A HOL Theorem Proving Interface for C

Yiyuan Cao[1,2], Jiayi Zhuang[1], Jinkai Fan[1], Di Wang[1,2] ✉, and Zhenjiang Hu[1,2]

[1] Key Lab of HCST (PKU), MOE; SCS, Peking University, China
{yiyuan,wybxc,2200013125}@stu.pku.edu.cn, {wangdi95,huzj}@pku.edu.cn
[2] Zhongguancun Laboratory

Abstract. C programmers who wish to perform interactive theorem proving (e.g., when verifying critical code they write) currently face a language barrier: working with existing provers requires familiarity with their own proof languages—usually a custom tactic language or a functional programming language—far removed from the C programming environment and paradigm. To bridge this gap, we present the first higher-order logic (HOL) theorem proving interface for C. Our interface is *comprehensive*: it covers the core data types and operations of HOL theorem proving in the LCF style, readily usable for forward derivation of theorems by invoking existing inference rules. Our interface is *trustworthy*: by extending the LCF architecture with a client-server model, it ensures every theorem constructed by arbitrary C code is necessarily derived from a small trusted proof kernel. Finally, programming derived proof facilities in C is *practical*: we demonstrate this by implementing commonly used tactics for goal-directed reasoning as a library on top of our interface, which utilizes a goal-tree data structure dedicated to the bookkeeping of proof states and applies imperative programming patterns naturally.

Keywords: C programming · HOL theorem proving · LCF architecture.

1 Introduction

Interactive theorem proving in expressive higher-order logics (abbreviated as HOL) has been successfully applied by expert teams to verify critical software, including OS kernels [9,12], compilers [14,13], and cryptographic implementations [29,2]. However, this powerful technique remains largely inaccessible to the broader community of systems programmers—despite the clear need for formal proofs, notably when verifying the security and correctness of critical software they write. One crucial barrier is that working in established theorem provers—e.g., Rocq, Isabelle, HOL4, Agda, Lean—requires mastery of custom tactic languages [5,16], declarative proof languages [28], or (dependently-typed) functional programming languages [6,15,18], out of reach of the typical C programming environment and paradigm.

We envision a world where C programmers can conduct theorem proving in a familiar C development setting, without switching to a foreign proof language and external environment. Just as a developer includes `<gtk/gtk.h>` and links

against the GTK library[3] to create a graphical interface, they should be able to include a theorem-proving header to develop theories and derive theorems in C, using operations exposed by the interface. Furthermore, just as developers customize GUI libraries by composing primitive functions into specialized widgets, they should be able to build derived facilities and proof strategies beyond the interface provided, naturally, using C programming techniques. While libraries for domain-specific tasks such as GUI design and database access are routinely used by developers, we are not aware of any comparable interface for theorem proving in C, let alone a usable suite of derived libraries.

The established method for engineering a trustworthy and programmable theorem prover is the LCF approach [6,7]. However, the meta languages of traditional LCF-style provers are ML-like functional languages (e.g., Standard ML and OCaml), featuring support for type abstraction, higher-order functions, and exceptions. Adapting the LCF approach to C presents two fundamental challenges:

(1) C's type system does not enforce type abstraction—unconstrained memory mutation and type casting make it possible to penetrate module boundaries and circumvent the guarded interface of the proof kernel that implements the primitive inference rules of the logic. This undermines the trustworthiness of the proof system: an inadvertent programming mistake or a piece of malicious code can lead to the construction of invalid theorems.
(2) C does not support higher-order functions or exceptions natively. In the LCF tradition, these features are considered essential: not only for defining the kernel interface, but more critically, for building essential proof facilities. The prime example is the notion of tactics for backward (i.e., goal-directed) reasoning [17]. Canonically, tactics are implemented as higher-order functions that return a list of subgoals together with a validation closure for reconstructing forward proofs from solved subgoals. These tactics are combined using tacticals, which compose validators via higher-order combinators; the implementation of tacticals additionally uses exceptions for backtracking the proof state.

We address these challenges in the implementation of our interface as well as in programming proof facilities as derived libraries in C. For the trustworthiness challenge (1), we extend the traditional LCF architecture with a client-server model. The proof kernel runs in a separate server process (we reuse the HOL Light theorem prover [11]). User code on the client side manipulates logical objects only through opaque handles via a core interface, so that process isolation prevents forgery of invalid theorems. For the proof programming challenge (2), we adopt C-idiomatic patterns at two levels. At the core interface level, we expose a first-order interface and use status-code error handling. More importantly, at the derived library level, we show that facilities for goal-directed reasoning—traditionally reliant on higher-order functions and exceptions—can be naturally realized in C. Our design uses a goal-tree data structure that explicitly reifies

[3] A popular GUI library for C (https://gtk.org/; accessed 2026-02-21).

the proof structure and stores validator functions at each internal node, together with imperative control flow for implementing more complex proof strategies.

Contributions. Our contributions are three-fold:

1. **A comprehensive core interface for HOL theorem proving in C (Section 3).** It covers term manipulation, inference rules, definitional principles, and proof-search procedures, enabling C developers to readily perform forward derivation of theorems in the LCF-style.
2. **A trustworthy implementation of the core interface (Section 4).** Our architecture guarantees that all theorems constructed by arbitrary user-written C code are necessarily derived from a small, trusted proof kernel.
3. **A practical demonstration of proof programming in C (Section 5).** Building on the core interface, we program support for goal-directed reasoning as derived proof libraries in C, including subgoaling functionalities and commonly used tactics for HOL reasoning.

Artifact availability. The source code of the core interface and derived libraries presented in this paper is available online [4].

2 Background

This section reviews the necessary background on higher-order logic and the LCF approach to theorem proving.

2.1 Higher-Order Logic

Higher-order logic (HOL) is the logic underlying a family of modern theorem provers including HOL Light [11], HOL4 [26], and Isabelle/HOL [21]. We briefly describe its syntax. For a full account of its semantics, see Pitts [25].

Types. HOL types are built from type variables (α, β, ...) and type constructors applied to argument types (e.g., natural numbers num, finite lists $list(\alpha)$, etc.). Primitive type constructors include the Boolean type $bool$ and the function type $\alpha \to \beta$ (written in infix), with their usual set-theoretic interpretations.

Terms. HOL terms are those of a simply-typed lambda calculus, comprising four primitive forms: variables x, constants c, function applications $f\,x$, and lambda abstractions $\lambda x.\, t$. Every term has a type; *formulas* are terms of Boolean type. The most important primitive constant is the (polymorphic) equality predicate $(=) : \alpha \to \alpha \to bool$. Common logical connectives are not built-in primitives but are defined in terms of equality. For example, conjunction and universal quantification (using higher-order abstract syntax [24]) can be defined as [11]:

$$\wedge \stackrel{\text{def}}{=} \lambda p.\, \lambda q.\, (\lambda f.\, f\, p\, q) = (\lambda f.\, f \top \top) \quad \text{and} \quad \forall \stackrel{\text{def}}{=} \lambda P.\, (P = \lambda x.\, \top)\ .$$

Theorems. A theorem is a provable sequent, represented as $A \vdash p$, where A is a finite set of formulas (the assumptions) and p is a formula (the conclusion). Formally, a proof of a theorem is a sequence of valid applications of primitive inference rules to existing theorems and axioms, ending with the desired sequent. For example, the following primitive inference rule MkComb can be used to prove that two applicative terms are equal, given that their function and argument terms are provably equal and their types are compatible:[4]

$$\frac{\Gamma_1 \vdash f_1 = f_2 : \mathcal{A} \to \mathcal{B} \qquad \Gamma_2 \vdash t_1 = t_2 : \mathcal{A}}{\Gamma_1 \cup \Gamma_2 \vdash f_1\, t_1 = f_2\, t_2 : \mathcal{B}} \quad \textsc{MkComb}$$

Higher-order logic contains a minimal set of primitive inference rules (only 10 rules in the HOL Light variant). Together with suitable axioms (e.g., the axiom of choice, excluded middle, etc.), they suffice as an expressive foundation for formal mathematical reasoning [10].

2.2 LCF-Style Theorem Proving

The LCF approach, introduced by Milner and colleagues in the Edinburgh LCF prover [6,7], is the established method for engineering a programmable yet trustworthy theorem prover. It mechanizes the object logic inside a general-purpose programming language (called the *meta language*, traditionally an ML-like functional language[5]). Logical objects (terms, theorems, etc.) are encoded as symbolic data values with predefined operations. For example, predefined constants of the theorem type represent axioms of the logic; and predefined operations that return theorems correspond to primitive inference rules.

Programmability. This approach gives a fully programmable theorem prover. The meta language readily serves as the proof language for users to interact with the prover. Most directly, to prove is just to program code that invokes the predefined operations to perform the inference steps. For example, the following HOL Light code proves $f\ x = x \vdash f\ (f\ x) = x$ by a sequence of rule applications:

```
let th1 = ASSUME `(f:A->A) x = x` in          (*  f x = x ⊢ f x = x          *)
let th2 = MK_COMB (REFL `(f:A->A)`, th1) in (*  f x = x ⊢ f (f x) = f x  *)
TRANS th2 th1                                 (*  f x = x ⊢ f (f x) = x    *)
```

Furthermore, derived inference rules and proof facilities can be programmed using the full power of a general-purpose programming language. A facility considered essential since the inception of LCF-style theorem proving [6,7,17] is goal-directed proof strategies, implemented as *tactics*. For example, the following HOL Light code first states the goal to be proved, i.e., commutativity of natural number addition, and proves it by composing a tactic that first applies

[4] More precisely, it is a rule scheme, since it is parameterized by meta-level type variables $\mathcal{A}$ and $\mathcal{B}$; not to be confused with object-logic type variables α, β, etc.

[5] The ML (an acronym for meta language) language was originally designed specifically for implementing and extending the Edinburgh LCF prover.

induction and then discharges the two subgoals by rewriting with the induction hypotheses and the definitional clauses of addition:

```
let ADD_SYM = prove
  (‘forall m n:num. m + n = n + m‘,
   INDUCT_TAC THEN
   (* subgoal 1: ⊢ ∀n. 0 + n = n + 0  *)
   (* subgoal 2: ∀n. m + n = n + m ⊢ ∀n. (SUC m) + n = n + (SUC m) *)
   ASM_REWRITE_TAC [ADD_CLAUSES])
```

Tactics and tacticals (e.g., `THEN` used in the example) are traditionally implemented using higher-order functions and exceptions in the meta language [22,17].

Trustworthiness. Trustworthiness of this approach relies on the typing discipline enforced by the meta language [6]. The data types for logical objects are encapsulated as *abstract types*, with their primitive operations implemented in a small, trusted module (called the *proof kernel*). Type abstraction ensures that all other code outside the kernel, no matter how complex, must ultimately construct theorems by calling the kernel's operations.

3 Core Interface

This section presents our core interface for HOL theorem proving in C.

3.1 Overview: A Complete Proof in C

To give a big picture of our core interface, we begin with a complete example using the core interface (Figure 1), which proves a theorem asserting that the right identity of the list-append operation is the empty list. The program includes `hol_prover_client.h` to gain access to the core interface types and functions, and connects to a proof server via `hol_prover_init()` at startup. It demonstrates several components of the core interface—definitional principles, inference rules, rewritings, parsing and printing. The reader need not understand every function used in the proof, but should note that the general structure of forward reasoning in C is the same as in ML-like languages (as shown in Section 2.2).

Theory extension. To introduce a new type for finite lists of natural numbers, it calls `new_datatype_definition` with the clauses of the inductive datatype definition as a string. It derives the induction and recursion theorems returned in the result structure `il`. It then uses `new_rec_definition` to define a recursive function `app` that appends two lists: it takes the list-recursion theorem `il.rec` and the defining equations as input, and returns the defining equation as a theorem.

Proving theorems. After the initial theory extensions, it proves the desired theorem, i.e., $\forall l : \mathit{nlist}.\ \mathit{app}\ l\ \mathit{Nil} = l$. The idea is simple: by induction on the list `l`. Theorems for the base case and the inductive step are established by rewriting using the defining equations for `app` and the induction hypothesis `ih`. After these are established, it specializes the induction theorem `il.ind` to match our goal, and then applies `mp_rule` (modus ponens) to obtain the final theorem.

```c
#include <stdio.h>
#include "hol_prover_client.h"

int main() {
  hol_prover_init();
  /* Step 1: define inductive type nlist */
  new_datatype_definition_results il =
      new_datatype_definition("nlist = Nil | Cons num nlist");
  /* Step 2: define recursive function app */
  thm app_def = new_rec_definition(
      il.rec, parse_term("app Nil l = l &&"
                         "app (Cons h t) l = Cons h (app t l)"));
  /* Step 3: prove !l. app l Nil = l by forward derivation */
  // Base case: |- app Nil Nil = Nil
  thm base_eq = apply_conversion(rewrite_conv(ThmList(app_def)),
                                 parse_term("app Nil Nil"));
  // Inductive step: |- !h t.app t Nil = t ==> app (Cons h t) Nil = Cons h t
  thm ih = assume_rule(parse_term("app t Nil = t"));
  thm step_eq = apply_conversion(rewrite_conv(ThmList(app_def, ih)),
                                 parse_term("app (Cons h t) Nil"));
  step_eq = gen_all_rule(disch_all_rule(step_eq));
  // Combine base and step cases and then apply the induction theorem
  thm ind_thm = beta_rule(spec_rule(parse_term("\\l. app l Nil = l"), il.ind));
  thm result = mp_rule(ind_thm, conj_rule(base_eq, step_eq));
  printf("result theorem: %s\n", string_of_thm(result)); // prints: |- !l. app l Nil = l
  return 0;
}
```

Fig. 1. A complete proof in C using the core interface.

3.2 Interface Description

The core interface adapts the common HOL API of ML-based provers [1] to C conventions. These form the building blocks for further proof programming in C, which will be demonstrated in Section 5.

Core types. The three kinds of logical objects in HOL—i.e., types, terms, and theorems—are represented as opaque types named `type`, `term`, and `thm`. Users create and inspect them exclusively through functions in the core interface.

Error handling. Error conditions during interface function calls are communicated through a status-code pattern: a function either succeeds and returns a valid object, or fails and (1) returns an empty value (constants `empty_type`, `empty_term`, `empty_thm` are defined for this purpose), (2) sets a global non-zero error code and an error message. Every function's documentation specifies the conditions for failure ("Fails if . . . ") or declares it total. For function calls that can fail, the caller should check the code after the call and handle the error appropriately (e.g., reset the code to zero and retry with an alternative proof function) or short-circuit to the upper level in the call stack. For concrete examples of error handling with a goto-based programming pattern, see Section 5.

Syntax Manipulation Recall (Section 2.1) that HOL types and terms have a simple structure: types are built from type variables and type constructors,

Table 1. The 10 primitive inference rules of higher-order logic.

C Function Signature	Brief Description (Simplified)
`refl_rule(t)`	Prove $\vdash t = t$ for any term t
`trans_rule(th1, th2)`	From $\vdash s = t$ and $\vdash t = u$, derive $\vdash s = u$
`mk_comb_rule(th1, th2)`	From $\vdash f = g$ and $\vdash x = y$, derive $\vdash f\ x = g\ y$
`abs_rule(x, th)`	From $\vdash s = t$, derive $\vdash \lambda x.\, s = \lambda x.\, t$
`beta_prim_rule(t)`	Prove $\vdash (\lambda x.\, s)\ x = s$
`assume_rule(p)`	Prove $p \vdash p$ for any proposition p
`eq_mp_rule(th1, th2)`	From $\vdash p \Leftrightarrow q$ and $\vdash p$, derive $\vdash q$
`deduct_antisym_rule(th1, th2)`	From $q \vdash p$ and $p \vdash q$, derive $\vdash p \Leftrightarrow q$
`inst_type_rule(ty_pairs, th)`	Instantiate type variables in a theorem
`inst_rule(tm_pairs, th)`	Instantiate free term variables in a theorem

and terms are built from variables, constants, applications, and lambda abstractions. The core interface systematically provides constructors (`mk_*`), destructors (`dest_*`), and discriminators (`is_*`) for these primitive syntactic categories. For example, below are the primitive term constructors:

```
term mk_var(const char* name, type ty);
term mk_const(const char* name, type ty);
term mk_comb(term fn, term arg);  /* fn arg  */
term mk_abs(term v, term body);   /* \v.body */
```

Constructors for standard data types (e.g., `mk_nat`), predicates (e.g., `mk_eq`), and logical connectives (e.g., `mk_forall`) are also provided as derived constructors. Other term manipulations include pattern matching `term_match` and capture-avoiding substitution `subst`. Functions `hyp(th)` and `concl(th)` extract the assumption terms and the conclusion term of a theorem.

Parsing and printing. It is convenient to construct and display terms using conventional mathematical notation. The core interface provides `parse_term`, `parse_type`, `string_of_term`, `string_of_type` to parse and print terms and types. The sequent representation of a theorem can be printed using `string_of_thm`.

Primitive and Derived Inference Rules The interface exposes the complete kernel interface to the deductive system of higher-order logic (the HOL Light version [11,10]), comprising 10 primitive inference rules. Table 1 lists them; essentially, they define the meaning of equality and functions. Together with suitable axioms, they suffice to derive all HOL theorems.

Beyond the primitive inference rules, the core interface exposes a rich set of derived rules.[6] These are defined on the server side but outside the proof kernel,

[6] We expose these derived rules in the core interface to reuse existing implementations from the proof server. Alternatively, they could be re-implemented on the client side, but we do not see clear benefits of doing so.

and thus not part of the trusted computing base (cf. Section 4). From a user's perspective, they are indistinguishable from the primitive ones.

Connective rules. All usual logical connectives are defined in terms of equality and functions (cf. Section 2.1). The interface provides derived introduction and elimination rules for each connective (e.g., `conj_rule` for conjunction introduction, `conjunct1_rule` and `conjunct2_rule` for conjunction elimination).

Rewriting and conversion. Rewriting is ubiquitous in theorem proving, especially in implementing tactics for simplification. The mechanism for rewriting in our interface is *conversions* [22], strategies mapping a term t to a theorem $\vdash t = t'$. Our interface exposes conversions as opaque `conversion` objects, applied via `apply_conversion(conv, tm)`. The built-in conversion `rewrite_conv` is used in the example proof in Figure 1: it takes a list of equational theorems and produces a conversion that rewrites a term using those equations.

Definitional Principles The logical theory can be extended conservatively by introducing new type and constant definitions (via `new_basic_definition` and `new_basic_type_definition`) or declarations (via `new_const` and `new_tyconst`). Users may also assert axioms via `new_axiom`, though they are responsible for justifying their validity. Information about the current theory can be queried by functions such as `get_all_axioms`, `get_all_definitions`, and `get_theorem_by_name`.

Atop these basic definition mechanisms, higher-level definitional principles are provided as derived rules: `new_fun_definition` for general recursive functions, `new_datatype_definition` for inductive datatypes, and `new_inductive_definition` for inductively defined relations. Each derives the characteristic theorem(s) from the input equational terms or definition clauses.

Proof-Search Procedures The interface provides automated provers for several domains, including `arith_rule` for linear arithmetic, `taut_rule` for propositional reasoning, and `meson_solver` for first-order reasoning. These provers take a term and helper lemmas and return a theorem when successful, or fail/loop if the goal cannot be proved. They are useful for finishing simple goals.

4 An Extended LCF Architecture

This section presents the architectural design that makes our core interface trustworthy under unconstrained proof programming in C, and briefly discusses our implementation.

4.1 Architectural Design

The challenge of trustworthiness. As reviewed in Section 2.2, the trustworthiness of the LCF approach rests on type abstraction provided by the meta language.

Specifically, the theorem type is abstract, so only the kernel may construct its values. This ensures that every theorem is necessarily derived from valid applications of the primitive inference rules, a property known as *full expansiveness* [8].

It is immediate that C *does not qualify* as the meta language for the LCF approach: by its low-level nature, C's type system is permissive of pointer arithmetic and type casting; this allows user code to circumvent the kernel interface and forge theorem values by direct memory manipulation. For instance, knowing the concrete layout of theorems in a kernel implementation (e.g., a pointer to a struct with two fields that store the hypothesis list and the conclusion term), a malicious user could mutate the conclusion to fabricate an invalid theorem:

```
struct layout { term* hyps; term concl; };
thm taut = assume_rule(mk_true());          // true |- true
((struct layout*)taut)->concl = mk_false(); // true |- false, unsound!
```

Design: A Client-Server Architecture. Our key observation is that the implementation language of the kernel and the user-facing language for proof programming need not be the same, as long as they can interoperate via a core interface with guards against invalid operations. This deviates from the use of a single meta language for both in the traditional LCF approach. We achieve this through a client-server model (Figure 2) that extends the LCF architecture, where user code runs in a client process and an LCF prover runs in a server.

The client. On the client side, a client-interface library implements and exposes the core-interface API to user code. The methods are implemented as remote procedure calls (RPCs) to the server. Importantly, logical objects always live in the server's memory: methods in the core interface manipulate them indirectly through opaque *handles* generated by the server. Because the client has no direct access to the server's memory, arbitrary user proof code—even code written in an unsafe language like C—cannot forge theorem values.

The server. The server has two main components: a traditional LCF prover and a service handler that manages the interaction between the client and the prover. As required by the LCF approach, the prover is implemented in a language with type abstraction. A *service handler* sits above the LCF prover as a thin wrapper: it maintains the current active logical objects during a proof session in an object store, validates incoming RPC requests, and dispatches them to the prover. The object store associates handles with logical objects: a handle uniquely identifies a logical object in the object store, allowing the client to pass the object around without direct access to its concrete memory representation.

Communication example. We illustrate the client-server interaction with a concrete example: the user code calls `mp_rule(hdl1, hdl2)`, where `hdl1` and `hdl2` are opaque handles to two theorems. The communication proceeds as follows:

1. *Client* → *Server.* The client-side `mp_rule` function sends an RPC request carrying `hdl1` and `hdl2` to the server.

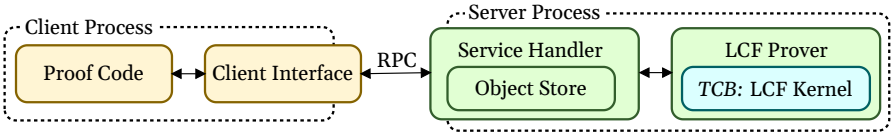

Fig. 2. The extended LCF architecture with a client-server design. The architecture maintains the full expansiveness property of the LCF approach while allowing users to program proofs and derived facilities in an arbitrary language of choice (such as C).

2. *Handle resolution.* The service handler looks up the actual theorem objects `thm1` and `thm2` from the object store using the provided handles. If either handle is invalid, the handler immediately signals an error to the client.
3. *Proof computation.* The service handler delegates to the LCF prover by calling the corresponding inference rule, e.g., `MP(thm1, thm2)`. Implementation of `MP` ultimately invokes the primitive inference rules in the kernel, e.g., `EQ_MP`. If any inference fails (e.g., `thm1` is not an implication), the handler signals an error to the client.
4. *Server → Client.* On success, the handler generates a fresh handle for the resulting theorem, stores the association in the object store, and returns the handle to the client for use in subsequent proof steps.

Trustworthiness guarantee. Our design relies on the assumption that the operating system provides reliable process isolation: user-space processes cannot directly access the memory of other processes. Under this assumption, the architecture ensures that the trusted computing base (TCB) consists solely of the proof kernel. All client-side code (including user-written proof code and the client interface library) is excluded from the TCB by virtue of memory isolation; server-side code beyond the kernel (e.g., the service handler and the derived libraries) is also excluded from the TCB, as guaranteed by type abstraction.

Note that we do not address broader security properties such as access control; for example, a client may forge handles to access theorems created by another user, but this does not compromise soundness, as the forged handles still refer to valid theorems constructed by the kernel.

4.2 Implementation Notes

TCB size. We implement this architecture in a C client and an OCaml server. For the server side, we choose to reuse the HOL Light theorem prover [11] as the supporting LCF prover. Its kernel is approximately 600 lines of OCaml code that have been widely used and scrutinized by the community.

RPC implementation. Cross-language RPC between the C client and the OCaml server is handled by Cap'n Proto [27], chosen for its multi-language support. The core interface is defined in Cap'n Proto's schema language, allowing generation

of stub code. Handles to logical objects are represented as byte strings, which are the actual payload of the RPC requests. Overall, the client interface comprises ∼7000 lines of C/C++ (mostly auto-generated from the schema) and the service handler comprises ∼4000 lines of OCaml. This shows that our approach is relatively lightweight for supporting proof programming in an arbitrary language of choice.

Memory management. To avoid manual memory management during proof programming, we employ the Boehm-Demers-Weiser conservative garbage collector [3]: when a handle is reclaimed on the client side, the client interface notifies the server to remove the corresponding object from its store, allowing the OCaml runtime to garbage-collect it as well.

5 Programming Derived Proof Facilities in C

The LCF approach allows users to program additional proof facilities in the meta language as derived libraries atop the kernel. This section demonstrates that *goal-directed reasoning* [17]—the prime example of such facilities, and an integral part of every LCF-style prover—can be programmed idiomatically in C upon our core interface. We first review the traditional ML-based approach to goal-directed reasoning, which relies on higher-order functional programming. We then present our alternative design based on an explicit goal-tree data structure.

5.1 Goal-Directed Reasoning in the LCF Tradition

In LCF-style provers, goal-directed reasoning is expressed in terms of *tactics* that implement basic backward proof steps. These tactics can be composed into full proof strategies by *tacticals*. A *subgoaling package* manages the proof state across individual tactic applications for interactive proof discovery [23].

Tactics and validation closures. A *tactic* is an ML function that, given a goal (a sequent to be proved), returns a list of subgoals together with a *validation*—a function that should derive a theorem achieving the goal once theorems for the subgoals are provided. Concretely, the relevant ML types are:

```
type goal = term list * term;
type validation = thm list -> thm;
type tactic = goal -> goal list * validation;
```

For example, read from bottom to top, the following rule for implication introduction (parameterized by any formula p) can be used in goal-directed reasoning to move the antecedent of an implicational goal into the assumptions:

$$\frac{\Gamma \vdash q}{\Gamma \setminus \{p\} \vdash p \Longrightarrow q} \quad \textsc{Disch}(p)$$

This can be implemented in ML as a tactic `DISCH_TAC`:

```
val DISCH_TAC : tactic = fn (asl, conclu) =>
    let val (antece, conseq) = dest_imp conclu (* decomposition *)
    in  ([(antece :: asl, conseq)],   (* singleton subgoal list *)
         fn [th] => DISCH antece th)      (* validation closure *)
    end handle _ => raise Fail "DISCH_TAC"    (* error handling *)
```

Note the validator captures the antecedent `antece` from the environment to create a *closure*. Once the subgoals are solved, the validator can be invoked with the list of theorems for the subgoals (singleton here) to derive the original theorem.

Tacticals and composite validators. Most problems cannot be solved by a one-step tactic. *Tacticals* such as `THEN` (sequential composition), `ORELSE` (choice), and `REPEAT` (iteration) are combinators that allow composing tactics into more powerful strategies. (e.g., until they fully solve the problem at hand; cf. the example in Section 2.2) Crucially, implementing these tacticals requires assembling individual validators into increasingly larger composite validators by *higher-order functional programming*. For instance, the following ML code implements the `THEN` tactical that applies two tactics sequentially:

```
fun (tac1:tactic) THEN (tac2:tactic) : tactic = fn g =>
  let val (gl,p) = tac1 g
      val (gll,pl) = unzip(map tac2 gl)
  in (flatten gll, (p o mapshape(map length gll) pl)) end;
```

It applies `tac1` to obtain a subgoal list `gl` and a validator `p`, then applies the second tactic `tac2` to each subgoal to obtain a list of subgoal lists `gll` and a list of validators `pl`. After this decomposition phase, it constructs a new validator that chains all of these: the combinator-style code corresponds to first invoking each `pl[i]` on the theorems for the subgoals `gll[i]`, then passing the resulting child theorems to `p` to derive the theorem for the original goal.

Tacticals like `ORELSE` and `REPEAT` additionally rely on *exceptions* for backtracking on the proof state: if a tactic fails (by raising an exception), the tactical catches the exception and either tries an alternative or terminates the iteration.

The subgoal package. For proof discovery, users typically make progress by applying small tactics in sequence. To relieve users from bookkeeping of subgoals and calling validators manually, a *subgoal package* manages the proof state across individual tactic applications [23]. It maintains two components: (a) all current unresolved subgoals, and (b) a monolithic validator that incorporates the full chain of composed validators linking the subgoals back to the original goal. Each time the user applies a tactic to one of the open subgoals, the subgoal package updates this monolithic validation by composing the new tactic's validation into the existing chain, similar to the above `THEN` tactical.

5.2 Goal-Directed Reasoning in C

The traditional approach to goal-directed reasoning fundamentally relies on higher-order functional programming (for defining tactics and composing val-

```
typedef thm (*validator)(thm* thl, goal g, void *env);

typedef struct goal_node {
  term *asmps, concl;          // goal sequent
  thm solved;                  // theorem proving the goal once solved; empty otherwise

  int num_subgoals;            // number of subgoal nodes
  struct goal_node **subgoals; // array of pointers to subgoal nodes

  validator valid;             // pointer to a validator function
  void *env;                   // custom data for validator
} *GN;                         // GN is short for Goal Node
```

Fig. 3. The `goal_node` structure representing a single node in the goal tree.

idators), and exceptions (for backtracking in tacticals). C provides none of these natively, motivating the alternative design described next.

Reified Proof Structure as a Goal Tree Our idea is simple: underlying every goal-directed proof is a tree of goals, where each internal node has been decomposed into subgoals represented by its children; the unsolved leaves constitute the remaining proof obligations.

Instead of making this tree implicit in the composition of validators by higher-order programming, we materialize it as an explicit data structure. The information stored in each node is shown in Figure 3, including fields for its goal, its child nodes, a validator closure (comprising a function pointer with a user-specified environment), and the theorem to be filled in during proof reconstruction.

Figure 4 schematically presents a (partial) goal tree for proving a goal with conclusion $A \wedge (B \to C)$. The tree results from first applying conjunction introduction to create two nodes with conclusions A and $B \to C$, and then applying an implication introduction tactic to the second node to create a third node with conclusion C. The root node stores a validator `conj_rule(_,_)` that, given theorems for A and $B \to C$, will produce a theorem for $A \wedge (B \to C)$. Its right child stores a validator `disch_rule(B,_)` that, given a theorem for C proved under the extra assumption B, will reconstruct the implication $B \to C$ with the assumption B discharged.

The goal tree serves dual purposes, i.e., subgoaling and validation chaining, *without the need for higher-order programming*. Specifically, for subgoaling, it provides a function `goal_node_get_leaves(gn)` that fetches the unsolved leaves from a node `gn`; for validation chaining, it provides a function `goal_node_prove(gn)` that traverses the tree bottom-up, checks the leaves are solved, and calls each node's validator with the list of theorems for its children. Once this traversal is complete, the final theorem should be available in the node `gn`.

Tactics as Tree-Growing Transformations A tactic in our system is a *tree-growing transformation*: it takes a goal node (usually an unresolved leaf node) and expands it by creating child nodes that represent the decomposed subgoals,

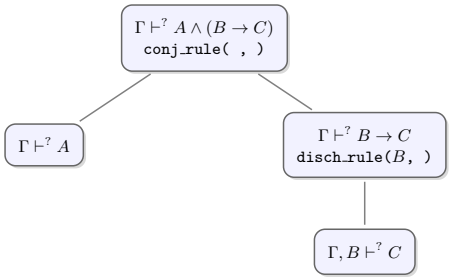

Fig. 4. A schematic partial goal tree for proving $\Gamma \vdash^? A \wedge (B \to C)$. The question mark in $\vdash^?$ is used to indicate that the goal is not yet solved.

storing a validator and its environment in the parent node. A tactic either succeeds (returning the new subgoal nodes) or fails by setting a global error status. As an example, the tactic of implication introduction is implemented as follows:

```
thm DISCH_TAC_VALID(thm_list thl, goal gl, void *antece) {
  return disch_rule(*(term *)antece, thl[0]);
}
GN DISCH_TAC(GN gn) {
  term_list asl = goal_node_get_asmps(gn);
  term conclu = goal_node_get_concl(gn);
  ENSURE_COND(is_imp(conclu), "the conclusion is not an implication"); // error checking
  dest_imp_results dir = dest_imp(conclu); // goal decomposition
  term antece = dir.tm1, conseq = dir.tm2;
  goal_node_set_the_subgoal(gn, terms_append(asl, antece), conseq); // set subgoal nodes
  goal_node_set_validator(
      gn, DISCH_TAC_VALID,      // set validator
      (void *)wrap_term(antece) // allocate memory for the term on heap
  );
  return goal_node_get_the_subgoal(gn); // return the new subgoal node
err: ... /* error handling */
}
```

We have implemented other common HOL tactics following the same pattern: they modify the goal tree, store a validator (plus environment) on the parent node, and return the new subgoal nodes.

Tacticals via C Control Flow With an explicit goal tree, composite proof strategies (e.g., `THEN`, `ORELSE`, `REPEAT`) can be effectively realized by composing tactics using standard C control flow—loops, conditionals, and `goto` statements. We illustrate the patterns for sequential composition and choice below.

Sequential composition. We achieve sequential composition by iterating over the list of resulting subgoal nodes:

```
GN gn = GOAL_INIT(parse_term("(A ==> B) /\\ (C ==> D)"));
GN *subgoals = CONJ_TAC(gn);   // the first tactic expands into two subgoal nodes
for (int i = 0; i < 2; i++)
    SOME_TAC(subgoals[i]);    // apply the second tactic to each subgoal node
GN *new_subgoals = goal_node_get_leaves(gn); // collect the unsolved leaves
```

```
#include <stdio.h>
#include "hol_prover_client.h"
#include "proof_backward.h" // a library of tactics

int main() {
  hol_prover_init();
  GN root = GOAL_INIT(parse_term("forall m n:num. m + n = n + m"));
  GN* subgoals = NUM_INDUCT_TAC(root); // apply the induction tactic
  for (int i = 0; i < 2; i++)          // apply the rewrite tactic to each subgoal
    subgoals[i] = ASM_REWRITE_TAC(subgoals[i],
                                  ThmList(get_theorem_by_name("ADD_CLAUSES")));
  thm result = goal_node_prove(root);  // validate the tree and get the final theorem
  printf("result: %s\n\n", string_of_thm(result));
  return 0;
}
```

Fig. 5. A complete goal-directed proof in C.

Choice. To implement try-then-else, we attempt the first tactic; if it fails (indicated by error status), we proceed with the second tactic on the same node:

```
GN try_discharge_else_gen(GN gn) {
  GN new_gn = DISCH_TAC(gn);   // attempt using DISCH_TAC
  CHECK_OK_OR_GOTO(after_try); // goto after_try if failed
  return new_gn;               // success, return the new goal node
after_try:
  new_gn = GEN_TAC(gn);        // using GEN_TAC
  ENSURE_OK("both failed");    // ensure success or goto error handling
  return new_gn;               // success, return the new goal node
err: ... /* error handling */
}
```

When the error status is not zero, the macro `CHECK_OK_OR_GOTO(lab)` resets the error code and jumps to `lab`; `ENSURE_OK(msg)` sets the error message to `msg` and jumps to the default error handling label `err`.

A Complete Goal-Directed Proof Figure 5 shows a complete goal-directed proof in C using our goal-tree based tactics (compare with the ML proof script in Section 2.2). It begins with `GOAL_INIT`, which creates a root node from a term. It then applies the tactic `NUM_INDUCT_TAC` to generate two subgoal nodes, and applies `ASM_REWRITE_TAC` to each subgoal to solve them by rewriting. Finally, it validates the tree and returns the final theorem.

6 Related Work

LCF-style theorem proving. The LCF approach [6,7] has been the foundation of a long line of theorem proving systems, especially those in the HOL family [23,11,26]. All of these use an ML-like typed higher-order language as both the implementation language and the user-facing proof language. Our work extends the LCF approach to C: we decouple the kernel implementation language from the proof programming language, ensuring trustworthiness through process isolation while enabling programmers to write proofs and build derived facilities naturally in C.

Supervisionary. Mulligan [20] proposes *Supervisionary*, a proof-checking system for HOL in which the trusted kernel manages logical objects in private heaps, runs at a higher privilege level than untrusted automation, and communicates with user-space through a low-level system-call interface using opaque handles. Our work shares the insight that the proof kernel can be protected by machine-level isolation rather than programming-language mechanisms. In contrast to Supervisionary's approach of building a custom WebAssembly host, our client-server design based on RPC provides a lighter-weight realization that reuses an existing LCF prover. Moreover, Supervisionary does not touch on practical proof programming; our work further demonstrates that goal-directed proof facilities such as tactics can be programmed idiomatically in C.

The Common HOL Platform. Adams [1] defines a standard API for HOL theorem provers, covering syntax utilities, inference rules, parsing and printing, and theory extension commands, with the goal of facilitating source-code and proof portability across the HOL family. Our core interface is partly inspired by this platform: we follow its categorization of components and its design principles of coherence and completeness, while adapting the conventions to C (e.g., replacing ML exceptions with status codes, replacing tuples with monomorphized result structures, and exposing conversions as opaque objects composed via a first-order interface rather than higher-order combinators).

7 Conclusion and Future Work

In this work, we adapt the LCF approach to theorem proving for C, maintaining a small trusted computing base and extending proof programmability to low-level languages without strong type safety guarantees.

One direction for future work is how to support an incremental proof-running experience: currently, proof code is compiled and executed in its entirety, which hinders interactive proof exploration. We also envision this work as a step towards *in-situ* program verification, where implementation code and its correctness proof coexist in the same program. By incorporating a proof-writing interface directly in C, as demonstrated in this paper, it is possible for developers to carry out program-reasoning steps within the same language and toolchain they already use. We leave the design of such a verifier as future work.

Acknowledgments. We thank the anonymous reviewers for their helpful comments; Zixun Guo for his contributions to the implementation; Houjin Chen, Wenbo Xu, Zhiyi Wang, Taofei Guo, Wenkai Xing, Yi Fang, Chengtao Li, Bingjie Duan and Xi He for being the first users of our library; and Qinxiang Cao and Haiyan Zhao for their long-term discussions.

This research was partly supported by the National Science and Technology Major Project of the Ministry of Science and Technology of China under Grant No. 2025ZD1503300.

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

References

1. Adams, M.: The common HOL platform. In: Kaliszyk, C., Paskevich, A. (eds.) Proceedings Fourth Workshop on Proof eXchange for Theorem Proving, PxTP 2015, Berlin, Germany, August 2-3, 2015. EPTCS, vol. 186, pp. 42–56 (2015). https://doi.org/10.4204/EPTCS.186.6, https://doi.org/10.4204/EPTCS.186.6
2. Appel, A.W.: Verification of a cryptographic primitive: SHA-256. ACM Trans. Program. Lang. Syst. **37**(2), 7:1–7:31 (2015). https://doi.org/10.1145/2701415, https://doi.org/10.1145/2701415
3. Boehm, H.J.: Space efficient conservative garbage collection. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation. pp. 197–206. PLDI '93, ACM, Albuquerque New Mexico USA (Jun 1993). https://doi.org/10.1145/155090.155109
4. Cao, Y., Zhuang, J., Fan, J., Wang, D., Hu, Z.: Artifact for "A HOL Theorem Proving Interface for C" (Mar 2026). https://doi.org/10.5281/zenodo.18887462, https://doi.org/10.5281/zenodo.18887462
5. Delahaye, D.: A tactic language for the system coq. In: Parigot, M., Voronkov, A. (eds.) 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, pp. 85–95. Springer (2000). https://doi.org/10.1007/3-540-44404-1_7, https://doi.org/10.1007/3-540-44404-1_7
6. Gordon, M.J.C., Milner, R., Morris, F.L., Newey, M.C., Wadsworth, C.P.: A metalanguage for interactive proof in LCF. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978. pp. 119–130. ACM Press (1978). https://doi.org/10.1145/512760.512773, https://doi.org/10.1145/512760.512773
7. Gordon, M.J.C., Milner, R., Wadsworth, C.P.: Edinburgh LCF, Lecture Notes in Computer Science, vol. 78. Springer (1979). https://doi.org/10.1007/3-540-09724-4, https://doi.org/10.1007/3-540-09724-4
8. Gordon, M.: From LCF to HOL: a short history. In: Plotkin, G.D., Stirling, C., Tofte, M. (eds.) Proof, Language, and Interaction, Essays in Honour of Robin Milner. pp. 169–186. The MIT Press (2000)
9. Gu, R., Shao, Z., Chen, H., Wu, X.N., Kim, J., Sjöberg, V., Costanzo, D.: CertiKOS: An extensible architecture for building certified concurrent os kernels. In: Keeton, K., Roscoe, T. (eds.) 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016. pp. 653–669. USENIX Association (2016), https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu
10. Hales, T.C.: Formal proof. Notices of the AMS **55**(11), 1370–1380 (2008)
11. Harrison, J.: HOL light: An overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) 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, pp. 60–66. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_4, https://doi.org/10.1007/978-3-642-03359-9_4
12. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D.A., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an os kernel. In: Matthews, J.N.,

Anderson, T.E. (eds.) Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009. pp. 207–220. ACM (2009). https://doi.org/10.1145/1629575.1629596, https://doi.org/10.1145/1629575.1629596

13. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 179–192. ACM (2014). https://doi.org/10.1145/2535838.2535841, https://doi.org/10.1145/2535838.2535841
14. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009). https://doi.org/10.1145/1538788.1538814, https://doi.org/10.1145/1538788.1538814
15. Martin-Löf, P.: Constructive mathematics and computer programming. In: Cohen, L.J., Łoś, J., Pfeiffer, H., Podewski, K.P. (eds.) Logic, Methodology and Philosophy of Science VI, Studies in Logic and the Foundations of Mathematics, vol. 104, pp. 153–175. Elsevier (1982). https://doi.org/https://doi.org/10.1016/S0049-237X(09)70189-2, https://www.sciencedirect.com/science/article/pii/S0049237X09701892
16. Matichuk, D., Murray, T.C., Wenzel, M.: Eisbach: A proof method language for isabelle. J. Autom. Reason. **56**(3), 261–282 (2016). https://doi.org/10.1007/S10817-015-9360-2, https://doi.org/10.1007/s10817-015-9360-2
17. Milner, R.: The use of machines to assist in rigorous proof. Philosophical Transactions of the Royal Society of London, Series A: Mathematical and Physical Sciences **312**(1522), 411–422 (10 1984). https://doi.org/10.1098/rsta.1984.0067, https://doi.org/10.1098/rsta.1984.0067
18. de Moura, L., Ullrich, S.: The lean 4 theorem prover and programming language. In: Platzer, A., Sutcliffe, G. (eds.) Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12699, pp. 625–635. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_37, https://doi.org/10.1007/978-3-030-79876-5_37
19. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24, https://doi.org/10.1007/978-3-540-78800-3_24
20. Mulligan, D.P.: All watched over by machines of loving grace. In: Kesner, D., Pédrot, P. (eds.) 28th International Conference on Types for Proofs and Programs, TYPES 2022, LS2N, University of Nantes, France, June 20-25, 2022. LIPIcs, vol. 269, pp. 1:1–1:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPICS.TYPES.2022.1, https://doi.org/10.4230/LIPIcs.TYPES.2022.1
21. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002). https://doi.org/10.1007/3-540-45949-9, https://doi.org/10.1007/3-540-45949-9
22. Paulson, L.C.: A higher-order implementation of rewriting. Sci. Comput. Program. **3**(2), 119–149 (1983). https://doi.org/10.1016/0167-6423(83)90008-4, https://doi.org/10.1016/0167-6423(83)90008-4

23. Paulson, L.C.: Logic and computation - interactive proof with Cambridge LCF, Cambridge tracts in theoretical computer science, vol. 2. Cambridge University Press (1987)
24. Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: Wexelblat, R.L. (ed.) Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988. pp. 199–208. ACM (1988). https://doi.org/10.1145/53990.54010, https://doi.org/10.1145/53990.54010
25. Pitts, A.M.: The HOL System: Logic, Trindemossen-2 Edition (2025), https://github.com/HOL-Theorem-Prover/HOL/releases/download/trindemossen-2/trindemossen-2-logic.pdf
26. Slind, K., Norrish, M.: A brief overview of HOL4. In: Mohamed, O.A., Muñoz, C.A., Tahar, S. (eds.) 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, pp. 28–32. Springer (2008). https://doi.org/10.1007/978-3-540-71067-7_6, https://doi.org/10.1007/978-3-540-71067-7_6
27. Varda, K.: Cap'n proto. https://capnproto.org/
28. Wenzel, M.: Isar - A generic interpretative approach to readable formal proof documents. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin-Mohring, C., Théry, L. (eds.) Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1690, pp. 167–184. Springer (1999). https://doi.org/10.1007/3-540-48256-3_12, https://doi.org/10.1007/3-540-48256-3_12
29. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: HACL*: A verified modern cryptographic library. In: Thuraisingham, B., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 1789–1806. ACM (2017). https://doi.org/10.1145/3133956.3134043, https://doi.org/10.1145/3133956.3134043