑与分离逻辑联结词表达

证明

`PROOF` 标注的 C 语句：计算产生定理值并更新验证状态

展示光标处的验证状态



开发过程中**实时同步展示验证状态**，无需切换环境。

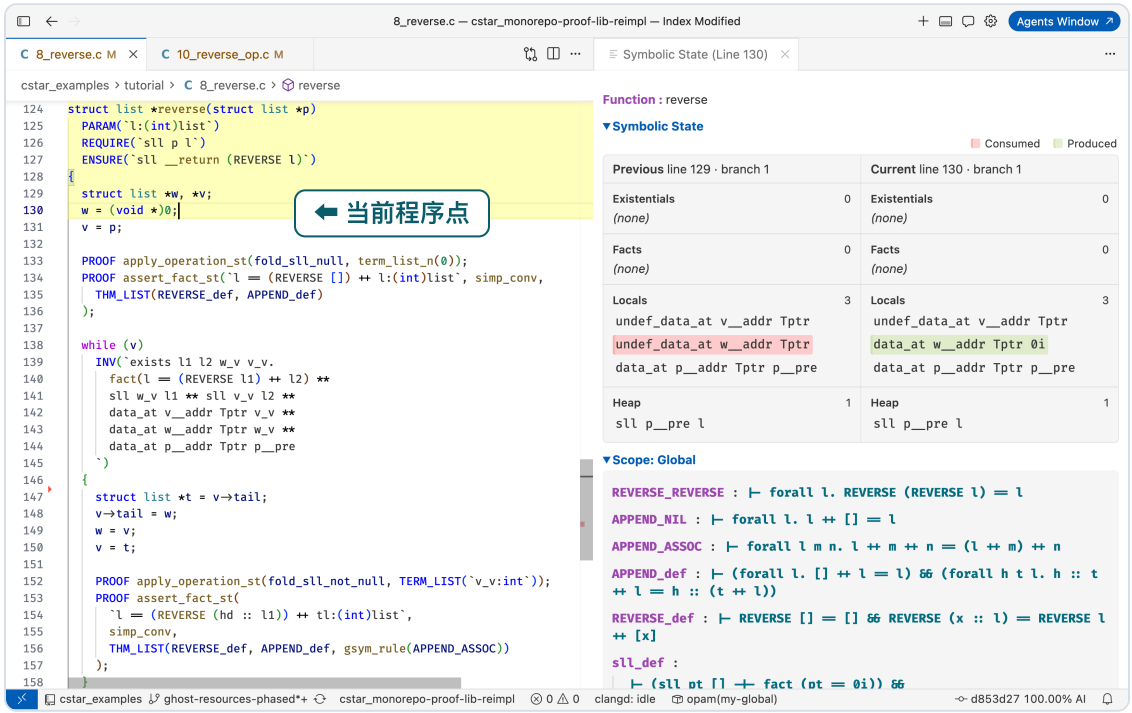
符号状态

`Existentials` 存在量词 · `Facts` 已知事实
`Locals` 局部变量 · `Heap` 堆资源

证明状态

可使用的定理、待证义务等

写下实现语句，更新验证状态

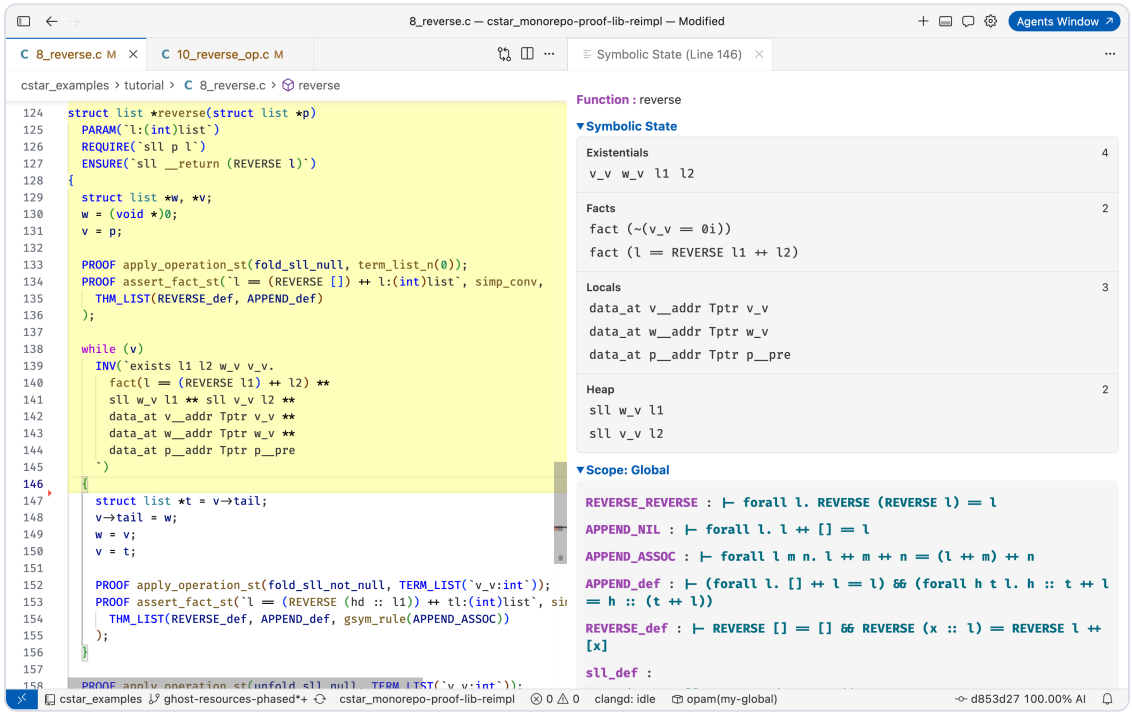


程序实现指令自动推进符号状态。

面板展示前后对比

左半是执行前的状态，右半是执行后的状态。

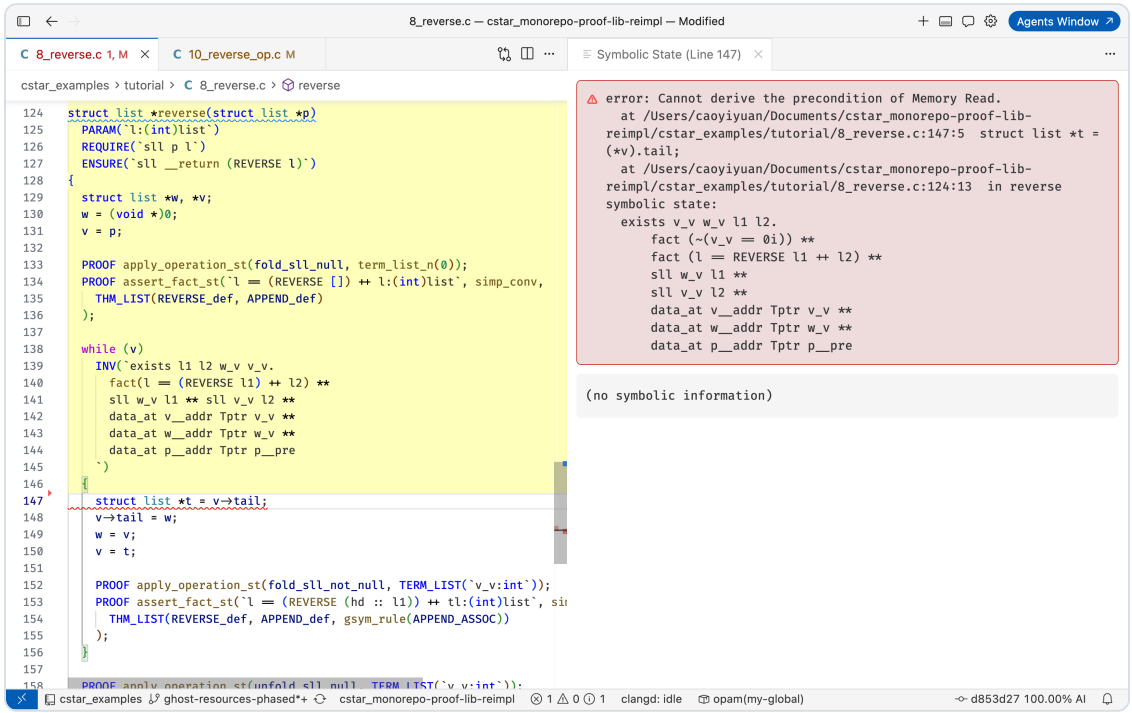
符号状态与语句要求不一致时**就地报错**



循环体入口的符号状态

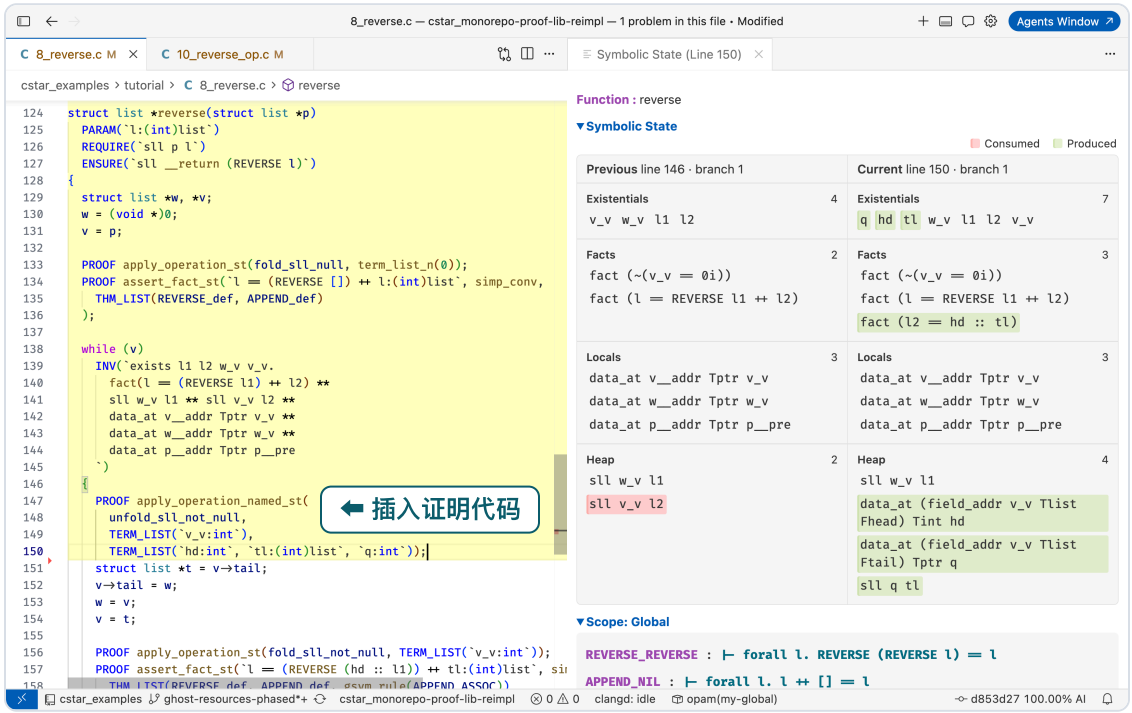
循环不变式加上循环条件带来的路径条件。

符号状态与语句要求不一致时**就地报错**



持有的是整条链表的所有权 `sll v_v l2`，而读字段需要的是字段所有权 `data_at`。

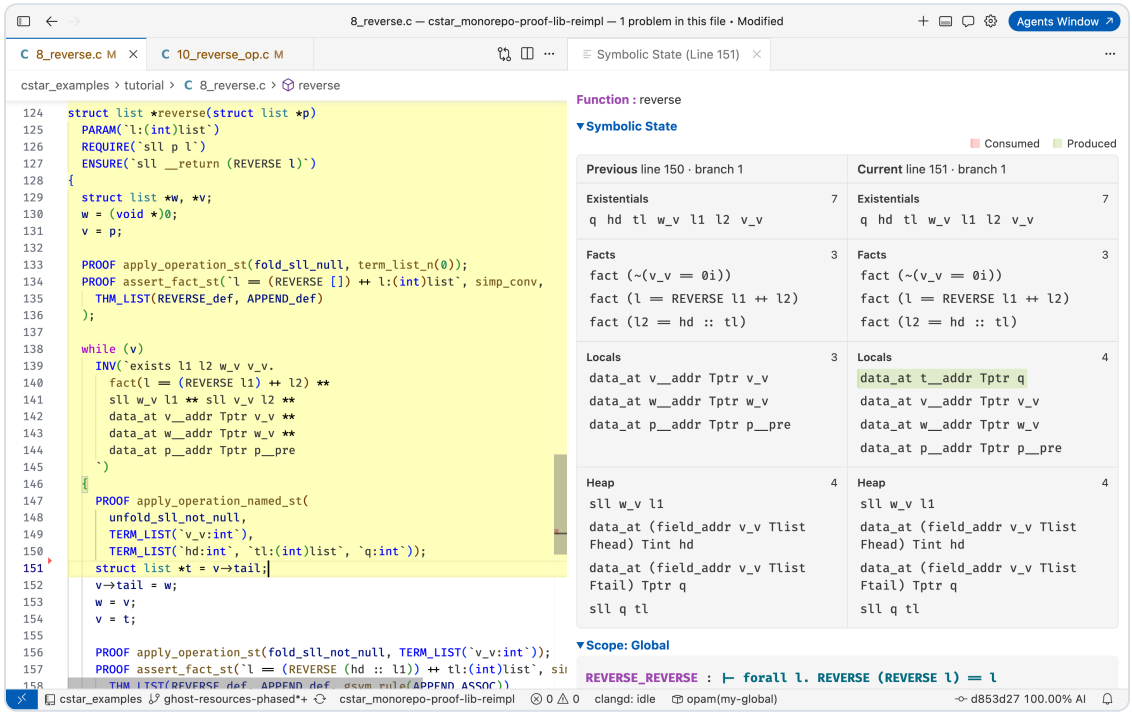
通过证明转换符号状态的内存视图后继续



通过证明变换符号状态的内存视图

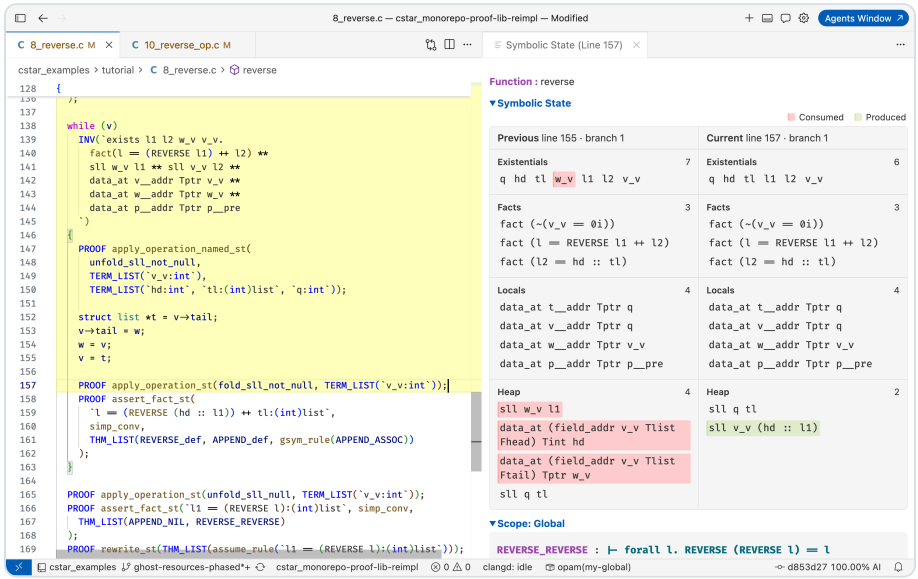
消耗 `sll v_v l2`，分离出头结点的两个字段所有权 `data_at` 与剩余段 `sll q tl`。

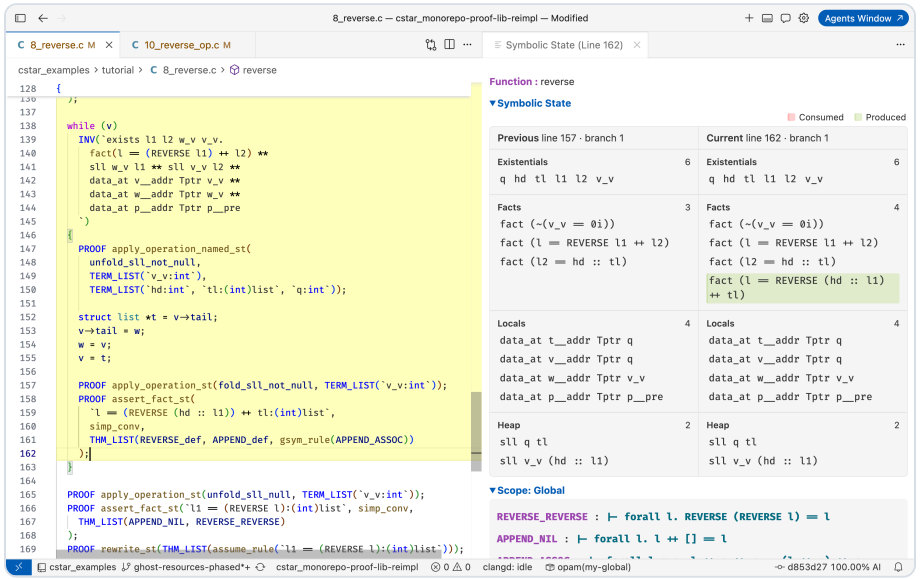
通过证明转换符号状态的内存视图后继续



读字段的前置条件可以推出，开发继续

重新建立循环不变式

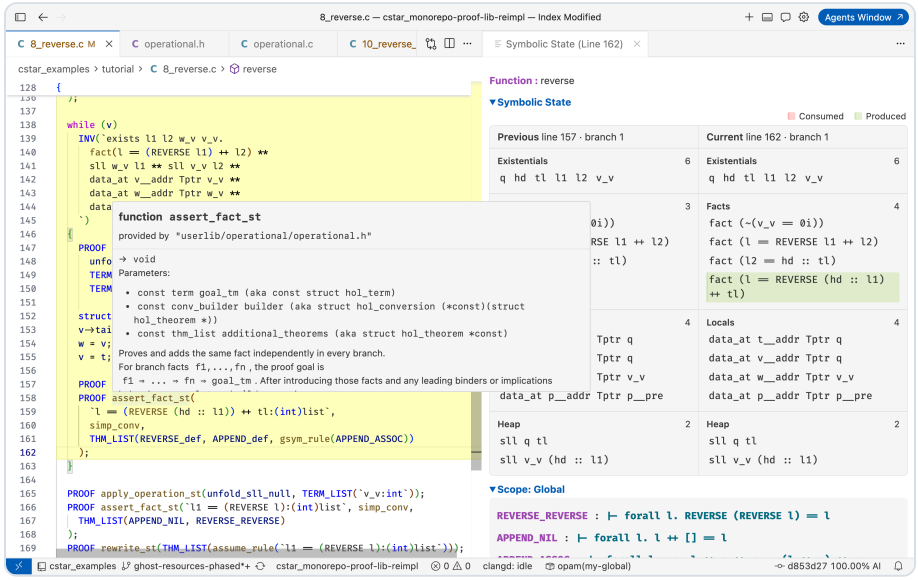




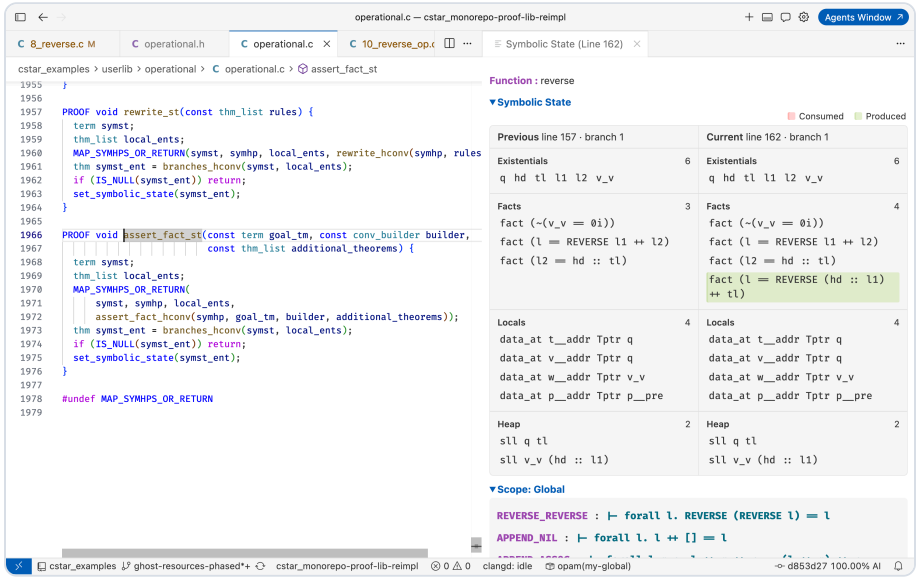
左：把新链接的结点合入已反转的链表

右：补充证明循环不变式所需的逻辑事实

证明代码复用 C 语言与编辑器支持



悬停 — 签名与文档



跳转定义 — 落进证明库源码

一体化范式 · C* 演示

证明函数如何改变验证状态

```
PROOF void prove_fact_by(term goal, conv_builder prover, thm_list extra) {
  term symst = get_symbolic_state();                  // 取: 读出当前符号状态 S
  term_list facts = symst_facts(symst);
  gnode root = gnode_new(facts, goal);
  CONV_TAC(root, prover, extra);
  thm vshift = local_apply(symst, gnode_prove(root)); // 证: 构造视图变换定理 S |-- S'
  set_symbolic_state(vshift);                         // 交: 以定理为可信凭据更新状态为 S'
}
```

1. 当前默认的符号执行器是 QCP (Wu et al., TASE 2026)。 感谢上海交通大学曹钦翔团队的长期合作与支持!
2. C 中 LCF 风格的可信证明接口实现架构基于我们的前序工作 Cao et al., TASE 2026。
3. 当前默认的证明内核是 HOL Light。

一体化范式 · C* 演示

证明函数如何改变验证状态

```
PROOF void prove_fact_by(term goal, conv_builder prover, thm_list extra) {
  term symst = get_symbolic_state();                  // 取: 读出当前符号状态 S
  term_list facts = symst_facts(symst);
  gnode root = gnode_new(facts, goal);
  CONV_TAC(root, prover, extra);
  thm vshift = local_apply(symst, gnode_prove(root)); // 证: 构造视图变换定理 S |-- S'
  set_symbolic_state(vshift);                         // 交: 以定理为可信凭据更新状态为 S'
}
```

1. 当前默认的符号执行器是 QCP (Wu et al., TASE 2026)。 感谢上海交通大学曹钦翔团队的长期合作与支持!
2. C 中 LCF 风格的可信证明接口实现架构基于我们的前序工作 Cao et al., TASE 2026。
3. 当前默认的证明内核是 HOL Light。

一体化范式 · C* 演示

证明函数如何改变验证状态

```
PROOF void prove_fact_by(term goal, conv_builder prover, thm_list extra) {
  term symst = get_symbolic_state();                    // 取: 读出当前符号状态 S
  term_list facts = symst_facts(symst);
  gnode root = gnode_new(facts, goal);
  CONV_TAC(root, prover, extra);
  thm vshift = local_apply(symst, gnode_prove(root)); // 证: 构造视图变换定理 S |-- S'
  set_symbolic_state(vshift);                           // 交: 以定理为可信凭据更新状态为 S'
}
```

1. 当前默认的符号执行器是 QCP (Wu et al., TASE 2026)。 感谢上海交通大学曹钦翔团队的长期合作与支持!
2. C 中 LCF 风格的可信证明接口实现架构基于我们的前序工作 Cao et al., TASE 2026。
3. 当前默认的证明内核是 HOL Light。

一体化范式 · C* 演示

证明函数如何改变验证状态

```
PROOF void prove_fact_by(term goal, conv_builder prover, thm_list extra) {
  term symst = get_symbolic_state();                    // 取: 读出当前符号状态 S
  term_list facts = symst_facts(symst);
  gnode root = gnode_new(facts, goal);
  CONV_TAC(root, prover, extra);
  thm vshift = local_apply(symst, gnode_prove(root)); // 证: 构造视图变换定理 S |-- S'
  set_symbolic_state(vshift);                           // 交: 以定理为可信凭据更新状态为 S'
}
```

1. 当前默认的符号执行器是 QCP (Wu et al., TASE 2026)。 感谢上海交通大学曹钦翔团队的长期合作与支持!
2. C 中 LCF 风格的可信证明接口实现架构基于我们的前序工作 Cao et al., TASE 2026。
3. 当前默认的证明内核是 HOL Light。

证明函数如何改变验证状态

```
PROOF void prove_fact_by(term goal, conv_builder prover, thm_list extra) {
  term symst = get_symbolic_state();                  // 取: 读出当前符号状态 S
  term_list facts = symst_facts(symst);
  gnode root = gnode_new(facts, goal);
  CONV_TAC(root, prover, extra);
  thm vshift = local_apply(symst, gnode_prove(root)); // 证: 构造视图变换定理 S |-- S'
  set_symbolic_state(vshift);                         // 交: 以定理为可信凭据更新状态为 S'
}
```

> 证明函数**不在可信计算基内**。

1. 当前默认的符号执行器是 QCP (Wu et al., TASE 2026)。 感谢上海交通大学曹钦翔团队的长期合作与支持!
2. C 中 LCF 风格的可信证明接口实现架构基于我们的前序工作 Cao et al., TASE 2026。
3. 当前默认的证明内核是 HOL Light。

总结

程序文本的一体化	**证明代码就地嵌入在普通 C 实现代码之中**
编程语言的一体化	**用 C 书写证明，以 C 代码库的形式扩展证明支持**
开发环境的一体化	**在 C 的编译器与语言服务器之上构建可实时观测验证状态的 IDE**

证明库

标准证明库约 6.5 千行（分离逻辑推理规则、目标树与后向证明策略），用户端通用扩展库约 2 千行（操作式推理函数等），以及列表、树、整数算术等领域理论库。

示例

约 40 个经验证程序：链表与数组算法（查找、排序、`memset`）、二叉搜索树、内存分配器（Google pKVM `hyp_allocator`、`buddy_allocator`），以及从 VST、VeriFast、LiveVerif 迁移的案例。

教程

安装指引与渐进式教程：从函数契约与循环不变式，到在 C 中进行定理证明与证明库扩充。

未来工作

- 扩展并发推理支持，目标是验证真实世界的系统软件。
- 定量评估真实任务上 C* 与其他非一体化的程序验证工具的效率差异。

C* 官方网站

cstarlang.org

安装指引 · 示例教程 · 证明库文档

讨论

一体化对大模型编写可信代码同样有益?

上下文局部性

程序与证明处于同一位置，完成任务无需大量跨文件跳转，上下文更局部。

反馈信号的质量

基于 SMT 求解的工具在验证失败时反馈往往不精确，且执行时间不稳定。C* 对证明代码和普通 C 语句都维护稳定、逐行的符号状态，可以更精确地指导模型修改代码。

安全性由小内核保证

模型生成的证明代码不在可信计算基中，出错的后果是证明不通过，而不是产生伪证。

初步实验

约 150 行的内存分配器 C 实现。 使用 Codex 自动完成所有证明开发。 等额计算 API 调用开销约 150 美元。

程序与规约取自 Appel & Naumann. Verified Sequential Malloc/Free. ISMM 2020 · 实现代码 `VST/progs/memmgr/malloc.c`