Flex: A Foundational Proof-Producing Horn Solver

JAM KABEER ALI KHAN, Independent, Hong Kong SAR
PETROS MARKOPOULOS, University of California San Diego, USA
NIKI VAZOU, IMDEA Software Institute, Spain
NICO LEHMANN, University of Chile, Chile
RANJIT JHALA, University of California San Diego, USA

Constrained Horn Clauses (CHCs) are a versatile intermediate representation for program verification, but leading CHC solvers rely on SMT-based reasoning over supported background theories. Specifications involving user-defined recursive functions or bespoke operators can require additional semantic encodings and reasoning beyond these theories. We observe that CHCs are existentially quantified propositions in a foundational logic, and so can be proved within an interactive theorem prover, where specifications can use native definitions and users can complete proofs when automation stalls. However, in a foundational setting, every Horn variable solution must be justified by a kernel-checked proof. We present Flex, a proof-producing CHC solver implemented as a Lean metaprogram, that combines predicate synthesis with proof construction. Flex comprises a suite of tactics: Zap, which synthesizes strongest predicates for acyclic Horn variables and exploits their structure to construct elimination proofs without any proof search; Fix, which uses predicate abstraction over templates built from arbitrary Lean terms, discovered by *scraping* the CHCs and associated definitions, to infer inductive invariants for cyclic Horn variables; and a certifying pipeline that keeps external SMT-based solvers like Spacer and Eldarica outside the trusted computing base. We use Flex to verify the functional correctness of sorting algorithms, hash tables, ring buffers from the Tock OS kernel, and Rust `std` library challenges. We show that Zap constructs elimination proofs 28.2× and 44.6× faster than `grind` and `aesop`, respectively. With Zap and Fix, Flex automatically proves 98.0% of the refinement-typechecker CHCs in our evaluation.

Additional Key Words and Phrases: Constrained Horn Clauses, Horn Solving, Lean 4, Interactive Theorem Proving, Refinement Types, Invariant Inference, Predicate Abstraction, Proof-Producing Automation

1 Introduction

Constrained Horn Clauses [5, 22, 50] have proven to be a versatile intermediate representation for encoding program verification conditions. Horn variables can be used to represent the unknown invariants or intermediate assertions, required to prove some verification goal, after which SMT-based algorithms like Spacer's Property Directed Reachability [36] or Eldarica's Lazy Predicate Abstraction [33] can be used to solve these CHCs to automate verification. Their versatility and automation has made CHCs the backend of choice for a variety of verifiers based on program logics for C/C++ [28], or refinement types for Rust [38, 42, 49] and Haskell [57].

However, leading CHC solvers rely on SMT-based reasoning over supported background theories. Specifications involving recursive functions or bespoke operators can require additional semantic encodings and reasoning beyond these theories. For example, consider the verification of a Rust function in Fig. 1 that computes the number of bytes needed to encode a character sequence `cs`. To *specify* the correctness of the result, we require a recursive user-defined function $utf8_len(s, k)$ that computes the UTF-8 size of the first k characters of the sequence s. To then *verify* the function, we require first, a loop invariant that states that the accumulator tracks the size of the prefix of size `i`, and second, a separate monotonicity lemma that establishes the arithmetic does not overflow.

Authors' Contact Information: Jam Kabeer Ali Khan, Independent, Hong Kong SAR, jamkhan@connect.hku.hk; Petros Markopoulos, University of California San Diego, San Diego, USA, pmarkopoulos@ucsd.edu; Niki Vazou, IMDEA Software Institute, Madrid, Spain, niki.vazou@software.imdea.org; Nico Lehmann, University of Chile, Santiago, Chile, nlehmann@dcc.uchile.cl; Ranjit Jhala, University of California San Diego, San Diego, USA, rjhala@ucsd.edu.

Sadly, these obligations can exceed the reasoning supported by SMT-based CHC solvers, leaving programmers to supply explicit loop invariants and proof blocks that clutter the source code.

One way to overcome the expressiveness limitation is to observe that CHCs are *existentially quantified propositions* in a foundational logic, such as that of Lean [45] or Rocq [55] or Isabelle [47]. Thus, *in principle* we might use an ITP to prove the CHC, instead of restricting ourselves to SMT solving. This would allow specifications to refer to native ITP definitions, and programmers can inspect the CHCs and intermediate proof states to complete their proofs when automation stalls.

Challenge: Proof-producing CHC Solvers Alas, merely translating the CHCs into ITP propositions is not terribly useful as the user is still left with the task of solving the Horn variables to prove the CHC. Thus our first challenge is to *solve* the Horn variables within the ITP to be able to instantiate them with predicates that render the CHC valid. However, the foundational setting poses a second challenge: unlike in the SMT-based setting where the solvers can report predicate solutions that the verifier simply trusts, in a foundational context, any solution whose substitution eliminates a Horn variable from the constraint must be justified by a kernel-checked proof. That is, any solver must be *proof-producing*, in that it must construct a kernel-checkable certificate witnessing that the validity of the proposition obtained by eliminating the Horn variable *implies* the validity of the original CHC. Since efficient solving may analyze sub-constraints locally, certification must be aligned with those local analyses and compose their proofs across eliminations.

Flex We present Flex, a foundational CHC solver implemented as a Lean metaprogram [45] that combines predicate synthesis with proof construction to address the challenge of proof-producing CHC solving. Following [14] we observe that the Horn variables in a constraint can be partitioned into acyclic ones, that can be solved exactly, and cyclic ones that require some form of fixpoint over-approximation. Thus, we design Flex as a suite of proof-producing tactics that separately solve the acyclic and cyclic variables, yielding the following contributions.

1. *Proof-producing* Acyclic Horn Solving (§4). Our first contribution is Zap, a proof-producing algorithm that synthesizes strongest predicates for acyclic Horn variables and constructs proofs of their elimination. We identify a crucial correspondence between the structure of constraints, the synthesized predicates, and the elimination proofs that Zap exploits to construct elimination witnesses deterministically and efficiently, without requiring any proof search. The construction preserves variable scopes and bindings, and composes individual eliminations into a certificate that the residual constraint implies the original.

2. Predicate Abstraction over Lean Functions (§5). Our second contribution is Fix, a certifying predicate-abstraction procedure [27] that infers inductive invariants for cyclic Horn variables. Following liquid inference [23, 51], Fix weakens conjunctions of predicate-template instances until it reaches a fixpoint. Unlike traditional SMT-based approaches, Fix operates directly within Lean's expressive logic and hence can use templates over arbitrary Lean terms, crucially enabling synthesis of invariants over user-defined functions or quantifiers. To discover qualifiers, we develop *scraping* algorithms that traverse the CHCs and associated Lean definitions to extract candidate invariant fragments from equalities or predicates in branch conditions (§6), to enable Fix to synthesize invariants beyond the theories supported by current SMT-based solvers.

3. Certifying External CHC Solvers (§6). Our third contribution is to show how to integrate the existing state-of-the-art SMT-based solvers — Spacer and Eldarica — through a certifying pipeline that translates Lean CHCs into SMT-LIB, reconstructs inferred invariants as Lean predicates, and certifies their elimination with respect to the original constraints. The obligations that remain after eliminating the Horn variables remain as ordinary Lean goals, allowing partial progress via the CHC oracles while keeping external solvers outside the trusted computing base.

```
fn encoded_len(cs: &[char][n,m])
-> u32[utf8_len(m,n)]
requires utf8_len(m,n) <= u32::MAX
{
  let mut acc = 0;
  let mut i = 0;
  while i < cs.len() {
    let c = cs[i] as u32;
    let w = if c < 128 { 1 }
      else if c < 2048 { 2 }
      else if c < 65536 { 3 }
      else { 4 };
    acc += w;
    i += 1;
  }
  acc
}
```

$$
\begin{aligned}
c_{top} \doteq\ & \exists \kappa_{\text{inv}}, \kappa_{\text{w}}. \forall m, n. \\
& \mathit{utf8_len}(m, n) \leq \text{MAX} \rightarrow \\
& \quad \kappa_{\text{inv}}(0, 0, m, n) \\
& \quad \wedge \forall \mathit{acc}, i.\ \kappa_{\text{inv}}(\mathit{acc}, i, m, n) \rightarrow \\
& \qquad (n \leq i \rightarrow \mathit{acc} = \mathit{utf8_len}(m, n)) \\
& \qquad \wedge (i < n \rightarrow c_{body})
\end{aligned}
$$

$$
\begin{aligned}
c_{body} \doteq\ & (m[i] < 128 \rightarrow \kappa_{\text{w}}(1, \mathit{acc}, i, m, n)) \\
& \wedge (128 \leq m[i] < 2048 \rightarrow \kappa_{\text{w}}(2, \mathit{acc}, i, m, n)) \\
& \wedge (2048 \leq m[i] < 65536 \rightarrow \kappa_{\text{w}}(3, \mathit{acc}, i, m, n)) \\
& \wedge (65536 \leq m[i] \rightarrow \kappa_{\text{w}}(4, \mathit{acc}, i, m, n)) \\
& \wedge c_{join}
\end{aligned}
$$

$$
\begin{aligned}
c_{join} \doteq\ & \forall w.\ \kappa_{\text{w}}(w, \mathit{acc}, i, m, n) \rightarrow \\
& \quad \mathit{acc} + w \leq \text{MAX} \\
& \quad \wedge \kappa_{\text{inv}}(\mathit{acc} + w, i + 1, m, n)
\end{aligned}
$$

Fig. 1. (L) Rust with Flux specification, (R) verification CHC generated by Flux.

4. Implementation and Evaluation (§7). Our final contribution is an implementation of Flex as a Lean tactic in approximately 9,200 lines, and an empirical evaluation of Flex on a variety of case studies.[1] First, we show that Flex' expressiveness enables the verification of functional correctness in a suite of case studies covering implementations of sorting algorithms, Hash tables, ring buffers from the Tock OS kernel, and Rust `std` library verification challenges including the correctness of string matching, string encoding and the correctness of the `vec-deque` module (§ 7.1). Second, we show that on a suite of existing, SMT-checkable, refinement typing and CHC-COMP benchmarks, Zap constructs elimination proofs 28–45× faster than `grind` and `aesop` on jointly reduced constraints (§ 7.2). Third, we demonstrate 98.0% coverage over refinement typing CHCs (§ 7.3). Finally, on selected cyclic CHCs, we show that incremental scraping improves proof coverage while controlling candidate growth and solving cost (§ 7.4).

2 An Overview of Flex

We begin with an overview, starting with the motivation behind Flex: solving CHCs in Lean generated by the Rust refinement type checker Flux (§ 2.1). Next, we show how predicate synthesis guides proof-producing elimination of acyclic Horn variables (§ 2.2). Finally, we describe cyclic invariant inference using native Lean predicates and certifying external oracles (§ 2.3).

2.1 Motivating Example

Fig. 1 shows a Rust function `encoded_len` that computes the *number of bytes*, represented as a `u32`, *i.e.* unsigned 32-bit integers, needed to encode `cs`, a slice of Unicode characters. The function iterates over the slice, accumulating the sum of the width of each character's encoding in `acc`.

Specification: Refinement Types. To *specify* the correctness requirement, the programmer writes a Flux refinement type contract where the input type `&[char][n, m]` represents the slice by its length `n` and contents `m` which is a map from slice-indices to the corresponding character. The specification uses a programmer-defined Lean function $\mathit{utf8_len}(m, k)$ denoting the number of

[1] Flex is available at https://github.com/LeanHorn/Flex; benchmarks are available at https://github.com/LeanHorn/flex-benchmarks.

```
theorem encoded_len_proof : EncodedLenCHC
  := by unfold EncodedLenCHC; zap ; fix
        have := utf8_len_mono m (i+1) n
          (by omega)
        grind [cw, utf8_len_succ]
```

```
theorem utf8_len_mono: ∀ m i j,
  i ≤ j → utf8_len m i ≤ utf8_len m j
  := by
  -- induction on j - i; each step
  -- ... adds cw (m k) ≥ 0
```

Fig. 2. Verifying `encoded_len` in Lean with an inductive lemma about *utf8_len*.

bytes needed to encode the first k elements of m, defined by recursion on k, *i.e.* $utf8_len(m, 0) \doteq 0$ and $utf8_len(m, k+1) \doteq utf8_len(m, k) + cw(m[k])$, where $cw(c)$ is the width of the encoding of c. The output type says that `encoded_len` should return exactly $utf8_len(m, n)$, subject to the precondition that the length fits in a `u32`.

Verification: CHC for `encoded_len`. To *verify* `encoded_len`, Flux generates the Constrained Horn Clause (CHC) shown on the right of Fig. 1. This CHC is a Lean proposition beginning with existentially quantified *Horn variables* (κ), which denote unknown refinements or invariants, followed by nested universally quantified binders naming statically unknown program values, conjunctions collecting requirements, and implications introducing hypotheses, ending in a *head* obligation that must hold under the preceding hypotheses. The CHC for `encoded_len` c_{top} has two Horn variables. The first, κ_{inv}, represents the `while`-loop invariant over the accumulator `acc`, the counter `i`, and the slice's elements `m` and length `n`. The second, κ_{w}, represents the unknown refinement on the width `w` obtained after "joining" the different branches of the nested `if-else` chain. For readability, we write c_{body} and c_{join} to name two nested sub-constraints. The first head $\kappa_{\text{inv}}(0, 0, m, n)$ requires the invariant to hold at loop entry. The second conjunct states that whenever the invariant κ_{inv} holds and the loop exits ($n \leq i$) that the returned `acc` equals the specified output $utf8_len(m, n)$, and that if instead the loop guard holds ($i < n$) then the loop-body sub-constraint c_{body} holds. The loop body reads the character $m[i]$ and has one conjunct for each branch of the `if` chain. Each branch's head, *e.g.* $\kappa_{\text{w}}(2, acc, i, m, n)$ checks that the κ_{w} join invariant holds for the value assigned to `w` on the corresponding branch. Its first argument is the width being refined; the remaining arguments record the program context. The last conjunct, c_{join}, encodes the code after the join: every width w satisfying $\kappa_{\text{w}}(w, acc, i, m, n)$ must not overflow `acc`, and the head $\kappa_{\text{inv}}(acc + w, i + 1, m, n)$ checks that the loop invariant is preserved by the loop body.

Foundational Solvers for Acyclic and Cyclic Horn Variables. To solve the CHC we must provide a proof of c_{top} by supplying *explicit solutions* for both Horn variables κ_{w} and κ_{inv}, that prove that they satisfy the constraints. SMT-based approaches can solve CHCs over their supported theories, but cannot handle arbitrary user-defined functions like *utf8_len*. Flex addresses this challenge in three steps. First, following [14] we use the notion of variable dependencies — where a Horn variable at the head of a constraint *depends on* the Horn variables in its hypotheses — to partition the variables into those that are *cyclic*, meaning they can transitively depend on themselves, and those that are *acyclic*. For example, in c_{top} (Fig. 1), κ_{inv} is cyclic: it appears in both the hypothesis $\kappa_{\text{inv}}(acc, i, m, n)$ and the head $\kappa_{\text{inv}}(acc + w, i + 1, m, n)$ of the inductive clause. In contrast, κ_{w} is acyclic: its heads depend on κ_{inv}, but no dependency leads from κ_{w} to itself. Second, we eliminate the acyclic variables like κ_{w} while constructing a proof certifying the soundness of the reduction (§ 2.2). Third, we use a fixpoint computation to infer and certify an inductive invariant for the cyclic variables like κ_{inv} (§ 2.3). Finally, at this point we are left with a plain (Horn-free) verification condition that can be discharged using standard Lean tactics and automation.

Verifying `encoded_len`. The left side of Fig. 2 shows how a user can prove the CHC for `encoded_len` using Flex's tactics. First we `unfold` the CHC definition, then we apply the `zap` tactic to solve

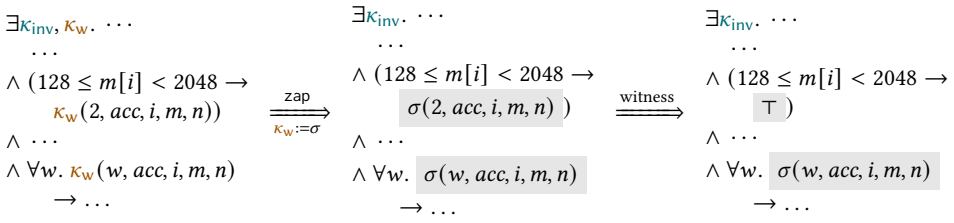


Fig. 3. Instantiating κ_w substitutes occurrences; applying proof-terms eliminates the heads.

and eliminate the acyclic Horn variables. Next, the `fix` tactic synthesizes an inductive invariant $0 \le i \le n \wedge acc = utf8_len(m, i)$ which, after substitution and elimination leaves the residual proof goal $acc + w \le \text{MAX}$ that checks the addition in the loop body does not overflow. The user looks at this residual obligation to separately write a lemma `utf8_len_mono` about the monotonicity of *utf8_len* that requires an inductive proof, and then applies the lemma to discharge the goal to verify `encoded_len`. Crucially, with our approach, the user can use the ITP's proof infrastructure to manage the entire complexity of the proof. The user can use Flex to synthesize invariants, and use the ITP's proof state context to identify relevant helper lemmas, and then apply them, all without cluttering the source code (Fig. 1) with proof artifacts. But all of that crucially hinges upon the ability to solve the Horn variables, which we describe next.

2.2 Solving *Acyclic* Horn Variables

We solve acyclic variables using Zap (§4) which exploits the structure of the constraint to synthesize a solution whose structure mirrors the constraint allowing us to construct the corresponding Lean witness deterministically, without search-based automation.

Computing the Solution. First, we compute an exact solution for the *acyclic* Horn variable focusing on the variable's *scope*: the smallest sub-constraint containing all the variable's occurrences. For κ_w, the scope is the body constraint c_{body}, and we can compute its strongest solution which is intuitively the disjunction of the hypotheses under which each head occurs:

$$\begin{aligned}\sigma \doteq \lambda z, acc, i, m, n. \quad & (m[i] < 128 \wedge z = 1) \\ & \vee (128 \le m[i] < 2048 \wedge z = 2) \\ & \vee (2048 \le m[i] < 65536 \wedge z = 3) \\ & \vee (65536 \le m[i] \wedge z = 4),\end{aligned}$$

Substituting σ as the solution for κ_w yields the instantiated constraint shown in the middle in Fig. 3. Each head application becomes the disjunction σ applied to the branch's width, and the hypothesis application becomes $\sigma(w, acc, i, m, n)$. Instantiating κ_w leaves behind the heads like $\sigma(2, acc, i, m, n)$ that must still be proved, *before* we can replace the head with $\top$ on the right. We could, in principle, prove those heads using proof search tactics such as `grind` or `aesop`, which suffice for this simple example. Sadly, in practice, proof search can be woefully slow (§ 7.2).

Proof-producing Elimination. Instead, our Zap tactic shows how to automatically construct the proof for each eliminated head implication, directly from the structure of the synthesized predicate. Consider the head obligation $\ldots (128 \le m[i] < 2048 \rightarrow \sigma(2, \ldots))$ and let g name the guard hypothesis $128 \le m[i] < 2048$. Recall the solution σ was computed as a *disjunction* of the different guards under which κ_w appears. Hence, we can prove this head via a proof term of the form $\mathsf{inr}(\mathsf{inl}(\langle g, \mathsf{rfl}\rangle))$ where the prefix of inr and inl record the path to this head, thereby identifying the specific disjunct of σ that must be proved, after which the pair $\langle g, \mathsf{rfl}\rangle$ proves the conjunct

Constraint	Solution	Head-proof construction
$c_1 \wedge c_2$	$sol\,c_1 \;\vee\; sol\,c_2$	inl / inr
$\forall x.\, c$	$\exists x.\, sol\,c$	$\langle x, \cdot \rangle$
$p \longrightarrow c$	$p \;\wedge\; sol\,c$	$\langle h, \cdot \rangle$
$\kappa(\bar{t})$	$\bigwedge_i z_i = t_i$	$\langle \mathsf{rfl}, \ldots, \mathsf{rfl} \rangle$

Fig. 4. Constraint structure determines the solution and its proof at an eliminated head. Here *sol* denotes the synthesized predicate body, z_i its formal arguments, and h the guard hypothesis.

$128 \leq m[i] < 2048 \wedge z = 2$ using the guard hypothesis g and reflexivity. We generalize this intuition in Fig. 4, and formalize it in a deterministic proof construction algorithm that lets Zap eliminate the heads and replace them with $\top$, as shown on the right in Fig. 3.

In our evaluation (§ 7.2), Zap successfully reduces 99.8% of 1,275 CHCs with eliminable predicates from refinement-typechecker suites and CHC-COMP. In contrast, when we instead use `grind` or `aesop` with the same resource limits, to prove the same synthesized heads, they only succeed on 32.2% and 33.4% of constraints respectively.

2.3 Solving *Cyclic* Horn Variables

The Horn variable κ_{inv} is cyclic: it appears as a hypothesis along the same path where it is the goal. Hence, we cannot apply Zap as it would loop indefinitely! Next, we show how Flex's Fix tactic infers an inductive invariant for κ_{inv} using *predicate-abstraction* [23, 27] in three steps (§ 5.2).

Step 1: Qualifiers The Fix tactic solves cyclic variables as *conjunctions of predicates* from a library of *qualifier* templates [51]. In Flex, qualifiers come from a standard library of arithmetic predicates, *e.g.* $\lambda a, b.a \leq b$, from user-supplied predicates, and from *scraping* the CHC and the defining equations and branch conditions of the relevant Lean definitions (§6). For example, the exit obligation of c_{top} already contains a useful predicate, and scraping abstracts its program variables to yield a qualifier over *utf8_len*:

$$acc = utf8_len(m, n) \xrightarrow{\text{scrape}} \lambda a, m, k.a = utf8_len(m, k).$$

Step 2: Initial Assignment Next, Fix uses the qualifiers to compute an initial assignment which instantiates *all* the qualifiers with all the possible κ_{inv} parameters that yield a well-typed Lean predicate. For example, the initial assignment for κ_{inv} would be

$$\kappa_{\mathsf{inv}}(a, z, m, n) \;\doteq\; \{0 \leq z, z \leq n, \ldots, a = utf8_len(m, z), a = utf8_len(m, n), \ldots\}$$

Step 3: Weaken till Fixpoint Finally, Fix enters an iterative weakening loop where we use an *oracle* — here, `grind` with the unfolding equation of *utf8_len* — to *discard* candidates that *cannot* be proven to hold at some CHC head, after substituting the current assignment for the Horn variables in the hypotheses. For example, $a = utf8_len(m, n)$ is discarded as it fails at loop entry: an arbitrary slice may have a nonzero encoded length. In contrast, $a = utf8_len(m, z)$ survives, as the oracle can use the recursive definition of *utf8_len* to prove that it holds initially and is preserved by the loop body:

$$0 = utf8_len(m, 0),$$
$$acc = utf8_len(m, i) \wedge 0 \leq i \wedge \sigma(w, acc, i, m, n) \;\longrightarrow\; acc + w = utf8_len(m, i + 1).$$

Once we have eliminated the impossible, whatever candidates remain, must be the truth. Hence, Fix returns a solution σ' for κ_{inv} comprising the conjunction of all remaining candidates

$$\sigma' \;\doteq\; \lambda a, z, m, n.0 \leq z \leq n \wedge a = utf8_len(m, z)$$

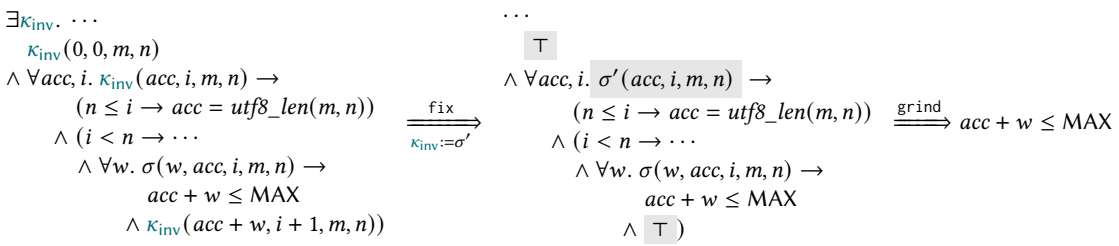


Fig. 5. (L) The CHC for `encoded_len` after `zap` (Fig. 3), (C) `fix` uses qualifiers to reduce the *cyclic* κ_{inv}, replacing head occurrences with $\top$ and body occurrences with the solution, (R) `grind` discharges every residual obligation except the overflow check proven by an inductive lemma in Fig. 2.

As with `zap`, we substitute this solution in for the hypotheses, but eliminate the heads with $\top$ as shown in Fig. 5. Finally, after applying `zap` and `fix`, we are left with the ordinary Lean goal corresponding to the overflow check $acc + w \leq \text{MAX}$, which the proof in Fig. 2 closes with an inductive lemma about *utf8_len*.

Certifying the Elimination. Again, to eliminate cyclic heads with $\top$, as shown in Fig. 5, Fix must construct Lean certificates that prove the corresponding implications, as detailed in our proof-producing predicate abstraction procedure in § 5.2. In addition to native Lean predicate abstraction which is useful for invariants for user-defined functions and bespoke theories, Flex also builds on the invariant-synthesis of Spacer [36] and Eldarica [33]. For SMT-LIB constraints these *external* solvers discover invariants independently of our qualifier library. Flex reconstructs their candidates as Lean predicates and certifies their use against the original constraint (§6), retaining any unproved obligations. We evaluate native expressive inference in § 7.1 and the proof coverage of these complementary approaches in § 7.3.

3 Constraints and Proofs

Flex represents CHCs as a restricted fragment of Lean propositions. This fragment preserves the structure needed for Horn solving while using native connectives such as $\forall$, $\wedge$, and $\rightarrow$. We therefore implement our solvers as metaprograms over Lean's *Expr* datatype, thus, like [26], getting correctness for free from Lean's kernel.

In this section we start by defining the syntax and semantics of constraints (§ 3.1) and then introduce notation for examining the structure of constraints (§ 3.2) and the dependencies between their unknown predicates (§ 3.3), preparing the ground for the solving algorithms that follow. As a running example throughout this section, consider a constraint that asserts $x \geq 0$ and an unknown refinement κ relating x to its successor v:

$$c_0 \doteq \exists \kappa.\ \forall x.\ 0 \leq x \rightarrow (\forall v.\ v = x + 1 \rightarrow \kappa(v, x)) \wedge (\forall a.\ \kappa(a, x) \rightarrow 1 \leq a) \tag{1}$$

3.1 Syntax, Semantics and Proofs

Fig. 6 defines the syntax of CHCs as Lean propositions and of the proof terms that certify their solutions.

Sorts and Terms *Sorts* b range over arbitrary Lean types, including integers, bit-vectors, sets, and maps. We write $x_1, x_2, x_3, \ldots$ for *variables* of sort b, and $t_1, t_2, t_3, \ldots$ for arbitrary Lean terms of the appropriate sort. We use φ for terms of sort *Prop*. We use h for proof variables whose types are propositions. We assume terms include the trivial propositions $\top$ and $\bot$ (of sort *Prop*).

Lean Term $t, \varphi \in \mathcal{T}$		***Variables*** $x, h \in \mathcal{V}$	***κ-variable*** $\kappa \in \mathcal{K} \subset \mathcal{V}$
Sort	b	::=	$\mathit{Prop} \mid \mathit{Int} \mid \mathit{Nat} \mid \mathit{Bool} \mid \mathit{BitVec}\ n \mid \cdots$
Atoms	p	::=	$\varphi \mid \kappa(\bar{t})$
Constraints	c	::=	$p \mid c \wedge c \mid p \to c \mid \forall x{:}b.\, c$
Closed Constraint	C	::=	$\exists \bar{\kappa}.c$
Proof term	q	::=	$t \mid \lambda x.\, q \mid q\ q \mid \langle q,\, q\rangle \mid q.1 \mid q.2 \mid \mathsf{inl}\, q \mid \mathsf{inr}\, q$
		$\mid$	$\langle t,\, q\rangle \mid \mathsf{let}\ \langle x,\, h\rangle = q\ \mathsf{in}\ q \mid \mathsf{rfl} \mid \langle\rangle$

Fig. 6. Grammar of the CHC fragment of Lean 4 propositions and their proof-terms as emitted by our solvers.

Horn Variables A *Horn (refinement) variable* is a Lean variable $\kappa : b_1 \to \cdots \to b_{n_\kappa} \to \mathit{Prop}$, and occurs in a constraint as horn applications $\kappa(\bar{t})$. Horn variables do not appear inside arbitrary terms t. We existentially quantify these variables to represent *unknown* refinements. For any constraint c, we write $\mathrm{KVars}(c)$ for the set of Horn variables occurring in c.

Atoms An *atom* p is either a κ-free term t of sort *Prop* or a *horn application* $\kappa(\bar{t})$.

Constraints A *Horn constraint* c is either (1) a *head* atom p, (2) a *conjunction* $c_1 \wedge c_2$, (3) an *implication* $p \to c$, or (4) a *quantification* $\forall x{:}b.\, c$ over a variable x of sort b. A *closed constraint* C is a Lean proposition of the form $\exists \kappa_1, \ldots, \kappa_n.c$, where all the refinement variables in c are existentially bound at the top. A *verification condition* (VC) is a closed constraint that has *no* Horn variables. We write c for both open and closed constraints when clear from context.

Predicates and Interpretations A *predicate* in $\mathcal{P}$ is a Lean term of sort $b_1 \to \ldots \to b_n \to \mathit{Prop}$ (*i.e.*, $\mathcal{P} \subset \mathcal{T}$). For a horn variable κ, an *interpretation* $\sigma \in \mathcal{P}$ is a predicate of the same type as κ. For c_0 of Eq. (1), the predicate $\sigma \doteq \lambda v\, x.\ v \geq 1$ is an interpretation for κ, since $\sigma : \mathit{Int} \to \mathit{Int} \to \mathit{Prop}$.

Satisfiability We *apply* an interpretation σ to a constraint c by replacing each application $\kappa(\bar{t})$ with $\sigma(\bar{t})$, writing $c[\kappa := \sigma]$ for the result. A closed constraint is *satisfiable* if interpretations for its Horn variables make the resulting κ-free proposition valid (*i.e.*, provable in Lean). Such interpretations form a *solution*. For example, applying σ to c_0 yields the following VC:

$$\forall x.\ 0 \leq x \to (\forall v.\ v = x + 1 \to v \geq 1) \wedge (\forall a.\ a \geq 1 \to 1 \leq a).$$

This VC is valid; hence σ is a solution and c_0 is satisfiable.

Proof Terms The solvers construct certificates using ordinary Lean proof terms (Fig. 6): λ-terms and applications for $\forall$ and implication; pairs and projections for conjunction; injections for disjunction; existential witnesses and unpacking; and rfl and $\langle\rangle$ for equality and $\top$. These forms, together with arbitrary Lean terms t, suffice for the constraint logic and the solutions computed by our solvers. Their typing is inherited from Lean, whose kernel checks each certificate. The same logical forms occur in Rocq [55] and Agda [48], so the certification approach is not specific to Lean.

3.2 Structural Operations: Scope & Contexts

Setting the ground for efficient constraint solving, here we define the structural operations that compute the smallest enclosing scope of each Horn variable and the hypotheses visible there.

Prefixes, Contexts, Heads A *prefix* ρ is a list of *steps* used to navigate a path through a constraint's syntax tree. A *step* s is either a left L or right R choice in a conjunction, a variable binding B, or an assumption G. A *context* Γ is an ordered list of variable bindings $x{:}b$ (one per $\forall$), named hypotheses

$\mathrm{Scope} : (\mathcal{K} \times C) \to \mathit{Prefix}$		
$\mathrm{Scope}(\kappa, \forall x{:}b.\, c)$	$\doteq$	$\mathsf{B} \cdot \mathrm{Scope}(\kappa, c)$
$\mathrm{Scope}(\kappa, c_1 \wedge c_2)$		
$\mid\ \kappa \in c_1,\ \kappa \notin c_2$	$\doteq$	$\mathsf{L} \cdot \mathrm{Scope}(\kappa, c_1)$
$\mid\ \kappa \notin c_1,\ \kappa \in c_2$	$\doteq$	$\mathsf{R} \cdot \mathrm{Scope}(\kappa, c_2)$
$\mathrm{Scope}(\kappa, p \to c)$		
$\mid\ \kappa \notin p$	$\doteq$	$\mathsf{G} \cdot \mathrm{Scope}(\kappa, c)$
$\mathrm{Scope}(\kappa, c)$	$\doteq$	ε

$(\downarrow) : (C \times \mathit{Prefix}) \to C$		
$(\forall x{:}b.\, c)\downarrow \mathsf{B} \cdot \rho$	$\doteq$	$c\downarrow\rho$
$(p \to c)\downarrow \mathsf{G} \cdot \rho$	$\doteq$	$c\downarrow\rho$
$(c_1 \wedge c_2)\downarrow \mathsf{L} \cdot \rho$	$\doteq$	$c_1\downarrow\rho$
$(c_1 \wedge c_2)\downarrow \mathsf{R} \cdot \rho$	$\doteq$	$c_2\downarrow\rho$
$c\downarrow\varepsilon$	$\doteq$	c

$(\uparrow) : (C \times \mathit{Prefix}) \to \mathit{Ctx}$		
$(\forall x{:}b.\, c)\uparrow \mathsf{B} \cdot \rho$	$\doteq$	$x{:}b; c\uparrow\rho$
$(p \to c)\uparrow \mathsf{G} \cdot \rho$	$\doteq$	$h{:}p; c\uparrow\rho$
$(c_1 \wedge c_2)\uparrow \mathsf{L} \cdot \rho$	$\doteq$	$\mathsf{L}; c_1\uparrow\rho$
$(c_1 \wedge c_2)\uparrow \mathsf{R} \cdot \rho$	$\doteq$	$\mathsf{R}; c_2\uparrow\rho$
$c\uparrow\varepsilon$	$\doteq$	ε

Fig. 7. $\mathrm{Scope}(\kappa, c)$ is the scope of a κ in c; $c\downarrow\rho$ is the *head of* c at ρ and $c\uparrow\rho$ is the *context of* c at ρ. In $\uparrow$, each guard hypothesis gets a fresh name h.

$h{:}p$ (one per $\to$), or a L or R choice gathered along a prefix.

$$\begin{array}{ll} \textbf{\textit{Prefixes}} & \rho ::= \varepsilon \mid s \cdot \rho \\ \textbf{\textit{Steps}} & s ::= \mathsf{L} \mid \mathsf{R} \mid \mathsf{B} \mid \mathsf{G} \\ \textbf{\textit{Contexts}} & \Gamma ::= \varepsilon \mid x{:}b; \Gamma \mid h{:}p; \Gamma \mid \mathsf{L}; \Gamma \mid \mathsf{R}; \Gamma \end{array}$$

The procedure $c\uparrow\rho$ computes this context, while $c\downarrow\rho$ returns the sub-constraint at the end of ρ (Fig. 7). The *truncated context* $\Gamma \setminus \rho$ drops the initial entries of Γ matching ρ.

Goals A prefix ρ is a *head-prefix* of c if $c\downarrow\rho$ is a head atom p. We write $\mathrm{heads}(c)$ for the set of Horn variables with a head application in c. A *goal* $\Gamma \vdash p$ is a context Γ paired with a head p. It is *valid* if p is derivable under the bindings and guards in Γ. For a head-prefix ρ, we define

$$\mathrm{Goal}(c, \rho) \doteq c\uparrow\rho \vdash c\downarrow\rho.$$

If its head is $\kappa(\bar{t})$, we call it a *κ-goal*; an interpretation σ solves it if $\Gamma \vdash \sigma(\bar{t})$ is valid.

Scopes The procedure $\mathrm{Scope}(\kappa, c)$ (Fig. 7) returns the prefix of the smallest sub-constraint containing all applications of κ, in both guards and heads. The constraint c_1 below extends c_0 with κ' under the guard $\kappa(a, x)$. Its scopes appear on the right ($\wedge$ is right associative).

$$c_1 \doteq \begin{array}{l} \forall x.\ 0 \le x \to \\ \quad \forall v.\ v = x + 1 \to \kappa(v, x) \\ \wedge\ \forall a.\ \kappa(a, x) \to \\ \quad 1 \le a \\ \wedge\ \kappa'(a + 1, a, x) \\ \wedge\ \forall v.\ \kappa'(v, a, x) \to 2 \le v \end{array} \qquad \begin{array}{lcl} \mathrm{Scope}(\kappa, c_1) & \doteq & \mathsf{B} \cdot \mathsf{G} \\ \mathrm{Scope}(\kappa', c_1) & \doteq & \mathsf{B} \cdot \mathsf{G} \cdot \mathsf{R} \cdot \mathsf{B} \cdot \mathsf{G} \cdot \mathsf{R} \end{array} \tag{2}$$

Well-formedness Let $\rho \doteq \mathrm{Scope}(\kappa, c)$, $\Gamma \doteq c\uparrow\rho$, and $\mathrm{dom}(\Gamma) \doteq x_0, \ldots, x_n$ list the scope's binders innermost first. A constraint c is *well-formed* for κ if every application has the form $\kappa(\bar{t}, x_0, \ldots, x_n)$: its trailing arguments are exactly these binders, while its leading arguments $\bar{t}$ are arbitrary terms. We write $\mathrm{rank}(\kappa)$ for the number of leading arguments. A constraint is *well-formed* if this holds for every $\kappa \in \mathrm{KVars}(c)$. We assume well-formed constraints throughout. For example, c_1 is well-formed: every application of κ ends in x, and every application of κ' ends in a, x, matching their respective scope binders.

3.3 Partitioning Horn Variables

Partitioning is a preprocessing step shared by the solvers; it determines which variables can be eliminated and which require invariant synthesis.

Dependencies The *dependencies* of a constraint c form a relation $\mathsf{Dep}(c) \subseteq \mathcal{K} \times \mathcal{K}$, defined by

$$\mathsf{Dep}(c) \doteq \{(\kappa, \kappa') \mid \exists \rho.\ \kappa \in c{\uparrow}\rho \text{ and } \kappa'(\bar{t}) = c{\downarrow}\rho\}.$$

We say κ' *depends on* κ in c if $(\kappa, \kappa') \in \mathsf{Dep}(c)$. This means that κ occurs in a guard along some path to a head application of κ'. The *dependency graph* $\mathsf{Graph}(c)$ has vertices $\mathsf{KVars}(c)$ and an edge from κ to κ' if κ' depends on κ. Intuitively, we solve for κ *before* solving for its dependents.

Cycles, Cuts and Partitions A *cut set* is a feedback vertex set of $\mathsf{Graph}(c)$: removing its vertices and incident edges makes the graph acyclic. The sets $\overline{\kappa_C}, \overline{\kappa_A}$ *partition* the Horn variables of c if (1) they are disjoint, (2) $\overline{\kappa_C} \cup \overline{\kappa_A} = \mathsf{KVars}(c)$, and (3) $\overline{\kappa_C}$ is a cut set. We call $\overline{\kappa_A}$ the *acyclic* set. The procedure $\mathsf{Partition}(\overline{\kappa}, c)$ computes such a partition. Within the acyclic set, each variable is bound outside its dependencies, so innermost-first elimination (§4) removes dependencies first. Finding a minimum feedback vertex set is NP-hard [35], so $\mathsf{Partition}(\overline{\kappa}, c)$ uses a greedy heuristic based on Tarjan's algorithm [54]. The cut set need not be minimum for soundness.

We pass the acyclic set $\overline{\kappa_A}$ to Zap for elimination (§4), and synthesize invariants for the cut set $\overline{\kappa_C}$ using predicate abstraction (§5) or external oracles (§6).

4 Zap: Proof-Producing Acyclic Horn Solving

Zap eliminates the Horn variables outside the cut set computed in § 3.3, which we call *acyclic* by a slight abuse of terminology. It computes strongest solutions and certifies the resulting rewrites. We describe its elimination (§ 4.1) and proof construction (§ 4.2) steps, then establish that the generated proofs are well-typed (§ 4.2). The remaining cut variables are handled by predicate abstraction (§5) or external invariant-synthesis oracles (§6).

4.1 Synthesizing Strongest Predicates

Top-level Algorithm (Zap): Fig. 8 defines the top-level Zap algorithm which takes as input a closed constraint $\exists\overline{\kappa}.\, c$, and returns a pair of a *new* constraint $\exists\overline{\kappa_C}.\, c'$ where all occurrences of the acyclic variables have been removed, and a *proof term* q' that shows that the output constraint *entails* the original constraint. The cut set $\overline{\kappa_C}$ and acyclic set $\overline{\kappa_A}$ are supplied by the preprocessing of § 3.3 and are left implicit in the notation $\mathsf{Zap}(c)$. Zap invokes Zaps to eliminate the acyclic variables from the innermost binder outward. On input $\exists\kappa.\, c$, Zaps first recursively eliminates the inner variables, obtaining c_1, then invokes Zap1 to eliminate κ from c_1. Zaps then composes the proofs $q_2 : c_2 \longrightarrow \exists\kappa.\, c_1$ and $q_1 : c_1 \longrightarrow c$ into a proof $q_3 : c_2 \longrightarrow \exists\kappa.\, c$ by unpacking and repacking the κ-witness. At the top, Zap re-bundles the cut and acyclic witnesses in the original binder order of $\overline{\kappa}$. The Zap1 function eliminates a *single* acyclic variable κ from a constraint c in two steps. In the first step, Zap1 invokes the Elim procedure to *eliminate* the variable κ by computing a triple comprising (1) the *strongest solution* σ for κ in c, (2) a prefix ρ corresponding to the scope of κ in c, and (3) a constraint c' where κ has been eliminated. In the second, Zap1 invokes $\mathsf{Cert}^{\kappa}_{\sigma,\rho}$ with the triple to construct the term $\lambda h.\langle \sigma,\ \mathsf{Cert}^{\kappa}_{\sigma,\rho}(\varepsilon, h, c)\rangle$ that *certifies* the elimination (as explained in § 4.2): it maps any proof h of c' to a proof of $\exists\kappa.\, c$, with σ as the witness. Fig. 9 illustrates the steps of Zap on the constraint c_1 from Eq. (2).

Variable Elimination (Elim): The procedure $\mathsf{Elim}(\kappa, c)$, shown in Fig. 8, eliminates κ from c in three steps. First, it computes the *scope* ρ of κ in c. Next, it calls Sol on $c{\downarrow}\rho$ — the sub-constraint of c where κ occurs — to compute σ: the *strongest solution* for κ. Finally, the procedure invokes $\mathsf{Red}(\kappa, \sigma, c)$, to compute the *reduced constraint* where all *head* applications of κ are replaced by $\top$, since they hold by construction of σ, and all other non-head occurrences of $\kappa(\bar{t})$ are replaced by the solution $\sigma(\bar{t})$, yielding a constraint *without* κ.

$\mathsf{Zap} : C \to (C \times \mathit{Proof})$

$$
\begin{array}{lcl}
\mathsf{Zap}(\exists \overline{\kappa}.\, c) & \doteq & (\exists \overline{\kappa_C}.\, c', q' \colon \exists \overline{\kappa_C}.\, c' \to \exists \overline{\kappa}.\, c) \\
\quad \textbf{where} & & \\
\quad (c', q) & \doteq & \mathsf{Zaps}(\exists \overline{\kappa_A}.\, c) \\
\quad q' & \doteq & \lambda h_1.\mathsf{let}\ \langle \overline{\kappa_C}, h_2 \rangle = h_1\ \mathsf{in} \\
& & \quad \mathsf{let}\ \langle \overline{\kappa_A}, h_3 \rangle = q\ h_2\ \mathsf{in} \\
& & \quad \langle \overline{\kappa},\ h_3 \rangle
\end{array}
$$

$\mathsf{Zaps} : C \to (C \times \mathit{Proof})$

$$
\begin{array}{lcl}
\mathsf{Zaps}(\exists \kappa.\, c) & \doteq & (c_2, q_3 \colon c_2 \to \exists \kappa.\, c) \\
\quad \textbf{where} & & \\
\quad (c_1, q_1) & \doteq & \mathsf{Zaps}(c) \\
\quad (c_2, q_2) & \doteq & \mathsf{Zap1}(\kappa, c_1) \\
\quad q_3 & \doteq & \lambda h_1.\mathsf{let}\ \langle \kappa, h_2 \rangle = q_2\ h_1\ \mathsf{in} \\
& & \quad \langle \kappa,\ q_1\ h_2 \rangle \\
\mathsf{Zaps}(c) & \doteq & (c, \lambda h.\ h)
\end{array}
$$

$\mathsf{Zap1} : (\mathcal{K} \times C) \to (C \times \mathit{Proof})$

$$
\begin{array}{lcl}
\mathsf{Zap1}(\kappa, c) & \doteq & (c', q \colon c' \to \exists \kappa.\, c) \\
\quad \textbf{where} & & \\
\quad (\sigma, \rho, c') & \doteq & \mathsf{Elim}(\kappa, c) \\
\quad q & \doteq & \lambda h.\langle \sigma,\ \mathsf{Cert}^{\kappa}_{\sigma,\rho}(\varepsilon, h, c) \rangle
\end{array}
$$

$\mathsf{Elim} : (\mathcal{K} \times C) \to (\mathcal{P} \times \mathit{Prefix} \times C)$

$$
\begin{array}{lcl}
\mathsf{Elim}(\kappa, c) & \doteq & (\sigma, \rho, c') \\
\quad \textbf{where} & & \\
\quad \rho & \doteq & \mathsf{Scope}(\kappa, c) \\
\quad \sigma & \doteq & \lambda \overline{z}.\ \lambda \overline{x}.\ \mathsf{Sol}(\kappa, c{\downarrow}\rho) \\
\quad c' & \doteq & \mathsf{Red}(\kappa, \sigma, c) \\
\quad \overline{z} & \doteq & z_0, \ldots, z_{n-1}\ \textbf{where}\ n = \mathsf{rank}(\kappa) \\
\quad \overline{x} & \doteq & \mathsf{dom}(c{\uparrow}\rho)
\end{array}
$$

$\mathsf{Red} : (\mathcal{K} \times \mathcal{P} \times C) \to C$

$$
\begin{array}{lcl}
\mathsf{Red}(\kappa, \sigma, c_1 \wedge c_2) & \doteq & \mathsf{Red}(\kappa, \sigma, c_1) \wedge \\
& & \quad \mathsf{Red}(\kappa, \sigma, c_2) \\
\mathsf{Red}(\kappa, \sigma, \forall x\!:\!b.\, c) & \doteq & \forall x\!:\!b.\, \mathsf{Red}(\kappa, \sigma, c) \\
\mathsf{Red}(\kappa, \sigma, p \to c) & \doteq & p[\kappa := \sigma] \to \mathsf{Red}(\kappa, \sigma, c) \\
\mathsf{Red}(\kappa, \sigma, \kappa(\overline{t})) & \doteq & \top \\
\mathsf{Red}(\kappa, \sigma, p) & \doteq & p
\end{array}
$$

$\mathsf{Sol} : (\mathcal{K} \times C) \to \mathcal{T}$

$$
\begin{array}{lcl}
\mathsf{Sol}(\kappa, c) & & \\
\quad \mid\ \kappa \notin \mathsf{heads}(c) & \doteq & \bot \\
\mathsf{Sol}(\kappa, c_1 \wedge c_2) & \doteq & \mathsf{Sol}(\kappa, c_1) \vee \mathsf{Sol}(\kappa, c_2) \\
\mathsf{Sol}(\kappa, \forall x\!:\!b.\, c) & \doteq & \exists x\!:\!b.\, \mathsf{Sol}(\kappa, c) \\
\mathsf{Sol}(\kappa, p \to c) & \doteq & p \wedge \mathsf{Sol}(\kappa, c) \\
\mathsf{Sol}(\kappa, \kappa(\overline{t})) & \doteq & \bigwedge_{i < \mathsf{rank}(\kappa)} z_i = t_i
\end{array}
$$

Fig. 8. Zap eliminates acyclic horn variables (left) using Elim to compute solutions (right).

$$
\begin{array}{l}
\exists \kappa', \kappa. \\
\forall x.\ 0 \le x \to \\
\quad \forall v.\ v = x + 1 \to \boxed{\kappa(v, x)} \\
\wedge\ \forall a.\ \boxed{\kappa(a, x)} \to 1 \le a \\
\quad \wedge\ \kappa'(a + 1, a, x) \\
\quad \wedge\ \forall v.\ \kappa'(v, a, x) \to 2 \le v
\end{array}
\xrightarrow[\lambda z_0\, x.\, z_0 = x+1]{\text{reduce } \kappa}
\begin{array}{l}
\exists \kappa'. \\
\forall x.\ 0 \le x \to \\
\quad \forall v.\ v = x + 1 \to \boxed{\top} \\
\wedge\ \forall a.\ \boxed{a = x + 1} \to 1 \le a \\
\quad \wedge\ \boxed{\kappa'(a + 1, a, x)} \\
\quad \wedge\ \forall v.\ \boxed{\kappa'(v, a, x)} \to 2 \le v
\end{array}
\xrightarrow[\lambda z_0\, a\, x.\, z_0 = a+1]{\text{reduce } \kappa'}
\begin{array}{l}
\forall x.\ 0 \le x \to \\
\quad \forall v.\ v = x + 1 \to \top \\
\wedge\ \forall a.\ a = x + 1 \to 1 \le a \\
\quad \wedge\ \boxed{\top} \\
\quad \wedge\ \forall v.\ \boxed{v = a + 1} \to 2 \le v
\end{array}
$$

Fig. 9. Eliminating acyclic variables with Zap. (L) initial constraint from Eq. (2), (C) result of reducing κ, and (R) result of reducing κ' yielding a VC with no Horn variables.

Strongest Solution (Sol): A solution σ is *valid* for κ in c if it is valid for every κ-goal (defined in §3). The procedure $\mathsf{Sol}(\kappa, c)$, shown in Fig. 8, computes the *body* of such a solution: the *disjunction* of the contexts at each head-prefix for κ, existentially quantifying the variables not in the scope; Elim closes the body under $\overline{z}$ and $\overline{x}$ to obtain σ. The guard $\kappa \notin \mathsf{heads}(c)$ collapses each sub-constraint without a κ head to an inert $\bot$, keeping dead-branch horn applications out of the solution. For example, Fig. 9 shows below each arrow, the solution computed for the κ reduced in the respective step. In the sub-constraint for κ there is exactly one head application, with the (truncated) context $v : \mathit{Int}; v = x + 1$; the conjunct with the *uses* of κ collapses to $\bot$. Thus, the computed solution is $\lambda z_0\, x.(\exists v.\ v = x + 1 \wedge z_0 = v) \vee \bot$ which we simplify to $\lambda z_0\, x.z_0 = x + 1$.

Scope avoids Exponential Blowup The procedure Elim is analogous to the similarly named one from [14], which introduces the scope optimization to sidestep an exponential blowup that otherwise occurs in practice. Specifically, if we compute $\mathsf{Sol}(\kappa, c)$ *without* considering the scope

$\mathsf{Cert}^{\kappa}_{\sigma,\rho}$	:	$(\mathit{Ctx} \times \mathit{Proof} \times C) \to \mathit{Proof}$
$\mathsf{Cert}^{\kappa}_{\sigma,\rho}(\Gamma, q, \forall x{:}b.\, c)$	$\doteq$	$\lambda x.\mathsf{Cert}^{\kappa}_{\sigma,\rho}(\Gamma; x{:}b, q\ x, c)$
$\mathsf{Cert}^{\kappa}_{\sigma,\rho}(\Gamma, q, p \to c)$	$\doteq$	$\lambda h_p.\mathsf{Cert}^{\kappa}_{\sigma,\rho}(\Gamma; h_p{:}p, q\ h_p, c)$
$\mathsf{Cert}^{\kappa}_{\sigma,\rho}(\Gamma, q, c_1 \wedge c_2)$	$\doteq$	$\langle\ \mathsf{Cert}^{\kappa}_{\sigma,\rho}(\Gamma; \mathsf{L}, q.1, c_1)$, $\mathsf{Cert}^{\kappa}_{\sigma,\rho}(\Gamma; \mathsf{R}, q.2, c_2)\ \rangle$
$\mathsf{Cert}^{\kappa}_{\sigma,\rho}(\Gamma, q, \kappa(\bar{t}))$	$\doteq$	$\mathsf{Nav}(\Gamma \setminus \rho, \sigma(\bar{t}))$
$\mathsf{Cert}^{\kappa}_{\sigma,\rho}(\Gamma, q, p)$	$\doteq$	q

Nav	:	$(\mathit{Ctx} \times \mathcal{P}) \to \mathit{Proof}$
$\mathsf{Nav}(\mathsf{L}; \Gamma, p_L \vee p_R)$	$\doteq$	$\mathsf{inl}\ \mathsf{Nav}(\Gamma, p_L)$
$\mathsf{Nav}(\mathsf{R}; \Gamma, p_L \vee p_R)$	$\doteq$	$\mathsf{inr}\ \mathsf{Nav}(\Gamma, p_R)$
$\mathsf{Nav}(x{:}b; \Gamma, \exists z{:}b.\, p)$	$\doteq$	$\langle x,\ \mathsf{Nav}(\Gamma, p[z := x])\rangle$
$\mathsf{Nav}(h{:}p; \Gamma, p \wedge p')$	$\doteq$	$\langle h,\ \mathsf{Nav}(\Gamma, p')\rangle$
$\mathsf{Nav}(\varepsilon, (\bigwedge_i z_i = t_i))$	$\doteq$	$\langle\ \mathsf{rfl}, \ldots, \mathsf{rfl}\ \rangle$
$\mathsf{Nav}(\varepsilon, \top)$	$\doteq$	$\langle\rangle$

Fig. 10. $\mathsf{Cert}^{\kappa}_{\sigma,\rho}$ transforms a proof of the reduced constraint into a proof of the original, invoking Nav at each divergent κ-head to rebuild a proof of $\sigma(\bar{t})$ by replaying the truncated context against the structure of σ.

ρ, then the assumptions from $c \uparrow \rho$ get (unnecessarily) included in the strongest solution, which causes an exponential duplication of assumptions as we eliminate multiple variables.

4.2 Constructing Elimination Proofs

In a foundational setting, we cannot just rewrite a constraint, we also need to prove the original one. To this end, $\mathsf{Cert}^{\kappa}_{\sigma,\rho}$ *converts* a proof of the rewritten constraint into a proof of the original one.

Certifying an Assignment ($\mathsf{Cert}^{\kappa}_{\sigma,\rho}$): The procedure $\mathsf{Cert}^{\kappa}_{\sigma,\rho}(\Gamma, q, c)$ (Fig. 10) takes two sets of inputs. First, a parametric set comprising κ, its solution σ and its scope ρ in c. Second, a set with a context Γ of accumulated bindings, the original sub-constraint c, and a proof q of the σ-reduced sub-constraint (*i.e.*, $c' \doteq \mathsf{Red}(\kappa, \sigma, c)$). The procedure then traverses c, transforming q at each step into a proof of the corresponding sub-constraint. For the $\forall x{:}b.\, c$ (resp. $p \to c$) case, the procedure emits a λ-term that applies the supplied input x (resp. h_p) to q to recursively generate the certificate for the sub-constraint c. For a $c_1 \wedge c_2$ the procedure emits a pair term comprising the sub-terms obtained from the proofs of c_1 and c_2 constructed using the proofs $q.1$ and $q.2$ respectively. The other heads p are unchanged so we can directly reuse the corresponding proof term q.

For example, with $\sigma \doteq \lambda z\, x.z = x + 1$, the clause $v = x + 1 \to \kappa(v, x)$ reduces to $v = x + 1 \to \top$. After substituting σ, $\mathsf{Cert}^{\kappa}_{\sigma,\rho}$ reconstructs its proof as: $\lambda(h : v = x + 1).\, h\ :\ v = x + 1 \to v = x + 1$.

Scope Optimization and Certification The scope optimization complicates proof construction: Sol operates on a scoped sub-constraint, omitting its enclosing guards, while $\mathsf{Cert}^{\kappa}_{\sigma,\rho}$ traverses the original constraint and accumulates the full context. Certification must therefore align the synthesized predicate with the corresponding part of this context, preserving the scope's variable bindings. The prefix machinery (§ 3.2 and Fig. 7) records this correspondence and guides the construction of the proof terms.

Reconciling divergent heads (Nav): At the divergent heads in $\mathsf{Cert}^{\kappa}_{\sigma,\rho}$ we call Nav to furnish us with a proof for the original goal $\kappa(\bar{t})$ post-substitution, *i.e.*, for $\sigma(\bar{t})$. Recall that the solution σ was a disjunction of contexts at each head-prefix for κ in the sub-constraint $c \downarrow \rho$ where κ occurs. Thus, we use the *truncated* context $(\Gamma \setminus \rho)$ that led to the divergent head to *reconstruct* the proof term for that disjunction from the context.

Example In the example in Fig. 9, κ has the scope prefix $\rho \doteq \mathsf{B} \cdot \mathsf{G}$. Thus, at the head $\kappa(v, x)$, $\mathsf{Cert}^{\kappa}_{\sigma,\rho}$ has accumulated the context $\Gamma \doteq x{:}\mathit{Int}; h_0{:}0 \le x; \mathsf{L}; v{:}\mathit{Int}; h_1{:}v = x + 1$ (naming the guard hypotheses h_0, h_1), and the truncated context is $\Gamma \setminus \rho \doteq \mathsf{L}; v{:}\mathit{Int}; h_1{:}v = x + 1$. Given the solution $\sigma \doteq \lambda z_0\, x.(\exists v'.\, v' = x + 1 \wedge z_0 = v') \vee \bot$ (the bound variable renamed to v'), Nav replays the path,

as follows, consuming one entry per node:

$$\begin{aligned}
& \mathsf{Nav}(\mathsf{L}; v{:}\mathit{Int}; h_1 : v = x+1, \sigma(v,x)) \\
= \ & \mathsf{inl}\ \mathsf{Nav}(v{:}\mathit{Int}; h_1 : v = x+1, \exists v'.\, v' = x{+}1 \wedge v = v') && \mathsf{L}\text{ picks the live disjunct} \\
= \ & \mathsf{inl}\ \langle v,\ \mathsf{Nav}(h_1 : v = x+1, (v = x{+}1) \wedge v = v)\rangle && v\text{ witnesses } \exists v' \\
= \ & \mathsf{inl}\ \langle v,\ \langle h_1,\ \mathsf{Nav}(\varepsilon, v = v)\rangle\rangle && h_1\text{ discharges } v = x{+}1 \\
= \ & \mathsf{inl}\ \langle v,\ \langle h_1,\ \mathsf{rfl}\rangle\rangle. && \text{slot equality is rfl}
\end{aligned}$$

Equivalence and Soundness Zap inherits satisfiability results from Cosman and Jhala [14]. We proved, in Appendix A, generated proofs of Zap are sound, with respect to a model of Lean kernel. That is, all proofs generated by Zap will type check under Lean kernel.

Theorem 4.1 (Soundness of Certificates)**.** *If* $(c', q) = \mathsf{Zap}(c)$ *then* $\vdash q : c' \to c$.

5 Fix: Proof-Producing Predicate Abstraction

After Zap (§4) eliminates acyclic Horn variables, Fix synthesizes inductive invariants for the remaining variables using *predicate abstraction* [27]. Each invariant is a conjunction of instances drawn from a library of predicate templates, called *qualifiers*. While traditional SMT-based implementations [23, 51] restrict qualifiers to theories supported by their solvers, in our foundational setting, qualifiers can be arbitrary Lean predicates. In this section, we define qualifiers and their instantiation (§ 5.1) and present the fixpoint algorithm (§ 5.2). Finally, we explain proof certification and establish soundness and relative completeness (§ 5.3). Qualifier discovery is described in the implementation (§6).

5.1 Qualifiers, Instances and Candidates

Informally, qualifiers are predicate *templates*. Formally, a *qualifier* q is a Lean predicate of the form

$$q \doteq \lambda z_0 : b_0, \ldots, z_n : b_n.t.$$

Let $\kappa : b'_0 \to \ldots \to b'_m \to \mathit{Prop}$ be a Horn variable. We say that ι is a κ/q *instance* if $\iota : [0,n] \to [0,m]$ and $b_j = b'_{\iota(j)}$ for each $0 \le j \le n$. That is, an instance ι maps each parameter of the qualifier to a parameter of κ of the same sort. A *candidate* is a pair of a qualifier q and an instance ι, written q_ι. We overload q_ι to denote the predicate induced by instantiating q along ι:

$$q_\iota \doteq \lambda z_0 \cdots z_m.q(z_{\iota(0)}, \ldots, z_{\iota(n)}).$$

Given a *candidate set* $\overline{q_\iota}$, we write $\bigwedge \overline{q_\iota}$ for its pointwise conjunction.

Example Consider a Horn variable $\kappa_{\mathsf{ex}} : \mathit{Int} \to \mathit{Int} \to \mathit{Prop}$ together with the binary qualifier

$$q_{\le} \doteq \lambda z_0 : \mathit{Int},\ z_1 : \mathit{Int}.z_0 \le z_1.$$

Each of its two parameters of sort *Int* can be mapped to either of the parameters of κ_{ex}. Hence, there are four $\kappa_{\mathsf{ex}}/q_{\le}$ instances, yielding the candidate set

$$\{\lambda z_0\, z_1.z_0 \le z_0,\quad \lambda z_0\, z_1.z_0 \le z_1,\quad \lambda z_0\, z_1.z_1 \le z_0,\quad \lambda z_0\, z_1.z_1 \le z_1\}\,.$$

Assignments, Application and Validity An *assignment* $\theta \in \mathcal{A}$ maps each κ to a candidate set. We *apply* θ to a Horn application, written $\kappa(\bar{t})[\theta]$, by applying the pointwise conjunction of the candidate set $\theta(\kappa)$. We lift application to contexts $\Gamma[\theta]$ by applying it to each hypothesis in Γ.

$$\begin{array}{lcll}
\kappa(\bar{t})[\theta] & \doteq & (\bigwedge \theta(\kappa))(\bar{t}) & \text{Horn applications} \\
t[\theta] & \doteq & t & \text{other atoms} \\
h{:}p; \Gamma[\theta] & \doteq & h{:}p[\theta]; \Gamma[\theta] & \text{hypotheses} \\
x{:}b; \Gamma[\theta] & \doteq & x{:}b; \Gamma[\theta] & \text{binders} \\
\mathsf{L}; \Gamma[\theta] & \doteq & \mathsf{L}; \Gamma[\theta] & \text{marks (R alike)}
\end{array}$$

$\mathsf{Fix} : (C \times Q) \to (C \times \mathit{Proof})$

$$
\begin{array}{ll}
\mathsf{Fix}(\exists\overline{\kappa_C}.\, c, Q) & \doteq (c', q{:}\ c' \to \exists\overline{\kappa_C}.\, c) \\
\quad \textbf{where} & \\
\quad\quad \theta & \doteq \mathsf{PredAbs}(\exists\overline{\kappa_C}.\, c, Q) \\
\quad\quad c' & \doteq \mathsf{Red}^*(\overline{\kappa_C}, c) \\
\quad\quad q & \doteq \lambda h.\langle \overline{\bigwedge \theta(\kappa_C)},\ \mathsf{Cert}_\theta(\varepsilon, h, c)\rangle \\
\quad\quad \mathsf{Red}^*(\varepsilon, c) & \doteq c \\
\quad\quad \mathsf{Red}^*(\kappa; \overline{\kappa}, c) & \doteq \mathsf{Red}^*(\overline{\kappa}, \mathsf{Red}(\kappa, \bigwedge \theta(\kappa), c))
\end{array}
$$

$\mathsf{Cert}_\theta : (\mathit{Ctx} \times \mathit{Proof} \times C) \to \mathit{Proof}$

$$
\begin{array}{ll}
\mathsf{Cert}_\theta(\Gamma, q, \forall x{:}b.\, c) & \doteq \lambda x.\mathsf{Cert}_\theta(\Gamma; x{:}b, q\ x, c) \\
\mathsf{Cert}_\theta(\Gamma, q, p \to c) & \doteq \lambda h_p.\mathsf{Cert}_\theta(\Gamma; h_p{:}p, q\ h_p, c) \\
\mathsf{Cert}_\theta(\Gamma, q, c_1 \wedge c_2) & \doteq \langle\ \mathsf{Cert}_\theta(\Gamma; \mathsf{L}, q.1, c_1) \\
 & \quad ,\ \mathsf{Cert}_\theta(\Gamma; \mathsf{R}, q.2, c_2)\ \rangle \\
\mathsf{Cert}_\theta(\Gamma, q, \kappa(\bar{t})) & \doteq \mathsf{Oracle}(\Gamma[\theta] \vdash (\bigwedge \theta(\kappa))(\bar{t})) \\
\mathsf{Cert}_\theta(\Gamma, q, p) & \doteq q
\end{array}
$$

$\mathsf{PredAbs} : (C \times Q) \to \mathcal{A}$

$$
\begin{array}{ll}
\mathsf{PredAbs}(\exists\overline{\kappa_C}.\, c, Q) & \doteq \mathsf{Fixpoint}(c, \theta) \\
\quad \textbf{where} & \\
\quad\quad \theta & \doteq [\kappa \mapsto \mathsf{Init}(\kappa, Q) \mid \kappa \in \overline{\kappa_C}] \\
\quad\quad \mathsf{Init}(\kappa, Q) & \doteq \{q_\iota \mid q \in Q, \iota \in \kappa/q \text{ instances}\}
\end{array}
$$

$\mathsf{Fixpoint} : (C \times \mathcal{A}) \to \mathcal{A}$

$$
\begin{array}{ll}
\mathsf{Fixpoint}(c, \theta) & \doteq \textbf{if } \exists\rho.\theta' \doteq \mathsf{Weaken}(c, \theta, \rho) \\
 & \quad\quad \textbf{then } \mathsf{Fixpoint}(c, \theta') \textbf{ else } \theta
\end{array}
$$

$\mathsf{Weaken} : (C \times \mathcal{A} \times \mathit{Prefix}) \to \mathcal{A} + \bot$

$$
\begin{array}{ll}
\mathsf{Weaken}(c, \theta, \rho) & \doteq \textbf{if } I \subset \theta(\kappa) \textbf{ then } \theta[\kappa \mapsto I] \textbf{ else } \bot \\
\quad \textbf{where} & \\
\quad\quad \Gamma \vdash \kappa(\bar{t}) & \doteq \mathsf{Goal}(c, \rho) \\
\quad\quad I & \doteq \{q_\iota \mid q_\iota \in \theta(\kappa), \mathsf{Oracle}(\Gamma[\theta] \vdash q_\iota(\bar{t}))\}
\end{array}
$$

Fig. 11. The Fix procedure for reducing cyclic horn variables via predicate abstraction.

An assignment θ is *valid* for a κ-goal $\Gamma \vdash \kappa(\bar{t})$ if the κ-free goal $\Gamma[\theta] \vdash \kappa(\bar{t})[\theta]$ is valid. We write $\theta \models c$ when θ is valid for every κ-goal of c.

5.2 Fixpoint Algorithm

Fig. 11 summarizes Fix, which takes a constraint c and qualifiers Q and returns a pair (c', q): a constraint c' with all Horn variables removed, and a proof q that $c' \to c$. It first computes an assignment θ, invoking PredAbs on the constraint and qualifiers. It solves *all* Horn variables simultaneously to account for their cyclic dependencies. Next, it folds Red over c and all the horn variables, to reduce the head applications of κ with $\top$ and the body applications with the computed candidate set $\theta(\kappa)$. Finally, it invokes Cert_θ to compute the proof q of $c' \to c$ (§ 5.3).

Predicate Abstraction (PredAbs**):** The right column in Fig. 11 shows how PredAbs takes Horn variables $\overline{\kappa}$, a constraint c, and qualifiers Q to compute an assignment valid for c. First, Init computes an *initial assignment* mapping each κ to all candidates obtained from κ/q instances of each qualifier in Q. Second, it invokes Fixpoint to distill these candidates down to those that can be proven valid at every head κ-application, allowing Fix to replace these heads with $\top$.

Fixpoint Computation (Fixpoint**):** Fixpoint takes a constraint c and assignment θ, and iteratively invokes Weaken to whittle away candidates that *cannot be proven* valid. For a head-prefix ρ, Weaken uses Oracle to determine the subset I of $\theta(\kappa)$ that can be proven valid at the corresponding head. If I is strictly smaller than $\theta(\kappa)$, *i.e.*, some whittling occurred, Weaken returns the assignment with κ updated to I; otherwise, it returns $\bot$. Fixpoint checks whether *any* head-prefix weakens θ to θ'. If so, it recurses on θ'. Otherwise, θ is valid for c and is returned. Fixpoint terminates: the initial assignment is finite, and each successful Weaken strictly decreases the total number of candidates. Weaken, and hence Fix, is parameterized by Oracle, which attempts to *prove* a goal. This lets users plug in theory-specific oracles to synthesize solutions for CHCs over arbitrary Lean propositions.

5.3 Certification and Correctness

Certification (Cert_θ**):** Finally, Fix invokes Cert_θ to justify the rewriting done by Red_θ, producing a proof that the reduced constraint c' *implies* the original c. Like Zap's certification (§ 4.2), it

traverses the structure of c, replaying the proof of c'. The only difference is at head applications $\kappa(\bar{t})$, where, instead of navigating the structure of the solution, it invokes Oracle to prove the applied conjunction $(\bigwedge \theta(\kappa))(\bar{t})$.

Correctness Fix requires that on any goal, Oracle either returns a valid proof term, or fails:

$$\textsc{Oracle-Soundness}\ \frac{\mathsf{Oracle}(\Gamma \vdash \varphi) = q}{\Gamma \vdash q : \varphi} \qquad \textsc{Oracle-Completeness}\ \frac{\Gamma \vdash \varphi}{\mathsf{Oracle}(\Gamma \vdash \varphi)\ \text{succeeds}}$$

Theorem 5.1 (Fix-Soundness)**.** *If* Oracle *is sound and* $\mathsf{Fix}(c, Q) = (c', q)$*, then* $\vdash q : c' \longrightarrow c$.

Relative Completeness It would be trivially sound, but not terribly useful to replace each horn application with $\top$. Fortunately, $\mathsf{Fix}(c, Q)$ does better: it returns the *strongest* valid solution expressible as conjunctions of predicates from Q. Given two Q-assignments, we say θ *is stronger than* θ', written $\theta \preceq \theta'$, if for every κ, we have $\theta'(\kappa) \subseteq \theta(\kappa)$.

Theorem 5.2 (Fix-Completeness [51])**.** *If* Oracle *is complete and* $\theta \doteq \mathsf{PredAbs}(\exists \overline{\kappa}.\, c, Q)$*, then* $\theta \preceq \theta'$ *for every* Q*-assignment* θ' *with* $\theta' \models c$.

6 Implementation

We implemented Flex in approximately 9,200 lines of Lean. Flex integrates native solvers (Zap and Fix) with external solvers (Spacer [36] and Eldarica [33]) in a unifying `flex` tactic. Horn solving is divided into *synthesis*, whereby a solver proposes solutions for Horn variables, and *certification*, whereby Flex proves that the residual constraint implies the original CHC.

Native Solvers. The implementations of Zap and Fix closely follow the algorithms described in §4 and §5. We instantiate the Fix oracle in the weakening loop with a query to Z3 [16] on an SMT-LIB encoding of each goal, falling back to native Lean automation when Z3 is inconclusive. The oracle need not be trusted as it only selects candidates, which certification must still prove.

External Solvers. Flex invokes Spacer [36] and Eldarica [33] on SMT-LIB exports of supported constraints, and reconstructs the computed solutions as Lean predicates. The kernel checks the resulting proofs, so neither the solvers nor the translation are trusted.

Discharging residual VC. After eliminating κ-variables in a CHC, we are left with a residual VC to prove in Lean. `flex` tries to construct a proof for the residual with a combination of native Lean tactics (`simp`, `omega`, `grind`, `aesop`) and `lean-smt` [44], which reconstructs cvc5 [2] proofs as Lean terms. When it succeeds, `flex` yields a *full* proof of the original constraint. Otherwise, it may still make *partial* progress if a Horn solution is proven strong enough. This is true by construction for solutions found by Zap and the external solvers when the SMT-LIB encoding is exact. Because Fix's completeness is relative to qualifiers and oracle, its solution may be too weak, so `flex` discards any Fix solution that cannot be certified.

Scraping Qualifiers The performance of Fix depends on the *qualifiers* supplied to it. Flex supports user-defined qualifiers over arbitrary Lean definitions and automates discovery by *scraping* CHCs and selected native definitions. Scraping abstracts predicates into templates, combines arithmetic atoms into guarded qualifiers, and inspects defining equations for further candidates. PredAbs checks their instances during inference. To control the search space, Flex starts with the built-in bank and progressively deepens scraping when a level fails to yield an accepted invariant. Detailed rules appear in §C; RQ4 (§7.4) evaluates the scraping strategies.

7 Evaluation

We evaluate Flex across four research questions:

					Zap + Fix				Spacer			Eldarica		
Case study	Rust	Spec.	Proof	CHCs	Full	Partial	Med.	Total	Full	Med.	Total	Full	Med.	Total
Fibonacci	147 (84)	50 (11)	119	4	3	0	7.3 s	76 s	0	<0.1 s	60 s	1	0.7 s	4 s
Sorting	273 (175)	103 (37)	620	11	6	5	17.0 s	196 s	0	0.2 s	63 s	6	4.7 s	269 s
RingBuffer	108 (65)	152 (29)	501	2	0	2	19.8 s	40 s	0	0.4 s	1 s	0	31.4 s	63 s
HashTable	260 (168)	302 (156)	1361	12	7	4	5.6 s	250 s	1	0.2 s	4 s	0	3.0 s	258 s
VecDeque	3147 (215)	1269 (109)	1060	11	0	11	9.5 s	199 s	0	2.2 s	89 s	0	60.2 s	550 s
CStr	1105 (64)	247 (19)	245	2	2	0	10.2 s	20 s	2	0.4 s	1 s	1	31.8 s	64 s
String searching	1685 (513)	659 (157)	2949	17	7	10	22.9 s	458 s	5	0.8 s	87 s	5	6.1 s	385 s
All	6725 (1284)	2782 (518)	6855	59	25	32	14.3 s	1238 s	8	0.4 s	305 s	13	6.1 s	1592 s

Table 1. Case-study sizes and results. Rust: lines of Rust code; Spec.: lines of Flux specifications; Proof: lines of Lean proof. The parenthesized numbers give the Rust and Spec lines relevant to the code verified in Lean. Full: the proof is fully automated; Partial: the proof is not fully automated, but Zap eliminates at least one predicate, leaving a residual proof with fewer predicates. Each solution found by Spacer and Eldarica ended in a full proof. Med.: median time per CHC including failures and timeouts; Total: the total time to run Flex.

- **RQ1:** How effective is Flex on expressive case studies? (§7.1)
- **RQ2:** How efficient is Zap elimination compared with search-based tactics in Lean? (§7.2)
- **RQ3:** How much automated proof coverage does Flex provide? (§7.3)
- **RQ4:** How do different *qualifier scraping* strategies affect solver performance? (§7.4)

To evaluate RQ2 and RQ3 we draw CHCs from two sources. First, we use 540 nested CHCs of the kind refinement type checkers generate: 337 from Flux's regression tests, 96 from the Flux guide including implementation of vector-manipulating algorithms, 36 from the verification of WaVe [34, 38], and 71 from Liquid Fixpoint's regression tests. To evaluate RQ4, we select set of fully proved CHCs by Fix to enable ablation on different scraping strategies. Second, to complement the nested constraints, we use the 1,559 distinct conventional clause-based CHCs that CHC-COMP 2026 [9] labels satisfiable, covering linear and nonlinear Horn clauses over linear integer arithmetic.

We run our experiments on a workstation running Ubuntu 24.04.4 LTS, with an AMD Ryzen 7 8845HS processor (8 cores, 16 threads) and 27 GB of RAM.

7.1 RQ1: Expressive Invariant Inference and Elimination by Flex

To evaluate how well Flex infers rich invariants, we extend Flux, a refinement type checker for Rust, with a Lean backend in about 1,800 lines of Rust that emits its verification conditions as Lean CHCs. We then study Rust programs whose correctness requires reasoning about recursive functions, quantified invariants over collections, or nonlinear arithmetic, which is beyond what Flux's existing SMT backend can express or automate. Table 1 summarizes the case studies, which total 6,725 lines of Rust, of which 1,284 are relevant to the code verified in Lean.

- **Fibonacci.** We verify Fibonacci number computations, including imperative and memoized versions, against pure recursive specifications.
- **Sorting.** We verify that quicksort, merge sort, and insertion sort return sorted permutations of their inputs.
- **RingBuffer.** We verify that a ring buffer adapted from Tock, a production embedded operating system [52], avoids reads from uninitialized memory and implements a queue interface.
- **HashTable.** We verify that insertion, lookup, removal, and resizing of a hash table with chaining are correct against functional Lean models.

- **VecDeque.** We verify that vector-backed double-ended queue operations adapted from Rust's standard library update the contents of the deque as expected.
- **CStr.** The safety of C-string operations is a target of the Rust standard library verification initiative [13]. We verify that operations adapted from the standard library maintain the invariant that a string contains exactly one null byte, at the end.
- **String searching.** String searching is also a target of the same initiative. We verify that string searchers adapted from Rust's standard library return matches that are aligned with UTF-8 character boundaries.

These case studies produce 126 VCs, including 61 CHCs. We exclude two trivial tests, `SortTest1` and `VectorsTest1`, evaluating 59 CHCs. All the CHCs fall outside the standard CHC fragment, in which constraints are quantifier-free formulas over linear arithmetic, possibly with arrays or algebraic data types [5].

Experiment design. We run Flex in three configurations that differ only in how solutions are synthesized: Zap followed by Fix, Spacer alone, and Eldarica alone. Fix uses the built-in bank and qualifiers scraped from the CHC and its referenced Lean definitions (§6); we supply no qualifiers or invariants by hand. In every configuration, Flex then certifies the solutions and discharges the residual in Lean. For each CHC, synthesis is limited to 60 s and a 180 s time budget for the entire proof. Eldarica supports some of the CHCs through its extensions for quantifiers and arrays, so we pass it quantified definitions directly. Natively, Spacer rejects all CHCs but one, which uses only nonlinear arithmetic, since it supports neither quantifiers in clause bodies nor recursive definitions. We therefore encode every definition a solver cannot handle, such as a recursive function, as an uninterpreted function, in the hope that the invariant does not require unfolding it. Following Bjørner et al. [5], each such function becomes an array over its parameters, universally quantified in every clause and passed to every Horn variable as an extra parameter.

Results. Of the 59 CHCs, Zap + Fix fully proves 25, while Spacer proves 8 and Eldarica 13. The solved sets largely coincide: every CHC proved by Spacer or Eldarica is also proved by Zap + Fix, with a single exception that only Eldarica proves. In addition, Zap + Fix makes partial progress on 32 more. For instance, on VecDeque, Zap + Fix makes partial progress on all 11 CHCs; leaving a residual proof with strictly fewer Horn variables which we then prove interactively. Spacer and Eldarica make no partial progress: each solution they find ends in a full proof. Zap + Fix takes a median of 14.3 s per CHC (1,238 s in total), including certification and residual discharge. Spacer is significantly faster, with a median of 0.4 s mostly because it fails quickly. Eldarica's median time is faster than Zap + Fix, but in total Eldarica takes 29% longer than Zap + Fix.

7.2 RQ2: Efficiency of Zap

Zap (§4) instantiates acyclic κ-variables with their strongest solutions and constructs proofs justifying replacement of their head occurrences with $\top$. We compare this deterministic proof construction with proof search using Lean's `grind` and `aesop` tactics.

Experiment design. We implemented two variants, Zap_grind and Zap_aesop, which compute the same strongest solution as Zap using Sol, but replace $\mathsf{Cert}^{\kappa}_{\sigma,\rho}$'s proof-term construction with calls to `grind` and `aesop`, respectively, each using its default settings. We run all three variants on 1,275 CHCs with eliminable variables: 415 acyclic-only and 860 mixed. These comprise 514 CHCs from the refinement-typechecker suites and case studies introduced above, and 761 from CHC-COMP. All variants use identical resource limits.

Results. Table 2 reports successful reductions and, over those, mean wall-clock time and heartbeats—Lean's deterministic proxy for proof effort. Zap reduces 1,272 constraints (99.8%), averaging 472 ms. Its three failures exceed the 60 s timeout; Zap_grind and Zap_aesop also fail on these constraints.

		CHCs			Zap			Zap_grind			Zap_aesop		
	Suite	Total	Acyclic	Mixed	Reduced	hb	ms	Reduced	×hb	×ms	Reduced	×hb	×ms
	`flux-tests`	295	240	55	295 (100%)	42.1	6.8	216 (73.2%)	44.9×	17.0×	250 (84.7%)	85.1×	31.6×
	`flux-medium`	80	36	44	80 (100%)	185	61.7	46 (57.5%)	141×	54.1×	48 (60%)	162×	63.7×
	`wave`	35	26	9	35 (100%)	174	36.2	19 (54.3%)	94.6×	32.6×	20 (57.1%)	125×	44.4×
	`liquid-fix`	57	50	7	57 (100%)	125	42.2	40 (70.2%)	33.8×	12.8×	49 (86%)	65.2×	26.7×
	Case studies	47	27	20	47 (100%)	1,167	563	9 (19.1%)	432×	133×	9 (19.1%)	628×	181×
CHC-COMP	Eldarica	97	0	97	97 (100%)	6,628	2,215	16 (16.5%)	128×	66.7×	4 (4.1%)	297×	176×
CHC-COMP	HCAI	104	0	104	104 (100%)	203	31.4	26 (25%)	228×	118×	14 (13.5%)	1,461×	549×
CHC-COMP	HoPV	78	32	46	78 (100%)	533	101	28 (35.9%)	189×	81.0×	25 (32.1%)	266×	128×
CHC-COMP	Kind2	395	0	395	393 (99.5%)	1,516	238	0 (0%)	–	–	0 (0%)	–	–
CHC-COMP	LIA-XS	6	0	6	6 (100%)	37.3	4.7	6 (100%)	155×	73.4×	6 (100%)	892×	398×
CHC-COMP	LLReve	11	2	9	11 (100%)	1,375	283	2 (18.2%)	534×	282×	0 (0%)	–	–
CHC-COMP	RustHorn	10	2	8	10 (100%)	254	35.9	0 (0%)	–	–	0 (0%)	–	–
CHC-COMP	Synthesis	60	0	60	59 (98.3%)	10,149	4,078	2 (3.3%)	371×	202×	1 (1.7%)	398×	161×
	All	1,275	415	860	1,272 (99.8%)	1,583	472	410 (32.2%)	71.7×	28.2×	426 (33.4%)	116×	44.6×

Table 2. Reduced: count (percentage of row total). *hb*, *ms*: mean heartbeats and milliseconds over successful reductions. ×hb, ×ms: respective cost ratios to Zap on jointly reduced CHCs.

Zap_grind reduces only 410 constraints (32.2%); on shared successes with Zap, it consumes 71.7× as many heartbeats and takes 28.2× as long. Zap_aesop fares slightly better in coverage, reducing 426 constraints (33.4%), but consumes 116× as many heartbeats and takes 44.6× as long on its shared successes with Zap. Both variants produce proof terms 40% deeper than Zap's, taking 3.7× and 2.5× as long for the kernel to check, respectively (Table 4 in § D.1). These proof-term comparisons use geometric means over jointly reduced constraints and include generated helper proofs. Thus, Zap's direct proof construction achieves higher coverage and greater efficiency than search-based automation.

7.3 RQ3: Automated Proof Coverage

Refinement type checkers and CHC solvers traditionally rely on SMT for automation. We evaluate how much of this automation Flex retains as a *foundational* solver in Lean, using refinement-typechecker CHCs and CHC-COMP.

Experiment Design. We run three configurations of Flex separately: Zap + Fix, and the certifying pipeline with Spacer or Eldarica as its external solver. Fix progressively deepens scraping when synthesis fails (§6). All configurations use identical preprocessing and discharge the remaining goals with the same combination of `simp`, `grind`, `omega`, `aesop` and `lean-smt`. Each configuration has a 180 s budget. Each Spacer or Eldarica call is limited to 90 s. Tactic times exclude Lean start-up.

Refinement-Typechecker CHCs. We run each configuration on the 540 CHCs from `flux-tests`, `flux-medium`, `wave`, and `liquid-fix` introduced above. We exclude the case studies since their CHCs were never amenable to SMT automation.

Results. All three configurations retain most of the automation on these CHCs (Table 3). Zap + Fix fully proves 529 of the 540 CHCs (98.0%), Spacer 501 (92.8%), and Eldarica 512 (94.8%), each with a median tactic time below one second. Zap + Fix additionally makes partial progress on five of the remaining 11 CHCs, leaving behind fewer Horn variables. The 11 CHCs not proved by Zap + Fix encounter timeouts, failed invariant inference, or residual obligations that cannot be handled automatically such as nonlinear arithmetic. For Spacer and Eldarica, nine cases fail during

		Zap + Fix			Flex Spacer		Flex Eldarica	
Suite	CHCs	Full	Partial	Med. (s)	Full	Med. (s)	Full	Med. (s)
`flux-tests`	337	333 (98.8%)	2	0.05	310 (92.0%)	0.12	320 (95.0%)	0.78
`flux-medium`	96	94 (97.9%)	0	0.32	91 (94.8%)	0.23	91 (94.8%)	1.05
`wave`	36	35 (97.2%)	0	0.31	35 (97.2%)	0.27	36 (100%)	1.44
`liquid-fix`	71	67 (94.4%)	3	0.04	65 (91.5%)	0.12	65 (91.5%)	0.73
All	540	**529** (98.0%)	5	0.08	**501** (92.8%)	0.15	**512** (94.8%)	0.81

Table 3. Coverage on 540 refinement-typechecker CHCs: CHCs with a full proof (Full, with percentage) and the median tactic time over the CHCs each configuration fully proves (Med.). For Zap + Fix, Partial counts CHCs not fully proved in which some Horn variables are eliminated by Zap.

encoding, and one further case is not solved after translation. The remaining failures arise from timeouts, failed certification, or residual goals that Lean automation cannot discharge. Zap + Fix has the lowest median tactic time, about half of Spacer's and one tenth of Eldarica's. Over all 540 CHCs it also spends the least time in total, since both solvers spend most of their time on CHCs they do not prove. On the 491 CHCs that all three prove, however, Spacer is fastest in total, because a few CHCs take Zap + Fix much longer. Table 5 in § D.2 gives the full timing statistics.

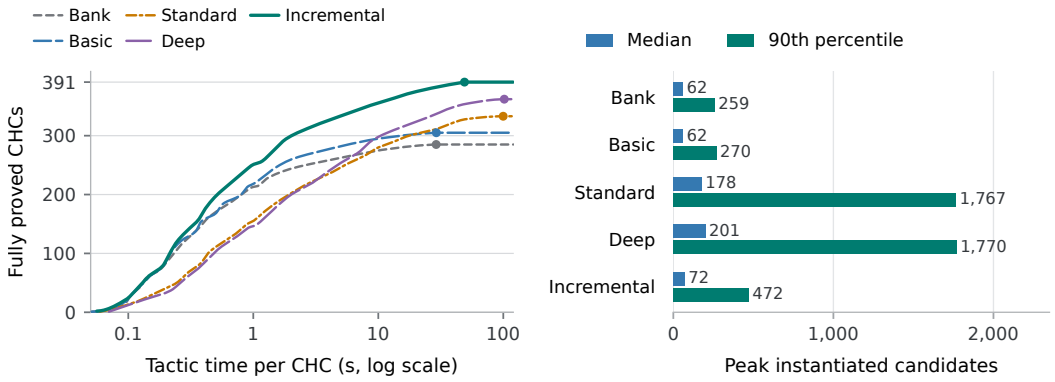
CHC-COMP. We run the same three configurations on the 1,559 satisfiable CHCs from CHC-COMP introduced above. As a baseline, we run Spacer and Eldarica natively on the original SMT-LIB inputs, with a 90 s limit each.

Results. Natively, Spacer solves 1,198 CHCs and Eldarica 937; together they solve 1,354 (86.9%; Table 6 in § D.2). Flex retains most of this automation with foundational proofs: Flex Spacer fully proves 983 of the 1,198 CHCs Spacer solves natively (82.1%), and Flex Eldarica 799 of the 937 Eldarica solves natively (85.3%). In total, Flex Spacer fully proves 997 CHCs (64.0%) and Flex Eldarica 822 (52.7%), with median tactic times of 2.9 s and 4.0 s. Together they prove 1,129 CHCs (72.4%): 1,118 of the 1,354 natively solved CHCs and 11 that neither native solver solves. Zap + Fix fully proves 221 CHCs (14.2%), with a median tactic time of 1.2 s, and makes partial progress on 654 others, almost all due to Zap eliminating Horn variables. Adding Zap + Fix to the two backends raises coverage to 1,133 CHCs. Spacer and Eldarica failures on CHC-COMP are analyzed in § D.2.

7.4 RQ4: Scraping Strategies

We evaluate how different scraping strategies affect Fix's solving time and candidate-pool size. Specifically, we ask two questions: (1) How does *incremental* scraping compare with applying a fixed scraping level directly, in terms of solving time and candidate-pool size? (2) How do solving time and candidate-pool size change across the fixed scraping levels?

Experimental Setup. We select 391 CHCs: 181 from the refinement-typechecker suites, 192 from CHC-COMP, and 18 from RQ1. Each selected CHC contains at least one *cyclic* Horn variable and was fully proved using only Zap and Fix with the default incremental scraping strategy. The cyclic-variable requirement ensures that Fix is used, and thus, allows us to perform ablation across different strategies. We evaluate four fixed scraping levels (§C). *Bank* uses 19 built-in templates; *Basic* adds predicates from CHC heads. *Standard* also scrapes premises and directly referenced Lean definitions and builds guarded arithmetic qualifiers. *Deep* explores helper definitions and adds generalizations such as quantified bounds and accumulator relations. We compare these with our default, *Incremental*, which tries the levels in order, retaining earlier candidates and advancing only when no invariant is accepted. Each run applies Zap followed by Fix, with a 60 s budget for Zap



(a) CHCs proved within each time budget.

(b) Candidate-pool sizes: median and 90th percentile.

Fig. 12. Scraping configurations on 391 selected CHCs. Coverage curves use monotone interpolation; dots mark each configuration's last observed proof. Candidate counts include unsuccessful runs.

and a 120 s budget shared across Fix's scraping levels. We report the number of fully proved CHCs, tactic time on those CHCs, and peak candidate-pool size. For each run, the latter is the maximum over PA stages of the initial candidate count summed across predicates (§ D.3).

Results. Compared with applying Deep directly, Incremental uses smaller candidate pools and reduces solving time on jointly proved CHCs (Fig. 12). Its median and 90th-percentile peak candidate-pool sizes are 72 and 472, versus 201 and 1,770 for Deep, with a geometric-mean speedup of 2.70× on shared successes. Incremental proves all 391 selected CHCs, compared with 285 for Bank and 362 for Deep; its median and 90th-percentile tactic times are 0.50 s and 8.82 s. Across the fixed levels, deeper scraping increases candidate-pool size and coverage, accompanied by higher observed median solving times. For Bank, Basic, Standard, and Deep, median peak candidate-pool sizes are 62, 62, 178, and 201, respectively, while the numbers of fully proved CHCs rise from 285 to 305, 333, and 362. Median tactic times are 0.37 s, 0.38 s, 1.16 s, and 1.56 s, respectively. Timing medians use each configuration's own successful CHCs. Table 9 in § D.3 reports the full timing distributions.

8 Related Work

CHC Solvers Flex is inspired by [14] to solve acyclic variables, and predicate abstraction [23] to compute (overapproximate) solutions for cyclic ones. Zap additionally constructs exact proof terms certifying acyclic *Horn* elimination. Spacer [36, 56], Eldarica [33], and SeaHorn [28] solve CHCs by inferring invariants using IC3 [8] or Lazy Abstraction and Craig Interpolation [31, 43], which do not require qualifiers, but which may diverge. A direction for future work is *proof-producing* IC3-style or interpolation-based solving over native Lean definitions. Several other SMT-based CHC solvers have been developed [6, 11, 18]. To our knowledge, Flex is the first CHC solver to operate directly over native ITP propositions while producing kernel-checkable proofs.

Proving or Certifying VCs Work on proof-carrying code showed how to avoid trusting SMT solvers by modifying decision procedures to emit certificates [46]. This idea was extended to model checkers [30], and refinement types [12], and several modern theorem provers emit independently checkable certificates [3, 4, 7, 20, 44]. Alternatively, interactive proof assistants can discharge VCs: Jahob [58] delegates VCs to backends including Isabelle, while the Creusot Rust verifier [17]

emits VCs in the Why3 format for SMT or interactive proof [21]. These techniques apply to plain VCs; Flex uses such provers as *oracles* to obtain certificates for CHC satisfiability.

SMT-based Program Verifiers Program-logic-based verifiers [19, 25, 32] typically emit Horn-variable-free VCs discharged by SMT solvers [15, 24, 39, 41]. Prusti [1] and Verus [37] implement such Floyd-Hoare style verifiers for Rust. F* generalizes the approach to the higher order setting using Dijkstra monads [53], while Stainless [29] uses refinement types. In contrast to the above, LiquidHaskell [57], Flux [38] and Thrust [49] use CHCs, which generalize VCs with existentially quantified variables that delegate the synthesis of refinements or invariants to an SMT-based CHC solver. When incomplete SMT heuristics [40] are opaque or hard to use [59], Flex supports interactive proof while providing foundational guarantees of CHC satisfiability.

AI Usage

The project has been ongoing for more than a year. The high-level architecture of Flex, the designs of Zap and predicate abstraction, and the evaluation were human-led. Over time, we used AI models, including Opus, to assist with implementation and mechanization, develop evaluation harnesses, and identify performance bottlenecks.

Data-Availability Statement

All Lean code for Flex is available in the supplementary material. Rust case studies, benchmarks, and evaluation setup will be submitted with artifact evaluation.

References

[1] Vytautas Astrauskas, Aurea Bílá, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. 2022. The Prusti Project: Formal Verification for Rust. In *NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, Proceedings* (Pasadena, CA, USA). Springer-Verlag, Berlin, Heidelberg, 88–108. doi:10.1007/978-3-031-06773-0_5

[2] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 415–442.

[3] Haniel Barbosa, Clark Barrett, Byron Cook, Bruno Dutertre, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Cesare Tinelli, and Yoni Zohar. 2023. Generating and Exploiting Automated Reasoning Proof Certificates. *Commun. ACM* 66, 10 (Sept. 2023), 86–95. doi:10.1145/3587692

[4] Filip Bártek, Ahmed Bhayat, Robin Coutelier, Márton Hajdu, Matthias Hetzenberger, Petra Hozzová, Laura Kovács, Jakob Rath, Michael Rawson, Giles Reger, Martin Suda, Johannes Schoisswohl, and Andrei Voronkov. 2025. The Vampire Diary. arXiv:2506.03030 [cs.LO] https://arxiv.org/abs/2506.03030

[5] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation II*. Springer, 24–51. doi:10.1007/978-3-319-23534-9_2

[6] Martin Blicha, Konstantin Britikov, and Natasha Sharygina. 2025. Golem: a flexible and efficient solver for constrained Horn clauses. *Formal Methods in System Design* 67 (2025), 143–160. doi:10.1007/s10703-025-00470-9

[7] Jonas Bodingbauer, Márton Hajdu, Laura Kovács, Axel Polaczek, and Michael Rawson. 2026. Lean on Vampire Proofs (Short Paper). arXiv:2603.26342 [cs.LO] https://arxiv.org/abs/2603.26342

[8] Aaron R Bradley. 2011. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Springer, 70–87. doi:10.1007/978-3-642-18275-4_7

[9] Konstantin Britikov and Levente Bajczi. 2026. CHC-COMP 2026: Constrained Horn Clause Solver Competition. https://chc-comp.github.io/chc-comp-2026/. Results presented at HCVS 2026. Accessed: 2026-11-11.

[10] Mario Carneiro. 2025. Lean4Lean: Verifying a Typechecker for Lean, in Lean. arXiv:2403.14064 [cs.PL] https://arxiv.org/abs/2403.14064

[11] Adrien Champion, Naoki Kobayashi, and Ryosuke Sato. 2018. HoIce: An ICE-Based Non-linear Horn Clause Solver. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Proceedings (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics))*, Sukyoung Ryu (Ed.). Springer Verlag, Germany, 146–156. doi:10.1007/978-3-030-02768-1_8 Publisher Copyright: © 2018, Springer

Nature Switzerland AG.; 16th Asian Symposium on Programming Languages and Systems, APLAS 2018 ; Conference date: 02-12-2018 Through 06-12-2018.

[12] Juan Chen, Ravi Chugh, and Nikhil Swamy. 2010. Type-preserving compilation of end-to-end verification of security enforcement. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, Benjamin G. Zorn and Alex Aiken (Eds.). ACM, 412–423. doi:10.1145/1806596.1806643

[13] Byron Cook, Remi Delmas, Zyad Hassan, Bart Jacobs, Ranjit Jhala, Rahul Kumar, Felipe R. Monteiro, Thanh Nguyen, Rebecca Rumbul, Michael Tautschnig, Celina Val, and Carolyn Zech. 2026. Verifying the Rust Standard Library. In *NASA Formal Methods: 18th International Symposium, NFM 2026, Los Angeles, CA, USA, May 5–7, 2026, Proceedings* (Los Angeles, CA, USA). Springer-Verlag, Berlin, Heidelberg, 415–435. doi:10.1007/978-3-032-28079-4_19

[14] Benjamin Cosman and Ranjit Jhala. 2017. Local refinement typing. *Proc. ACM Program. Lang.* 1, ICFP, Article 26 (Aug. 2017), 27 pages. doi:10.1145/3110270

[15] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C: a software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods* (Thessaloniki, Greece) *(SEFM'12)*. Springer-Verlag, Berlin, Heidelberg, 233–247. doi:10.1007/978-3-642-33826-7_16

[16] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[17] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering: 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24–27, 2022, Proceedings* (Madrid, Spain). Springer-Verlag, Berlin, Heidelberg, 90–105. doi:10.1007/978-3-031-17244-1_6

[18] Daniel Dietsch, Matthias Heizmann, Jochen Hoenicke, Alexander Nutz, and Andreas Podelski. 2019. Ultimate TreeAutomizer (CHC-COMP Tool Description). *Electronic Proceedings in Theoretical Computer Science* 296 (July 2019), 42–47. doi:10.4204/eptcs.296.7

[19] Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J.

[20] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. 2017. SMTCoq: A plug-in for integrating SMT solvers into Coq. In *Computer Aided Verification - 29th International Conference*. Heidelberg, Germany. https://hal.science/hal-01669345

[21] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3: where programs meet provers. In *Proceedings of the 22nd European Conference on Programming Languages and Systems* (Rome, Italy) *(ESOP'13)*. Springer-Verlag, Berlin, Heidelberg, 125–128. doi:10.1007/978-3-642-37036-6_8

[22] Cormac Flanagan. 2004. Automatic software model checking via constraint logic. *Science of Computer Programming* 50, 1-3 (2004), 253–270.

[23] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2021)*, José Nuno Oliveira and Pamela Zave (Eds.). Springer, 500–517. doi:10.1007/3-540-45251-6_29

[24] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 234–245. doi:10.1145/512529.512558

[25] Robert W. Floyd. 1967. Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science*, J. T. Schwartz (Ed.), Vol. 19. American Mathematical Society, 19–32.

[26] Vladimir Gladshtein, George Pîrlea, Qiyuan Zhao, Vitaly Kurin, and Ilya Sergey. 2026. Foundational Multi-Modal Program Verifiers. *Proc. ACM Program. Lang.* 10, POPL, Article 77 (Jan. 2026), 32 pages. doi:10.1145/3776719

[27] Susanne Graf and Hassen Saidi. 1997. Construction of abstract state graphs with PVS. In *Computer Aided Verification*, Orna Grumberg (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 72–83.

[28] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *International Conference on Computer Aided Verification*. 343–361. doi:10.1007/978-3-319-21690-4_20

[29] Jad Hamza, Nicolas Voirol, and Viktor Kunčak. 2019. System FR: formalized foundations for the stainless verifier. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 166 (Oct. 2019), 30 pages. doi:10.1145/3360592

[30] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Grégoire Sutre, and Westley Weimer. 2002. Temporal-Safety Proofs for Systems Code. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02)*. Springer-Verlag, Berlin, Heidelberg, 526–538.

[31] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL '02)*. Association for Computing Machinery, New York, NY, USA, 58–70. doi:10.1145/503272.503279

[32] Charles Anthony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[33] Hossein Hojjat and Philipp Rümmer. 2018. The ELDARICA Horn Solver. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. 1–7. doi:10.23919/FMCAD.2018.8603013

[34] Evan Johnson, Evan Laufer, Zijie Zhao, Dan Gohman, Shravan Narayan, Stefan Savage, Deian Stefan, and Fraser Brown. 2023. WaVe: a verifiably secure WebAssembly sandboxing runtime. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2940–2955.

[35] Richard M. Karp. 1972. Reducibility Among Combinatorial Problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA (The IBM Research Symposia Series)*, Raymond E. Miller and James W. Thatcher (Eds.). Plenum Press, New York, 85–103. doi:10.1007/978-1-4684-2001-2_9

[36] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based model checking for recursive programs. *Form. Methods Syst. Des.* 48, 3 (June 2016), 175–205. doi:10.1007/s10703-016-0249-4

[37] 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, Article 85 (April 2023), 30 pages. doi:10.1145/3586037

[38] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. *Proc. ACM Program. Lang.* 7, PLDI, Article 169 (June 2023), 25 pages. doi:10.1145/3591283

[39] K. Rustan M. Leino. 2010. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (Dakar, Senegal) *(LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370.

[40] K. Rustan M. Leino and Clément Pit-Claudel. 2016. Trigger Selection Strategies to Stabilize Program Verifiers. 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)*. Springer, 361–381. doi:10.1007/978-3-319-41528-4_20

[41] D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. 1979. *Stanford Pascal Verifier user manual.* Technical Report STAN-CS-79-731. Stanford University, Department of Computer Science.

[42] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-Based Verification for Rust Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43, 4 (2021), 1–47. doi:10.1145/3462205

[43] Kenneth L McMillan. 2006. Lazy abstraction with interpolants. In *International Conference on Computer Aided Verification (CAV)*. Springer, 123–136. doi:10.1007/11817963_14

[44] Abdalrhman Mohamed, Tomaz Mascarenhas, Harun Khan, Haniel Barbosa, Andrew Reynolds, Yicheng Qian, Cesare Tinelli, and Clark Barrett. 2025. lean-smt: An SMT Tactic for Discharging Proof Goals in Lean. In *Computer Aided Verification*, Ruzica Piskac and Zvonimir Rakamarić (Eds.). Springer Nature Switzerland, Cham, 197–212.

[45] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 625–635. doi:10.1007/978-3-030-79876-5_37

[46] George Ciprian Necula. 1998. *Compiling with Proofs.* Ph. D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, USA. https://www.cs.cmu.edu/~rwh/students/necula.pdf Available as Technical Report CMU-CS-98-154.

[47] 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

[48] Ulf Norell. 2009. *Dependently Typed Programming in Agda.* Springer Berlin Heidelberg, Berlin, Heidelberg, 230–266. doi:10.1007/978-3-642-04652-0_5

[49] Hiromi Ogawa, Taro Sekiyama, and Hiroshi Unno. 2025. Thrust: A Prophecy-Based Refinement Type System for Rust. *Proc. ACM Program. Lang.* 9, PLDI, Article 230 (June 2025), 25 pages. doi:10.1145/3729333

[50] Andreas Podelski and Andrey Rybalchenko. 2007. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In *Practical Aspects of Declarative Languages (PADL) (Lecture Notes in Computer Science, Vol. 4354)*. Springer, 245–259. doi:10.1007/978-3-540-69611-7_16

[51] Patrick M. Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 159–169. doi:10.1145/1375581.1375602

[52] Leon Schuermann, Brad Campbell, Branden Ghena, Philip Levis, Amit Levy, and Pat Pannuto. 2025. Tock: From Research to Securing 10 Million Computers. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 36–49.

[53] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the dijkstra monad. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 387–398. doi:10.1145/2491956.2491978

[54] Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160. doi:10.1137/0201010

[55] The Coq Development Team. 2024. The Coq Reference Manual – Release 8.19.0. https://coq.inria.fr/doc/V8.19.0/refman.

[56] Takeshi Tsukada and Hiroshi Unno. 2024. Inductive Approach to Spacer. *Proc. ACM Program. Lang.* 8, PLDI, Article 227 (June 2024), 24 pages. doi:10.1145/3656457

[57] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 269–282. doi:10.1145/2628136.2628161

[58] Karen Zee, Viktor Kuncak, and Martin Rinard. 2008. Full functional verification of linked data structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 349–361. doi:10.1145/1375581.1375624

[59] Yi Zhou, Jay Bosamiya, Yoshiki Takashima, Jessica Li, Marijn Heule, and Bryan Parno. 2023. Mariposa: Measuring SMT Instability in Automated Program Verification. In *2023 Formal Methods in Computer-Aided Design (FMCAD)*. 178–188. doi:10.34727/2023/isbn.978-3-85448-060-0_26

A Zap: Soundness

This appendix proves the correctness of the Zap algorithm of §4, whose two results are *Soundness* (Theorem 4.1) and *Equivalence* (Theorem A.11).

Overview Equivalence, together with the fact that Sol computes the strongest valid solution, is the Fusion result of Cosman and Jhala [14]. Soundness we prove as a pure typing result: the proof term q emitted by $\mathsf{Zap}(C)$ is a closed, well-typed term of type $C' \to C$, so the Lean kernel turns any proof of the residual C' into a proof of the original C.

Setup Throughout the appendix, we fix a closed constraint

$$C \;\equiv\; \exists \overline{\kappa}.\, c,$$

whose Horn variables $\overline{\kappa}$ are partitioned (§ 3.3) into the cut set $\overline{\kappa_C}$ and the acyclic set $\overline{\kappa_A}$.

Organization

- § A.1 fixes the conventions and the proof-term calculus;
- § A.2 establishes the shape of the synthesized solution σ;
- § A.3 characterizes Red and the elimination order;
- § A.4 types Nav, $\mathsf{Cert}^{\kappa}_{\sigma,\rho}$, and the emitted bridge.

A.1 Conventions

Definitions and uses A κ-*definition* of a constraint c is a head occurrence of κ, i.e., a sub-constraint $c\!\downarrow\!\rho = \kappa(\bar{t})$ for some head-prefix ρ; a κ-*use* is an occurrence of κ in a guard. In input constraints, guards are atoms (Fig. 6), so every use is a single application $\kappa(\bar{t})$ occurring negatively, and Horn variables occur nowhere else (§ 3.1); in particular they never occur inside argument terms.

Validity Validity is Lean provability: we write $\Gamma \vdash \varphi$ for derivability under a context and use the connectives' introduction and elimination laws without comment. The judgment reads the binders and hypotheses of Γ and ignores the routing marks.

Scope, contexts, truncation For an acyclic κ in c, let $\rho = \mathsf{Scope}(\kappa, c)$ (Fig. 7). We use the following facts.

- $c\!\downarrow\!\rho$ is the sub-constraint at the lowest common ancestor (LCA) of κ's occurrences, and $c\!\uparrow\!\rho$ is the context above it.
- A context (§ 3.2) lists binders $x{:}b$, hypotheses $h{:}p$, and marks L/R; each entry *matches* the prefix step that produced it: $x{:}b$ matches B, $h{:}p$ matches G, and a mark matches itself.
- Given a context Γ along a path extending ρ, the *truncation* $\Gamma \setminus \rho$ drops the first $|\rho|$ entries; this is the context $\mathsf{Cert}^{\kappa}_{\sigma,\rho}$ hands Nav at a divergent head (Fig. 10).
- By well-formedness (§ 3.2), every application of κ has $\mathsf{rank}(\kappa)$ leading arguments followed by the binders $\overline{x} = \mathsf{dom}(c\!\uparrow\!\rho)$, innermost first, as trailing arguments.

Typing the proof-term fragment Fig. 13 fixes the typing judgment $\Gamma \vdash q : \varphi$ for the proof terms of Fig. 6: the introduction and elimination forms of the constraint logic. The following conventions apply:

(1) In the $\exists$-introduction and $\exists$-elimination rules, the sort b annotating the existential binder $\exists z{:}b.\, \varphi$ ranges over base sorts and predicate types $b_1 \to \cdots \to b_n \to \mathit{Prop}$.
(2) A bundle $\langle t_1, \ldots, t_k;\, q\rangle$ abbreviates iterated $\exists$-introduction.
(3) A multi-binder $\mathsf{let}\ \langle z_1, \ldots, z_k, h\rangle = q\ \mathsf{in}\ q'$ abbreviates iterated $\exists$-elimination: k nested **let**s that bind the witnesses $z_1, \ldots, z_k$ one at a time, with h naming the proof of the fully unpacked body.
(4) The premise $\Gamma \vdash t : b$ is Lean's term-typing judgment.

$$\frac{x{:}b \in \Gamma}{\Gamma \vdash x : b} \qquad \frac{h{:}\varphi \in \Gamma}{\Gamma \vdash h : \varphi} \qquad \frac{\Gamma, x{:}b \vdash q : \varphi}{\Gamma \vdash \lambda x.q : \forall x{:}b.\, \varphi} \qquad \frac{\Gamma \vdash q : \forall x{:}b.\, \varphi \qquad \Gamma \vdash t : b}{\Gamma \vdash q\,t : \varphi[x := t]}$$

$$\frac{\Gamma, h{:}g \vdash q : \varphi}{\Gamma \vdash \lambda h.q : g \to \varphi} \qquad \frac{\Gamma \vdash q : g \to \varphi \qquad \Gamma \vdash q' : g}{\Gamma \vdash q\,q' : \varphi} \qquad \frac{\Gamma \vdash q : \varphi_1 \qquad \Gamma \vdash q' : \varphi_2}{\Gamma \vdash \langle q, q' \rangle : \varphi_1 \wedge \varphi_2}$$

$$\frac{\Gamma \vdash q : \varphi_1 \wedge \varphi_2}{\Gamma \vdash q.i : \varphi_i}\ (i \in \{1,2\}) \qquad \frac{\Gamma \vdash q : \varphi_1}{\Gamma \vdash \mathsf{inl}\, q : \varphi_1 \vee \varphi_2} \qquad \frac{\Gamma \vdash q : \varphi_2}{\Gamma \vdash \mathsf{inr}\, q : \varphi_1 \vee \varphi_2}$$

$$\frac{\Gamma \vdash t : b \qquad \Gamma \vdash q : \varphi[z := t]}{\Gamma \vdash \langle t, q \rangle : \exists z{:}b.\, \varphi} \qquad \frac{\Gamma \vdash q : \exists z{:}b.\, \varphi \qquad \Gamma, z{:}b, h{:}\varphi \vdash q' : \psi}{\Gamma \vdash \mathsf{let}\ \langle z, h \rangle = q\ \mathsf{in}\ q' : \psi}\ (z, h \notin \mathrm{FV}(\psi))$$

$$\frac{}{\Gamma \vdash \mathsf{rfl} : t = t} \qquad \frac{}{\Gamma \vdash \langle\rangle : \top}$$

Fig. 13. Typing of the proof-term fragment of Fig. 6.

Sol	:	$(\mathcal{K} \times C) \to \mathcal{T}$
$\mathrm{Sol}(\kappa, c)$		
$\quad \mid\ \kappa \notin \mathrm{heads}(c)$	$\doteq$	$\bot$
$\mathrm{Sol}(\kappa, c_1 \wedge c_2)$	$\doteq$	$\mathrm{Sol}(\kappa, c_1) \vee \mathrm{Sol}(\kappa, c_2)$
$\mathrm{Sol}(\kappa, \forall x{:}b.\, c)$	$\doteq$	$\exists x{:}b.\, \mathrm{Sol}(\kappa, c)$
$\mathrm{Sol}(\kappa, p \to c)$	$\doteq$	$p \wedge \mathrm{Sol}(\kappa, c)$
$\mathrm{Sol}(\kappa, \kappa(\overline{t}))$	$\doteq$	$\bigwedge_{i<\mathrm{rank}(\kappa)} z_i = t_i$

Fig. 14. The Sol procedure, reproduced from Fig. 8.

These rules are not the Lean kernel itself but our model of it, restricted to this fragment and following the kernel formalization of Carneiro [10]; every rule is admissible in the kernel, so a term well-typed here is accepted by the kernel, and Soundness is proved against this system.

A.2 The synthesized solution

Fix an acyclic κ in a constraint c, and let $\rho = \mathrm{Scope}(\kappa, c)$. For this κ, Elim (Fig. 8) computes the solution using Sol as in Fig. 14, with the slot names $\overline{z}$ fresh for the binders of c:

$$\sigma \;=\; \lambda\overline{z}.\lambda\overline{x}.\mathrm{Sol}(\kappa, c{\downarrow}\rho), \qquad \overline{z} = z_0, \ldots, z_{\mathrm{rank}(\kappa)-1}, \quad \overline{x} = \mathrm{dom}(c{\uparrow}\rho).$$

The soundness proof needs two properties of σ:

(1) σ must be κ-free: it is substituted at the uses of κ and supplied as the witness for $\exists\kappa$.
(2) The body of σ must mirror the structure of $c{\downarrow}\rho$: along every path to a κ-definition it is the same tree with each connective dualized, so Nav can walk both in lockstep.

The next lemma establishes both.

Lemma A.1 (Solution shape)**.**

(a) $\mathrm{Sol}(\kappa, c^*) = \bot$ *for every sub-constraint* c^* *of* $c{\downarrow}\rho$ *containing no* κ*-definition.*

(b) On a path from the root of $c \downarrow \rho$ to a κ-definition, Sol *dualizes each connective, preserving binder names, and emits the slot equalities at the definition:*

$$\begin{aligned} c_1^* \wedge c_2^* &\mapsto \mathsf{Sol}(\kappa, c_1^*) \vee \mathsf{Sol}(\kappa, c_2^*) \\ \forall x{:}b.\, c^* &\mapsto \exists x{:}b.\, \mathsf{Sol}(\kappa, c^*) \\ g \to c^* &\mapsto g \wedge \mathsf{Sol}(\kappa, c^*) \\ \kappa(\overline{t}) &\mapsto \textstyle\bigwedge_{i<\mathsf{rank}(\kappa)} z_i = t_i \end{aligned}$$

(c) $\mathrm{FV}(\sigma) \subseteq \mathsf{KVars}(c) \setminus \{\kappa\}$*; in particular,* σ *never mentions* κ*.*

Proof.

(a) Immediate from the guarded equation of Fig. 14: $\kappa \notin \mathrm{heads}(c^*)$ (§ 3.2) holds exactly when c^* contains no κ-definition.

(b) Each node on the path has a κ-definition in its subtree, so the guarded equation does not apply: at each node above the definition, the equation matching the connective applies and copies its binder or guard verbatim; at the definition itself, the head equation emits the slot equalities.

(c) By induction on the sub-constraints c^* of $c \downarrow \rho$, following the equations of Fig. 14,

$$\mathrm{FV}(\mathsf{Sol}(\kappa, c^*)) \;\subseteq\; (\mathrm{FV}(c^*) \setminus \{\kappa\}) \cup \overline{z}.$$

- The guarded equation emits $\bot$, with no free variables.
- The $\wedge$ and $\forall$ equations preserve the binding structure: the disjunction takes the same union, and the $\exists$ binds the same x.
- The guard equation copies g, and $\kappa \notin \mathrm{FV}(g)$: a κ-use above a κ-definition is a self-edge at κ in $\mathsf{Graph}(c)$ (§ 3.3), so every cut set would contain κ, contradicting that κ is acyclic.
- The head equation emits the slots $\overline{z}$ and the arguments $\overline{t}$, which never contain Horn variables (§ A.1).

Since C is closed, every free variable of $c \downarrow \rho$ is a Horn variable or a binder of $c \uparrow \rho$, i.e., among $\overline{x}$; the $\lambda\overline{z}.\lambda\overline{x}.\cdot$ closure of Elim then leaves free only Horn variables other than κ.

□

A.3 Reduction and the elimination order

We write $c[\kappa := \sigma]$ for the substitution of σ for every occurrence of κ in c.

Substituting a synthesized solution can produce compound guards containing cut variables. We admit these formulas as residual guards and count their Horn-variable occurrences when computing dependencies. Sol copies each guard as a whole, and Nav uses the corresponding hypothesis.

Lemma A.2 (Reduction)**.** $\mathsf{Red}(\kappa, \sigma, c)$ *(Fig. 8) is* κ*-free and coincides with* $c[\kappa := \sigma]$*, except that each definition head* $\kappa(\overline{t})$ *becomes* $\top$ *rather than* $\sigma(\overline{t})$*.*

Proof. By structural induction on c, comparing each equation of Fig. 8 with the substitution:

- at $c_1 \wedge c_2$ and $\forall x{:}b.\, c^*$, both sides recurse componentwise; at $p \to c^*$, both also substitute the guard p identically;
- at a definition head $\kappa(\overline{t})$, Red gives $\top$ and the substitution gives $\sigma(\overline{t})$: the sole divergence;
- at any other head, both are the identity, since κ occurs neither in a κ-free atom nor inside the arguments of a foreign application $\kappa'(\overline{s})$ (§ A.1).

The result is κ-free: heads become $\top$, and uses are replaced by σ, which is κ-free (Lemma A.1). □

The fold of Fig. 8 eliminates the acyclic variables innermost first: writing $\overline{\kappa_A} = [\kappa_1, \ldots, \kappa_n]$ in elimination order, $c_0 = c$ and $c_i = \mathsf{Red}(\kappa_i, \sigma_i, c_{i-1})$, with σ_i synthesized on the residual c_{i-1}. The

partition orders the acyclic set topologically (§ 3.3): every variable that κ_i depends on is some κ_j with $j < i$ or a cut variable.

Lemma A.3 (Solutions are cut-only). *For each i, $\mathrm{FV}(\sigma_i) \subseteq \overline{\kappa_C}$.*

Proof. By induction on i: assuming the lemma for all $j < i$, we prove $\mathrm{FV}(\sigma_i) \subseteq \overline{\kappa_C}$ in two steps:

- *κ_i is acyclic in c_{i-1}.* Red preserves the skeleton, and the spliced σ_j ($j < i$) are cut-only by induction; so every edge of $\mathrm{Graph}(c_{i-1})$ out of an acyclic variable is already in $\mathrm{Graph}(c)$. Hence $\overline{\kappa_C}$ remains a cut set of c_{i-1}, κ_i remains acyclic, and Lemma A.1 applies at step i.
- *No acyclic variable is free in σ_i.* By Lemma A.1(c), the free variables of σ_i are Horn variables of c_{i-1}; by (b), each occurs in a guard above a κ_i-definition. An acyclic one is impossible: its use is inherited from c (first step), so κ_i depends on it in c and, by topological order, it is some κ_j with $j < i$; but c_{i-1} is κ_j-free (Lemma A.2; the splices are cut-only). Hence all are cut variables.

□

A.4 Soundness

Finally, we bring our attention to Soundness (Theorem 4.1) for Zap. It is stated under the typing relation, shown in Fig. 13 for the fragment of Lean proof terms in Fig. 6.

$\mathsf{Cert}^{\kappa}_{\sigma,\rho}$ accumulates one context entry per node it descends past, whereas $\rho = \mathrm{Scope}(\kappa, c)$ was computed on c up front. The next lemma aligns the two at every κ-definition: truncating ρ from the accumulated context recovers the definition's address inside $c{\downarrow}\rho$, the directions Nav walks by.

Lemma A.4 (Prefix correspondence). *Let $\rho = \mathrm{Scope}(\kappa, c)$. If the root call $\mathsf{Cert}^{\kappa}_{\sigma,\rho}(\varepsilon, h, c)$ (Fig. 10) reaches $\mathsf{Cert}^{\kappa}_{\sigma,\rho}(\Gamma, q, \kappa(\bar{t}))$, then $\kappa(\bar{t}) = c{\downarrow}\rho \cdot \pi$ for some path π, and*

(a) Γ matches $\rho \cdot \pi$;
(b) $\Gamma \setminus \rho$ matches π.

Proof. By induction on the recursion of $\mathsf{Cert}^{\kappa}_{\sigma,\rho}$ (Fig. 10), every call $\mathsf{Cert}^{\kappa}_{\sigma,\rho}(\Gamma, q, c^*)$ reached from $\mathsf{Cert}^{\kappa}_{\sigma,\rho}(\varepsilon, h, c)$ satisfies the invariant: $c^* = c{\downarrow}\tau$ for some path τ, and Γ matches τ. At the root, $\tau = \Gamma = \varepsilon$; each recursive equation extends τ by one step and Γ by the matching entry:

- at $\forall x{:}b.\, c_0$: the step B, the entry $x{:}b$;
- at $p \to c_0$: the step G, the entry $h{:}p$;
- at $c_1 \wedge c_2$: the mark L for c_1 and R for c_2, as both step and entry.

At $c^* = \kappa(\bar{t})$, this κ-definition lies inside $c{\downarrow}\rho$ (§ A.1), so τ extends ρ: $\tau = \rho \cdot \pi$ for some path π. The invariant at this call is (a), and dropping the first $|\rho|$ entries gives (b). □

When Red rewrites a definition head $\kappa(\bar{t})$ to $\top$, Nav supplies the proof that justifies it: a proof of the applied solution $\sigma(\bar{t})$. The next lemma states that the term Nav builds under the truncated context $\Gamma \setminus \rho$ is indeed well-typed (valid proof) at $\sigma(\bar{t})$.

Lemma A.5 (Nav). *Let π be a path with $c{\downarrow}\rho \cdot \pi = \kappa(\bar{t})$ and $\Gamma = c{\uparrow}\rho \cdot \pi$. Then*

$$\Gamma \;\vdash\; \mathsf{Nav}(\Gamma \setminus \rho, \sigma(\bar{t})) \;:\; \sigma(\bar{t}).$$

Proof. The β-reduction on $\sigma(\bar{t})$ performs substitution on the leading arguments among $\bar{t}$ by the params $\bar{z}$. By well-formedness, we are left with the binders $\bar{x}$, each of x_i gets replaced by the corresponding $\forall x_i$ in the context. Finally, we get fully reduced σ in the head.

By Lemma A.1(b),

(1) along the branch that π addresses, the β-reduced $\sigma(\bar{t})$ is the dual tree of $c{\downarrow}\rho$;
(2) each $\exists$-binder on that branch is named after a binder of Γ.

Since the substituted terms also mention binders of Γ, the reduction is capture-avoiding: it renames each $\exists$-binder on the walked branch to a fresh name. Nav emits no binders: only pairs, injections, witnesses, and hypothesis references. The context therefore never grows, and every judgment below is under the same Γ.

By induction on the suffix Γ' of $\Gamma \setminus \rho$, we prove

$$\Gamma \;\vdash\; \mathsf{Nav}(\Gamma', p) \;:\; p,$$

where p is the subtree of the β-reduced $\sigma(\bar{t})$ that Γ' addresses: $\Gamma = c{\uparrow}\rho \cdot \pi$ matches $\rho \cdot \pi$ entry by entry (§ 3.2), so $\Gamma \setminus \rho$ matches π. The lemma is the instance $\Gamma' = \Gamma \setminus \rho$. Each Nav equation (Fig. 10) is matched by one typing rule.

- *Case* $\Gamma' = \mathsf{L}; \Gamma''$ at $p_L \vee p_R$: the branch continues in p_L. The induction hypothesis certifies p_L, and $\vee$-introduction with inl gives the disjunction; the untaken p_R appears only in the type.
- *Case* $\Gamma' = \mathsf{R}; \Gamma''$: by symmetry.
- *Case* $\Gamma' = x : b; \Gamma''$ at $\exists z : b.\, p$: z is the fresh rename of the walk binder x, and the witness substitution $p[z := x]$ (Fig. 10) undoes the renaming; on the walked branch the two substitutions compose to the identity. The induction hypothesis certifies the body, and $\exists$-introduction with the witness $x \in \Gamma$ gives the existential.
- *Case* $\Gamma' = h : g; \Gamma''$ at $g \wedge p$: the substitutions leave g unchanged (the slots are fresh, each x_i replaced itself, the renamings above were undone), so the first conjunct is syntactically the guard named by h: $h : g \in \Gamma$. $\wedge$-introduction pairs h with the recursive certificate.
- *Case* $\Gamma' = \varepsilon$ at $\bigwedge_{i<\mathsf{rank}(\kappa)} z_i = t_i$: this leaf belongs to the definition that supplied $\bar{t}$, so every conjunct reads $t_i = t_i$, and $\langle \mathsf{rfl}, \ldots, \mathsf{rfl} \rangle$ types it. When $\mathsf{rank}(\kappa) = 0$, the empty conjunction is $\top$ and $\langle\rangle$ does.

□

We write $\Gamma[\kappa := \sigma]$ for the context Γ with σ substituted in each hypothesis type; binders and marks are unchanged.

Lemma A.6 ($\mathsf{Cert}^{\kappa}_{\sigma,\rho}$). *For every prefix π, let $c^* = c{\downarrow}\pi$ and $\Gamma = c{\uparrow}\pi$. If $\Gamma[\kappa := \sigma] \vdash q : \mathsf{Red}(\kappa, \sigma, c^*)$, then*

$$\Gamma[\kappa := \sigma] \;\vdash\; \mathsf{Cert}^{\kappa}_{\sigma,\rho}(\Gamma, q, c^*) \;:\; c^*[\kappa := \sigma].$$

Proof. By induction on c^*, pairing each equation of Fig. 10 with the matching Red equation (Lemma A.2) and one typing rule.

- *Case* $\forall x : b.\, c_0$: both operators pass the quantifier through. Under $\Gamma; x : b$, the term $q\,x$ proves $\mathsf{Red}(\kappa, \sigma, c_0)$. The induction hypothesis and $\forall$-introduction give $\lambda x.\cdot$.
- *Case* $g \to c_0$: both operators substitute the guard to $g[\kappa := \sigma]$. Given $h : g[\kappa := \sigma]$, the term $q\,h$ proves $\mathsf{Red}(\kappa, \sigma, c_0)$. The induction hypothesis and $\to$-introduction give $\lambda h.\cdot$.
- *Case* $c_1 \wedge c_2$: the context extends by L, and $q.1$ proves the reduced c_1, so the induction hypothesis certifies c_1; symmetrically for c_2 with R and $q.2$. $\wedge$-introduction pairs the two certificates.
- *Case* $\kappa(\bar{t})$: here $\mathsf{Red}(\kappa, \sigma, c^*) = \top$ and $c^*[\kappa := \sigma] = \sigma(\bar{t})$. The vacuous q is discarded, and $\mathsf{Cert}^{\kappa}_{\sigma,\rho}(\Gamma, q, c^*) = \mathsf{Nav}(\Gamma \setminus \rho, \sigma(\bar{t}))$. The head lies inside $c{\downarrow}\rho$, so $\pi = \rho \cdot \pi'$ for some path π' (Lemma A.4). Every guard on this path is κ-free: above the LCA by the side condition of Scope, below it by acyclicity, as in Lemma A.1(c). Hence $\Gamma[\kappa := \sigma] = \Gamma$, and Lemma A.5 types the term at $\sigma(\bar{t})$.
- *Case* p, any other head: $\mathsf{Red}(\kappa, \sigma, p) = p[\kappa := \sigma] = p$, and $\mathsf{Cert}^{\kappa}_{\sigma,\rho}(\Gamma, q, p) = q$.

□

Now, we state soundness of Zap1. The composition in § A.4 additionally needs a bound on the free variables of the emitted proof term, which we record as a second conclusion.

Lemma A.7 (Zap1). *If* $\mathsf{Zap1}(\kappa, c) = (c', q)$ *then* $q : c' \to \exists\kappa.\, c$ *and* $\mathrm{FV}(q) \subseteq \mathsf{KVars}(c) \setminus \{\kappa\}$.

Proof. Zap1 (Fig. 8) emits $q = \lambda h.\langle \sigma,\ \mathsf{Cert}^{\kappa}_{\sigma,\rho}(\varepsilon, h, c)\rangle$, so suppose $h : c' = \mathsf{Red}(\kappa, \sigma, c)$.

Typing. Instantiate Lemma A.6 at $\pi = \varepsilon$: there $c^* = c \downarrow \varepsilon = c$ and $\Gamma = c \uparrow \varepsilon = \varepsilon$, so it gives $\mathsf{Cert}^{\kappa}_{\sigma,\rho}(\varepsilon, h, c) : c[\kappa := \sigma]$. The witness σ is κ-free and of κ's predicate sort (Lemma A.1), so $\exists$-introduction yields $\langle \sigma,\ \mathsf{Cert}^{\kappa}_{\sigma,\rho}(\varepsilon, h, c)\rangle : \exists\kappa.\, c$, and $\to$-introduction over h gives $q : c' \to \exists\kappa.\, c$.

Free variables. Every binder in q (the outer h and the λs emitted by $\mathsf{Cert}^{\kappa}_{\sigma,\rho}$) is unannotated, and every variable Nav emits is bound by one of them. The only other constituent of q is the witness σ, so $\mathrm{FV}(q) \subseteq \mathrm{FV}(\sigma) \subseteq \mathsf{KVars}(c) \setminus \{\kappa\}$ by Lemma A.1(c). □

In the proof of Theorem 4.1, each step bridge delivers $\exists\kappa_i.\, c_{i-1}$ while the accumulated bridge expects c_{i-1}, so bridges must compose underneath an existential binder. The next lemma justifies this. Here κ has a predicate sort $b_1 \to \cdots \to b_n \to \mathit{Prop}$, the case the $\exists$-rules of Fig. 13 admit.

Lemma A.8 (Existential lifting). *Let* κ *be a variable of predicate sort, and let*

$$\mathsf{lift}\, q \ \doteq\ \lambda h_1.\mathsf{let}\ \langle \kappa, h_2\rangle = h_1\ \mathsf{in}\ \langle \kappa,\ q\, h_2\rangle.$$

If $\Gamma, \kappa \vdash q : A \to B$ *then* $\Gamma \vdash \mathsf{lift}\, q : (\exists\kappa.\, A) \to (\exists\kappa.\, B)$.

Proof. The let rule binds κ and $h_2 : A$; then $q\, h_2 : B$, and $\exists$-introduction at κ gives $\langle \kappa, q\, h_2\rangle : \exists\kappa.\, B$, which does not mention the bound κ, discharging the side condition. □

Lifting leaves the composed bridge's $\exists$-prefix in elimination order; a final permutation restores the binder order of C, justified with the following lemma:

Lemma A.9 (Prefix permutation). *Let* $\overline{\kappa}'$ *be a reordering of* $\overline{\kappa}$*, and let*

$$\mathsf{perm}\ \doteq\ \lambda h_1.\mathsf{let}\ \langle \overline{\kappa},\ h_2\rangle = h_1\ \mathsf{in}\ \langle \overline{\kappa}';\ h_2\rangle.$$

Then $\vdash \mathsf{perm} : (\exists\overline{\kappa}.\, c) \to (\exists\overline{\kappa}'.\, c)$.

Proof. The nested let binds the distinct witnesses $\overline{\kappa}$ and exposes $h_2 : c$; iterated $\exists$-introduction re-bundles the same variables in the order $\overline{\kappa}'$. □

Theorem A.10 (Soundness, § 4.2). *If* $(C', q) = \mathsf{Zap}(C)$ *then* $\vdash q : C' \to C$*, and* q *is closed.*

Proof. With c_i as in § A.3, $c' = c_n$ and $C' = \exists\overline{\kappa_C}.\, c'$. By Lemma A.7, step i emits $q_i : c_i \to \exists\kappa_i.\, c_{i-1}$. Define $B_0 = \lambda h.\ h$ and $B_i = (\mathsf{lift}\, B_{i-1}) \circ q_i$, where $g \circ f$ abbreviates $\lambda h.g\,(f\,h)$:

$$\begin{aligned} q_i &: c_i \to \exists\kappa_i.\, c_{i-1} && \text{Lemma A.7}\\ \mathsf{lift}\, B_{i-1} &: \exists\kappa_i.\, c_{i-1} \to \exists\kappa_i\cdots\kappa_1.\, c_0 && \text{Lemma A.8}\\ B_i &: c_i \to \exists\kappa_i\cdots\kappa_1.\, c_0 \end{aligned}$$

All three judgments hold under the context of the variables still in scope at step i: $\kappa_{i+1}, \ldots, \kappa_n$ and $\overline{\kappa_C}$. By Lemma A.7, that is where q_i lives; Lemma A.8 lifts exactly the κ_i that q_i introduced. By Lemma A.3, each σ_j is cut-only, so no lift captures an acyclic variable.

At $i = n$, $B_n : c' \to \exists\kappa_n\cdots\kappa_1.\, c$ holds under the cut variables alone. Applying Lemma A.8 once per cut variable, written $\mathsf{lift}^{\overline{\kappa_C}}$, binds the cut prefix outside, since the σ_j may mention cut variables. By Lemma A.9, perm restores C's binder order:

$$q \ =\ \mathsf{perm} \circ \mathsf{lift}^{\overline{\kappa_C}} B_n \ :\ \exists\overline{\kappa_C}.\, c' \to \exists\overline{\kappa}.\, c \ =\ C' \to C.$$

This judgment holds under the empty context, so q is closed.

Up to flattening the intermediate **let**-bindings, q is the bridge Fig. 8 assembles: Zaps's unpack-/repack is lift, and Zap's top-level let in is the composite of $\mathsf{lift}^{\overline{\kappa_C}}$ and perm. □

Theorem A.11 (Zap-Equivalence). *If $(C', \cdot) = \mathsf{Zap}(C)$ then C is satisfiable iff C' is satisfiable.*

Proof. This is the Fusion result of Cosman and Jhala [14]: each Red step substitutes the strongest valid solution for an acyclic variable and preserves satisfiability. □

B Fix: Soundness and Completeness

This appendix proves the correctness of the Fix algorithm of § 5.2, whose two results are *Soundness* (Theorem 5.1) and *Completeness* (Theorem 5.2).

Overview Soundness is again a typing result. Completeness is the argument of Rondon et al. [51], with Lean derivability in place of semantic entailment.

Setup Throughout, we fix a closed constraint

$$C \;\equiv\; \exists \overline{\kappa_C}.\, c.$$

In the pipeline, C is Zap's residual and $\overline{\kappa_C}$ its cut set, but nothing below requires this: predicate abstraction applies to any closed constraint. We reuse the proof-term calculus of Fig. 13 and the conventions of § A.1.

Organization

- § B.1 recalls candidates, assignments, and the Oracle ;
- § B.2 characterizes the fixpoint computation;
- § B.3 types Cert_θ and the emitted bridge.

B.1 Preliminaries

Candidates and assignments Recall (§ 5.1) that a candidate q_ι instantiates a qualifier along an instance into a κ-free predicate of κ's type (closed, qualifiers being top-level definitions), that an assignment θ maps each κ to a finite candidate set, and that $\cdot[\theta]$ substitutes $\bigwedge \theta(\kappa)$ for each application of κ, extended homomorphically to atoms, contexts, and constraints. The fold $\mathsf{Red}^*(\overline{\kappa_C}, c)$ of Fig. 11 agrees with $c[\theta]$ except at heads, which it sends to $\top$; because candidates are κ-free, the fold order is immaterial.

Validity and order θ is *valid* for c, written $\theta \models c$, if at every head-prefix ρ with $c \downarrow \rho = \kappa(\bar{t})$ the κ-free goal $c \uparrow \rho[\theta] \vdash \kappa(\bar{t})[\theta]$ is derivable. Assignments are ordered by $\theta \preceq \theta'$ (θ *stronger*) iff $\theta'(\kappa) \subseteq \theta(\kappa)$ for every κ: fewer candidates is a weaker conjunction.

The Oracle Weaken and Cert_θ call an Oracle subject to the two laws of § 5.3: by *(Oracle-Soundness)* a returned term proves the queried goal, and by *(Oracle-Completeness)* the Oracle succeeds on every derivable goal. Soundness uses *(Oracle-Soundness)* only; *(Oracle-Completeness)* is used only for completeness, without which the completeness claim weakens to the strongest Oracle *-certifiable* assignment.

B.2 The fixpoint computation

Lemma B.1 (Monotonicity). *Let $\theta \preceq \theta'$.*

(a) For every atom p, $p[\theta]$ entails $p[\theta']$.
(b) For every context Γ and goal ψ, if $\Gamma[\theta'] \vdash \psi$ then $\Gamma[\theta] \vdash \psi$.

Proof.

(a) A κ-free atom is unchanged. At $\kappa(\bar{t})$ the two sides are conjunctions over $\theta(\kappa) \supseteq \theta'(\kappa)$, so the first contains every conjunct of the second; project and re-pair.
(b) Every hypothesis of $\Gamma[\theta']$ is an applied atom. By (a), each is derivable from the corresponding hypothesis of $\Gamma[\theta]$; cut these derivations into the given one.

□

Part (c) below assumes *(Oracle-Completeness)*; parts (a) and (b) do not.

Lemma B.2 (Fixpoint). *Let* $\theta^\star = \mathsf{Fixpoint}(c, \theta_0)$ *with* $\theta_0 = [\kappa \mapsto \mathsf{Init}(\kappa, Q) \mid \kappa \in \overline{\kappa_C}]$.

(a) Termination. Fixpoint *terminates.*
(b) Validity. $\theta^\star \models c$.
(c) Strength. $\theta^\star \preceq \theta'$ *for every Q-assignment* θ' *with* $\theta' \models c$.

Proof.

(a) Termination. θ_0 is finite, and each recursive call strictly decreases $\sum_\kappa |\theta(\kappa)| \in \mathit{Nat}$: Weaken returns either a strictly smaller candidate set or $\bot$ (Fig. 11).

(b) Validity. On exit, $\mathsf{Weaken}(c, \theta^\star, \rho) = \bot$ at every head-prefix ρ: the surviving subset I equals $\theta^\star(\kappa)$, so the Oracle succeeded on $q_\iota(\bar{t})$ under $c{\uparrow}\rho[\theta^\star]$ for every $q_\iota \in \theta^\star(\kappa)$. By *(Oracle-Soundness)*, each success is a derivation of $q_\iota(\bar{t})$, and $\wedge$-introduction assembles them into validity at ρ.

(c) Strength. By induction on the iterates θ^k, with invariant $\theta'(\kappa) \subseteq \theta^k(\kappa)$ for every κ, i.e. $\theta^k \preceq \theta'$. At the base, θ_0 contains all instances. For the step, suppose Weaken at ρ drops some $q_\iota \in \theta'(\kappa)$: the Oracle failed on $q_\iota(\bar{t})$ under $c{\uparrow}\rho[\theta^k]$. By validity of θ' and $\wedge$-projection, $c{\uparrow}\rho[\theta'] \vdash q_\iota(\bar{t})$. By the invariant and Lemma B.1(b), this derivation transports to $c{\uparrow}\rho[\theta^k]$, and by *(Oracle-Completeness)* the Oracle succeeds, a contradiction. Hence no candidate of θ' is ever dropped, and $\theta^\star \preceq \theta'$. □

B.3 Soundness and completeness

Lemma B.3 (Cert_θ). *Let* $\theta^\star$ *be as in Lemma B.2, and suppose every* Oracle *query made by* Cert_θ *succeeds. For every prefix* π, *let* $c^* = c \downarrow \pi$ *and* $\Gamma = c \uparrow \pi$. *If* $\Gamma[\theta^\star] \vdash q : \mathsf{Red}^*(\overline{\kappa_C}, c^*)$ *then* $\Gamma[\theta^\star] \vdash \mathsf{Cert}_\theta(\Gamma, q, c^*) : c^*[\theta^\star]$.

Proof. By induction on c^*, as in Lemma A.6: the $\forall$, $\to$, $\wedge$, and κ-free-head cases are verbatim, with both operators substituting the guards identically.

At a head $c^* = \kappa(\bar{t})$, $\mathsf{Red}^*(\overline{\kappa_C}, c^*) = \top$ and $c^*[\theta^\star] = (\bigwedge \theta^\star(\kappa))(\bar{t})$. Cert_θ (Fig. 11) discards the vacuous q and queries the Oracle on this applied conjunction under the applied context. The query succeeds by hypothesis, and by *(Oracle-Soundness)* the returned term proves $(\bigwedge \theta^\star(\kappa))(\bar{t})$. □

Theorem B.4 (Soundness, § 5.3). *If* $\mathsf{Fix}(C, Q) = (C', q)$ *then* C' *is a VC and* $\vdash q : C' \to C$, *and* q *is closed.*

Proof. By Fig. 11, $C' = \mathsf{Red}^*(\overline{\kappa_C}, c)$ and, writing $\overline{\kappa_C} = [\kappa_1, \ldots, \kappa_r]$,

$$q \;=\; \lambda h.\Big\langle \bigwedge \theta^\star(\kappa_1), \ldots, \bigwedge \theta^\star(\kappa_r);\; \mathsf{Cert}_\theta(\varepsilon, h, c)\Big\rangle.$$

C' is a VC: heads go to $\top$, and guards go to $\cdot[\theta^\star]$ with κ-free candidates.

Typing. Since Fix returned, every Oracle query made by Cert_θ succeeded, so Lemma B.3 applies. Given $h : C'$, Lemma B.3 at $\pi = \varepsilon$ gives $\mathsf{Cert}_\theta(\varepsilon, h, c) : c[\theta^\star]$, which is c with each κ_j replaced by the closed, κ-free predicate $\bigwedge \theta^\star(\kappa_j)$ of κ_j's sort. Iterated $\exists$-introduction at predicate sort (Fig. 13) rebinds the prefix, yielding $\exists\overline{\kappa_C}.\, c = C$, and $\to$-introduction over h gives $q : C' \to C$. The judgment holds under the empty context, so q is closed. □

Completeness assumes *(Oracle-Completeness)*, as scoped in § 5.3.

Theorem B.5 (Completeness, § 5.3). *If* $\theta^\star = \mathsf{PredAbs}(C, Q)$, *then* $\theta^\star \preceq \theta'$ *for every Q-assignment* θ' *with* $\theta' \models c$.

Proof. This is Lemma B.2(c). □

C Qualifier-Scraping Rules

Flex implements qualifier scraping to generate candidates for Fix. Here, we describe selected rules implemented in Flex; their instances are validated by PredAbs. Applying all rules at once can cause the candidate space to grow rapidly. We therefore extend the qualifier bank *incrementally*, moving to the next level when synthesis fails or the resulting assignment fails its safety checks:

- **Bank:** Use the existing qualifier bank without scraping.
- **Basic:** Extract predicates from CHC heads.
- **Standard:** Add premise predicates, guarded arithmetic, and candidates from directly referenced definitions and source properties at κ-argument positions.
- **Deep:** Explore helper definitions and add further numeric and structural generalizations, including quantified bounds, accumulator completion, and affine relations.

Each level retains the candidates accumulated at earlier levels.

Qualifier Bank. We provide 19 built-in qualifier templates. The predicates below are abstracted over x for unary templates and x, y for binary templates, with variables of the indicated type:

Type	Unary: $\lambda x.\, p$	Binary: $\lambda x\, y.\, p$
Int	$x = 0,\ 0 < x,\ 0 \le x,\ x < 0,\ x \le 0$	$x = y,\ y < x,\ y \le x,\ x < y,\ x \le y,\ x \le y - 1$
Nat	$x = 0,\ 0 < x$	$x = y,\ x < y,\ x \le y$
Bool	$x = \mathsf{true},\ x = \mathsf{false}$	$x = y$

Scope and Notation. Let Γ contain caller locals, which remain fixed, and define

$$\mathrm{abs}_\Gamma(p) \doteq \lambda \overline{x}.p, \qquad \overline{x} = \mathrm{FV}(p) \setminus \mathrm{dom}(\Gamma).$$

From a source predicate p, $\mathrm{abs}_\Gamma(p)$ forms a qualifier template by abstracting the free variables outside Γ. The abstraction must be well-scoped, κ-free, and contain no unresolved metavariables. Its nonempty parameter list is bounded, and each parameter has a closed data type; function types are allowed. We exclude $\top$, $\bot$, and predicates with an outermost $\forall$ or $\to$.

Constraint Atoms. To scrape qualifiers from atoms in the constraint, such as $x \le y$, we traverse conjunctions, open universal binders, and substitute let-bound values. Basic scraping selects heads; Standard and Deep also select premise conjuncts. The following rule describes *atom* scraping for a selected predicate p:

$$\frac{q = \mathrm{abs}_\Gamma(p)}{p \rightsquigarrow q}\ \textsc{Atom}$$

Here, $p \rightsquigarrow q$ means that scraping p proposes qualifier q. For example, $x \le y$ proposes $\lambda x\, y.x \le y$ when x and y are clause-local variables.

Guarded Arithmetic. We generate guarded qualifiers only when the whole constraint lies in the supported arithmetic fragment. Let B contain one premise branch and its head, and let $A(B)$ collect its selected arithmetic atoms, including bounds from equalities to literals. We split disjunctive premises and open existential premises with fresh witnesses.

$$\frac{\begin{array}{c} a, b \in A(B) \quad \kappa(\bar{t}) \in B \quad \mathrm{FV}(a), \mathrm{FV}(b) \neq \emptyset \\ \mathrm{FV}(a) \neq \mathrm{FV}(b) \qquad \mathrm{FV}(a) \cup \mathrm{FV}(b) \subseteq \mathrm{FV}(\bar{t}) \quad g \in \{a, \bar{a}\} \quad p \in \{b, \bar{b}\} \end{array}}{B \rightsquigarrow \mathrm{abs}_\Gamma(\bar{g} \vee p)}\ \textsc{Guard}$$

Here $\bar{a}$ is the normalized complement, e.g., $\overline{s \le t} = t < s$ and $\overline{s < t} = t \le s$. Thus $3 \le y$ and $0 \le x$, observed with $\kappa(x, y)$, can suggest $\lambda x\, y.y < 3 \vee 0 \le x$.

Definition Calls and Tests. We inspect calls and branch conditions in the constraint and selected defining equations. Let T collect the visited terms, with binders opened. Calls are inspected without recursive unfolding.

$$\frac{f(\overline{t}) \in T \quad f \text{ selected} \qquad f(\overline{t}) : b \quad v : b \text{ fresh}}{T \rightsquigarrow \mathrm{abs}_\Gamma(v = f(\overline{t}))}\ \textsc{Call}$$

$$\frac{(\textbf{if } p \textbf{ then } t \textbf{ else } u) \in T \qquad a \in \mathrm{atoms}(p) \cup \mathrm{atoms}(\neg p)}{T \rightsquigarrow \mathrm{abs}_\Gamma(a)}\ \textsc{Test}$$

Here, b is closed and is neither a proposition, function type, nor sort. The operation atoms splits conjunctions and substitutes let-bound values without distributing negation.

Argument Positions. Consider a source occurrence $\kappa(x_0, \ldots, x_n)$ whose arguments are distinct variables of closed data types. For a resolved, κ-free proposition p, we abstract the arguments it uses and record their original positions:

$$\mathrm{at}_\kappa(p) \doteq q_\iota, \qquad q = \lambda z_0 \cdots z_{r-1}.p[z_j/x_{i_j}]_{j=0}^{r-1}, \quad \iota(j) = i_j.$$

Here, $x_{i_0}, \ldots, x_{i_{r-1}}$ are exactly the arguments occurring free in p. We require $r > 0$ and $\mathrm{FV}(q) \subseteq \mathrm{dom}(\Gamma)$, so any remaining free variables belong to the caller's context. Unlike ordinary atoms, these candidates may retain quantifiers.

Quantified Bounds. Let O collect source observations at $\kappa(\overline{x})$. A quantified property can suggest an invariant by replacing a free numeric bound with a same-typed argument of κ:

$$\frac{p \in O \quad p \text{ contains } \forall \qquad n \in \mathrm{FV}(p) \setminus \{\overline{x}\} \quad n, x_i : b \quad b \in \{Int, Nat\}}{O, \kappa(\overline{x}) \rightsquigarrow \mathrm{at}_\kappa(p[x_i/n])}\ \textsc{Bound}$$

For example, replacing n by the current index i in $\forall j.\ 0 \le j < n \to m(j) \neq 0$ proposes the corresponding property over the processed prefix.

Accumulator Completion. An observed final equality $a = E[xs]$ can suggest an invariant relating accumulated and remaining work. We combine a with $E[r]$ for a compatible list argument r, where $E[r]$ replaces all free occurrences of xs by r in $E[xs]$:

$$\frac{\begin{array}{c}(a = E[xs]) \in O \quad a, r \in \overline{x} \\ xs, r : \mathsf{List}\ b \quad xs \notin \overline{x} \qquad \mathrm{FV}(E[xs]) \cap \{\overline{x}\} = \varnothing \quad a, E[xs] : Int\end{array}}{O, \kappa(\overline{x}) \rightsquigarrow \mathrm{at}_\kappa(a + E[r] = E[xs])}\ \textsc{Complete}$$

For example, $a = sum(xs)$ suggests $a + sum(r) = sum(xs)$. Substitution avoids variable capture, and PredAbs must validate the candidate; scraping assumes no additivity of E.

D Additional Evaluation Results

D.1 Elimination Proof Metrics

Table 4 reports elimination certificate depth, distinct constants, and kernel-checking time for § 7.2.

D.2 RQ3 Results

Table 5 gives the tactic-time distribution for the refinement-typechecker experiment.

Table 6 gives the per-suite results for the CHC-COMP experiment.

Failure Breakdown. Of the 1,198 CHCs that Spacer solves natively, Flex Spacer proves 983. Of the remaining 215, 31 yield no model after the round trip through Lean, 10 fail during model certification, and two exhaust memory. The other 172 have all predicates instantiated but leave proof obligations. Of the 937 CHCs that Eldarica solves natively, Flex Eldarica proves 799; of the

		CHCs		Zap			Zap_grind			Zap_aesop		
	Suite	Acyclic only	Mixed	Depth	Consts	Kernel (μs)	×depth	×consts	×kernel	×depth	×consts	×kernel
	`flux-tests`	240	55	45.4	24.4	759	1.4×	2.3×	3.5×	1.4×	1.7×	2.1×
	`flux-medium`	36	44	75.6	27.8	2,180	1.5×	2.9×	5.5×	1.4×	1.7×	2.3×
	`wave`	26	9	99.4	34.6	2,828	1.6×	2.2×	3.9×	1.5×	1.6×	2.6×
	`liquid-fix`	50	7	47.6	20.0	1,041	1.6×	2.8×	5.7×	1.8×	2.0×	4.2×
	Case studies	27	20	166	47.1	14,496	1.3×	1.9×	2.8×	1.4×	1.9×	2.5×
CHC-COMP	Eldarica	0	97	402	42.3	253,140	1.3×	2.8×	3.5×	1.5×	1.7×	5.6×
CHC-COMP	HCAI	0	104	183	42.9	11,718	1.0×	1.3×	1.0×	1.0×	1.2×	1.0×
CHC-COMP	HoPV	32	46	161	30.6	10,791	1.3×	2.6×	4.0×	1.4×	1.6×	3.8×
CHC-COMP	Kind2	0	395	492	44.2	95,315	–	–	–	–	–	–
CHC-COMP	LIA-XS	0	6	64.8	32.4	1,972	1.5×	3.0×	6.2×	2.0×	1.9×	7.9×
CHC-COMP	LLReve	2	9	335	41.8	59,332	1.9×	3.6×	14.0×	–	–	–
CHC-COMP	RustHorn	2	8	140	26.0	22,640	–	–	–	–	–	–
CHC-COMP	Synthesis	0	60	270	36.0	609,511	1.0×	1.7×	1.5×	1.8×	2.0×	7.6×
	All	415	860	168	34.4	14,179	1.4×	2.4×	3.7×	1.4×	1.7×	2.5×

Table 4. Elimination certificate depth, distinct constants, and kernel-checking time, including generated helper proofs. Values are geometric means; ratios compare each variant with Zap on jointly reduced CHCs.

			Tactic time (s)					
Configuration	Suite	Proved	Mean	P25	Median	P75	P90	Max
Zap + Fix	`flux-tests`	333	0.42	0.03	0.05	0.30	1.04	14.7
	`flux-medium`	94	1.59	0.07	0.32	0.77	2.02	49.3
	`wave`	35	1.95	0.06	0.31	0.75	2.83	24.0
	`liquid-fix`	67	0.75	0.03	0.04	0.14	0.70	25.2
	All	529	0.77	0.03	0.08	0.42	1.12	49.3
Flex Spacer	`flux-tests`	310	0.24	0.09	0.12	0.22	0.47	7.2
	`flux-medium`	91	0.49	0.16	0.23	0.37	0.56	8.4
	`wave`	35	0.38	0.17	0.27	0.41	0.87	1.4
	`liquid-fix`	65	0.69	0.10	0.12	0.19	0.48	30.2
	All	501	0.36	0.10	0.15	0.27	0.51	30.2
Flex Eldarica	`flux-tests`	320	1.32	0.66	0.78	1.10	1.67	87.7
	`flux-medium`	91	1.73	0.81	1.05	1.34	1.84	39.5
	`wave`	36	2.20	1.00	1.44	2.53	4.48	8.8
	`liquid-fix`	65	1.13	0.68	0.73	0.82	1.63	8.7
	All	512	1.43	0.69	0.81	1.23	1.90	87.7

Table 5. Tactic time in seconds over the CHCs each configuration of Table 3 fully proves. Percentiles are linearly interpolated.

remaining 138, 16 yield no model, five fail during model certification, one exhausts memory, and 116 leave proof obligations. Almost all of these fail at head elimination (170 of 172 for Spacer and 115 of 116 for Eldarica): their imported invariants are large LIA formulas that `omega` and `lean-smt` cannot prove within the budget. Table 7 gives the per-suite breakdown.

Suite	CHCs	Native Spacer	Native Eldarica	Zap+Fix		Flex Spacer	Flex Eldarica
				Full	Partial		
VMT	474	412	183	0	0	301	151
Kind2	395	267	196	0	395	199	152
Eldarica	138	108	121	24	83	101	108
Synthesis	109	95	67	22	54	88	63
HoPV	108	100	107	76	23	99	107
HCAI	104	92	90	23	81	91	85
LLReve	104	90	96	56	6	85	95
LIA-XS	55	16	36	17	3	15	31
AEval	55	7	27	2	0	7	20
RustHorn	10	8	10	1	9	8	10
Ultimate	7	3	4	0	0	3	0
All	1,559	1,198	937	221	654	997	822

Table 6. Coverage on 1,559 distinct CHC-COMP constraints. *Native Spacer* and *Native Eldarica*: CHCs the solver solves on its own. Zap+Fix: CHCs Zap and Fix prove outright (Full), and CHCs with a partial result (Partial): Zap eliminates at least one predicate, or Fix proves every constraint involving a predicate Zap cannot eliminate. *Flex Spacer* and *Flex Eldarica*: CHCs Flex fully proves using that solver as its backend.

Suite	Flex Spacer				Flex Eldarica			
	Solver	Cert.	Disch.	Process	Solver	Cert.	Disch.	Process
VMT	3	1	108	0	5	0	40	0
Kind2	18	4	47	2	6	4	40	1
Eldarica	2	0	6	0	2	0	11	0
Synthesis	4	2	6	0	2	0	3	0
HoPV	1	0	0	0	0	0	0	0
HCAI	0	1	1	0	0	0	5	0
LLReve	3	0	3	0	1	0	2	0
LIA-XS	0	1	0	0	0	0	5	0
AEval	0	1	1	0	0	1	6	0
RustHorn	0	0	0	0	0	0	0	0
Ultimate	0	0	0	0	0	0	4	0
All	31	10	172	2	16	5	116	1

Table 7. CHCs that Spacer (resp. Eldarica) solves natively but Flex Spacer (resp. Flex Eldarica) does not fully prove, by the furthest stage reached. Solver: no model; Cert.: model-certification failure, including timeouts; Disch.: proof obligations left undischarged, including predicate heads; Process: out of memory.

D.3 RQ4 Scraping Results

The 391 CHCs comprise 274 cyclic-only and 117 mixed constraints. We ran each CHC once per configuration. Tactic times exclude startup, imports, and final kernel checking.

Corpus	CHCs	Bank	Basic	Standard	Deep	Incremental
Refinement-typechecker	181	154	166	158	166	181
CHC-COMP	192	130	130	157	178	192
RQ1 case studies	18	1	9	18	18	18
Total	**391**	**285**	**305**	**333**	**362**	**391**

Table 8. Fully proved CHCs by scraping configuration.

Configuration	n	Mean	P25	Median	P75	P90	Max
Bank	285	1.53	0.18	0.37	1.07	3.64	29.04
Basic	305	1.51	0.19	0.38	1.19	3.60	29.12
Standard	333	6.12	0.38	1.16	5.38	17.59	99.80
Deep	362	7.05	0.44	1.56	6.99	21.46	101.82
Incremental	391	3.19	0.21	0.50	1.80	8.82	48.93

Table 9. Tactic time in seconds over fully proved CHCs (n). Quantiles use linear interpolation.

Level	CHCs	Share of 391
Bank	285	72.9%
Basic	22	5.6%
Standard	52	13.3%
Deep	32	8.2%
Total	**391**	**100%**

Table 10. First level where Incremental accepted an invariant.

One Standard and three Deep runs hit the 300 s process watchdog. Standard and Deep each eliminate all predicates in three further CHCs but leave residual proof obligations.