Foundational Constraint Solving for Expressive Refinement Typing

JAM KABEER ALI KHAN, Independent, Hong Kong SAR
PETROS MARKOPOULOS, University of California San Diego, USA
NICO LEHMANN, University of Chile, Chile
RANJIT JHALA*, University of California San Diego, USA

SMT-based program verifiers are hamstrung by two problems: *expressiveness*, because predictable verification restricts to the boundaries of SMT decidability, and *trust*, because the solver is a large, unverified artifact whose soundness bugs may quietly compromise every tool built on it. We present Flex, a foundational Constrained Horn Clause (CHC) solver implemented in Lean, that reduces the trusted base to the kernel alone, and allows using Lean's entire proof ecosystem to verify low-level systems code, via three contributions. First, Flex encodes CHCs as plain Lean propositions where the Horn variables are existentially bound predicates, and shows how to implement CHC solvers as tactics (meta-programs) that compute kernel-checkable proofs of the CHC propositions. Second, we show how to implement two verified CHC generators in Lean: a Floyd-Hoare style generator for an imperative language, and a refinement-type-based generator for a functional calculus, which can be composed with the solving tactics to yield the first end-to-end foundational CHC-based verifiers. Finally, we show how Flex allows us to leapfrog the expressiveness limitations of SMT by unleashing Lean's entire ecosystem of proof machinery to prove arbitrary functional correctness properties of various low-level Rust libraries using the Flux refinement type checker, and demonstrate the viability of Flex as a trustworthy CHC backend, by showing it automatically discharges 95.7% of the CHCs from Flux's benchmark suite.

Additional Key Words and Phrases: Constraint Horn Clauses, Theorem Provers, Refinement Types, Verification

1 Introduction

SMT solvers are the beating heart of automated verification tools [2, 22, 44, 45, 48, 67, 73] that convert messy, impure code into purely logical formulas whose validity, automatically checked by the solver, guarantees the code meets critical correctness and security requirements [31, 43, 46, 61, 79].

However, experience shows that SMT solvers are hamstrung by two problems: *expressiveness* and *trust*. First, the solvers implement decision procedures for a fixed set of decidable theories [55], and use heuristics to reason about formulas that lie beyond [17]. While the decision procedures make short work of formulas that arise in proving specifications over integers, arithmetic, or uninterpreted functions, verification becomes a slog for more *expressive* specifications with quantifiers or user-defined predicates, or even when reasoning with decidable but hard theories like bit-vectors. Here, the incomplete heuristics make solving unpredictable due to "proof instability" [49, 78], and the black-box nature of SMT — where it simply says whether a formula is valid or not — leaves the programmer flinging darts in the dark to divine what extra hints the solver might require to complete a proof [26, 53]. Second, SMT engines are substantial artifacts: Z3 and CVC4 weigh around half a million lines of C++, with known soundness issues [11, 74–76] — a significant *trusted computing base* even before accounting for the verification formula generators.

Flex We present Flex, a new approach to engineering program verifiers using foundational Horn constraint solving. Flex is based on the observation that the task of verification can be split into a front-end that generates *Constrained Horn Clauses* (CHCs) and a back-end that determines their satisfiability [6]. A CHC is a logical formula over a set of *Horn variables* representing unknown

*Corresponding author.

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; Nico Lehmann, University of Chile, Santiago, Chile, nlehmann@dcc.uchile.cl; Ranjit Jhala, University of California San Diego, San Diego, USA, rjhala@ucsd.edu.

program invariants, which must satisfy requirements captured by conjunctions and implications over those variables. Our key insight is that CHCs are *existentially quantified propositions* in a foundational logic, such as that of Lean or Rocq or Isabelle. Hence, we can entirely leapfrog the expressiveness and trust limitations of SMT by implementing a CHC solver *within* a foundational prover, as a meta-program that *computes proofs of* CHC propositions. In this paper, we develop, implement and evaluate this approach via three contributions.

1. Foundational Solving (§4) Our first contribution is to show how CHCs can be shallowly embedded as Lean propositions, and then show how existing (SMT-based) CHC solving techniques can be reimagined as tactics that *reduce* away the Horn variables from the proposition, leaving behind a plain formula that can be discharged with existing Lean machinery. To solve CHCs our tactics must compute *witness predicates* for the existentially bound Horn variables. Following [14], we observe that these variables can be partitioned into *acyclic* variables which can be solved exactly or *cyclic* variables which must be approximated because they arise respectively from "straight-line" or "looping" code. We show how to compute this partition, and then how to adapt the Fusion [14] and Predicate Abstraction [21, 62] CHC solving algorithms to the foundational setting, where we can parameterize them with extensible *oracles* that enable synthesizing invariants over user-defined theories. Of course, a foundational kernel will not take any reduction of the Horn variables on faith: it demands proof that the reduction is legitimate. A key contribution of our tactics is to show how to exploit the structure of CHCs and their solutions to *automatically construct certificates* that the reduced CHC logically implies the original one, yielding the first foundational CHC solver.

2. Foundational Generation (§5) Our second contribution is to implement two foundational CHC *generators*, which, when connected with Flex, yield the first end-to-end CHC-based verifiers. As a warmup, we develop a *Floyd-Hoare* style generator [24, 35] for an imperative language Imp, that follows the textbook recipe [56, 60] but crucially uses Horn variables to defer loop-invariant synthesis to the back-end solver. Next, we present the first mechanization of *algorithmic refinement typing* [38], by deeply embedding a core functional calculus λ_{RK} in Lean, formalizing a big-step operational semantics for it, defining a semantic interpretation for refinement types, implementing a CHC generator that uses Horn variables for unknown *refinements*, and proving that if the generated CHCs are satisfiable, then the source yields — without getting stuck — a value in its type's interpretation. Together these verifiers offer a blueprint for how to engineer CHC-based verification tools entirely within a proof assistant.

3. Expressive Verification (§6) Our final contribution demonstrates Flex's ultimate payoff: liberating verification from SMT's expressiveness and trust limitations by unleashing Lean's entire ecosystem of proof machinery — theories, tactics, agents, and all — to prove arbitrary functional correctness properties of low-level Rust code. The Flux verifier uses refinement types to distill impure Rust code — with the classic challenges of mutable state like references and loops, compounded by hairy modern constructs like traits, closures and asynchrony — into purely logical CHCs. Previously, the expressiveness of Flux's specifications was carefully restricted to ensure the CHCs fell within the boundaries of SMT decidability. Flex tears down this wall, by using Lean as a backend to discharge functional correctness specifications, as illustrated by a variety of case studies: correctness of in-place sorting algorithms; correctness of a hash table with chaining for collision resolution; modular arithmetic and bit-vector properties required for secure memory isolation in the Tock embedded kernel [61]; and proving that a ring-buffer-backed dequeue from the embedded kernel [64] never reads from uninitialized memory. Further, we show that Flex is viable as a *trustworthy* back-end CHC solver that is able to automatically discharge more than 95% of the 880 CHCs that arise in Flux's existing benchmark suite spanning 20K lines of Rust source. This high degree of automation is made possible by Flex's certificate generation machinery which

```
fn foreach<F>(lo:usize, hi:usize, mut f:F)
where F: FnMut(usize{v:lo<=v && v<hi})
{
  let mut i = lo;
  while i < hi {
    f(i);
    i += 1;
  }
}
```

$$c_{for} \doteq \exists \kappa : Int \to Int \to Int \to Prop.\ \forall lo\ hi : Int.\ \kappa(lo, hi, lo) \wedge \forall i : Int.\ \kappa(i, hi, lo) \to i < hi \to lo \le i \wedge i < hi \wedge \kappa(i+1, hi, lo)$$

```
fn dot(n:usize, x:&[f64][n], y:&[f64][n])
   -> f64
{
  let mut res = 0.0;
  let body = |j| { res += x[j] * y[j] };
  foreach(0, n, body);
  res
}
```

$$c_{dot} \doteq \exists \kappa : Int \to Int \to Prop.\ \forall n : Int.\ \forall j : Int.\ \kappa(j, n) \to j < n \wedge \forall v : Int.\ (0 \le v \wedge v < n) \to \kappa(v, n)$$

Fig. 1. (L) Rust with Flux specifications, (R) verification CHCs generated by Flux

successfully reduces *every* acyclic variable across 369 CHCs without search, compared to a baseline of Lean's general-purpose `grind` and `aesop` which close only 67% and 90% of CHCs while taking 18 to 34× longer to complete the proof. [1]

2 Overview

Let us begin with an overview of Flex. We start by recalling how refinement type checking reduces to *generating* CHC constraints (§ 2.1). Next, we illustrate our first contribution by showing how Flex *solves* the constraints in a foundational setting (§ 2.2). Then, we demonstrate our second contribution where we use Flex's solvers to implement foundational verifiers in Lean (§ 2.3). Finally, we conclude by showing how by tapping into Lean's full machinery, Flex lets Flux verify arbitrarily expressive specifications of Rust code (§ 2.4).

2.1 Horn Constraints

Consider the two Rust functions shown on the left in Fig. 1. The function `foreach` is a higher-order loop, that takes as input three parameters: a *range* denoted by a lower bound `lo` and upper bound `hi`, and a body `f`. The function uses a `while` loop to evaluate `f(i)` for each index `i` in the supplied range. The function `dot` takes as input two Rust slices `x` and `y` of dimension `n`, and then computes the dot product of the two slices by invoking `foreach` on the range `[0, n)`, and the *closure* `body` that accumulates the contribution of the j^{th} dimension of each slice.

Specification: Refinement Types Suppose we wish to statically verify that `dot` does not panic by performing an out-of-bounds access. To do so, the programmer writes Flux *refinement type specifications* for each function. The specification for `foreach` says that the closure `f` will only be called at indices `v` that are in the range `[lo, hi)`, as stipulated by the closure's precondition: an existential refinement on the input type of `f`. For `dot`, the specification requires callers pass in slices `x` and `y` with `n` elements, as prescribed by the indexed slice type `&[f64][n]`.

[1] Flex can be found at https://github.com/jam-khan/Flex.

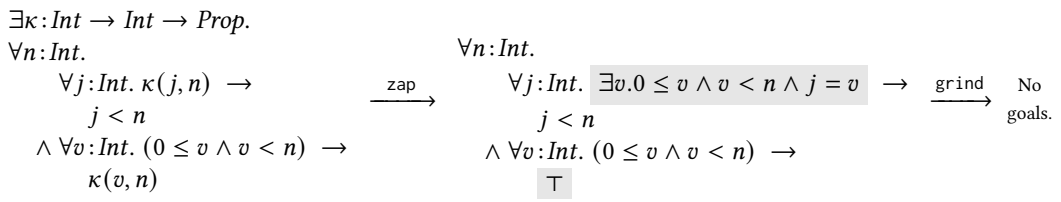

Fig. 2. (L) The CHC for `dot`, (C) zap the *acyclic* κ replacing head occurrences with $\top$ and body occurrences with the strongest solution, (R) `grind` then discharges the residual VC to verify `dot`.

Verification: Constraints To *verify* both functions in Fig. 1, a refinement type verifier like Flux runs the type checker to generate a *verification constraint* shown on the function's right. Each constraint is a *Constrained Horn Clause* (CHC) that comprises the following elements. At the top, a CHC has some number of existentially quantified *Horn variables* (denoted by κ) which represent *unknown* refinements or invariants. Next, the constraint itself, comprises a sequence of nested (1) *universal* bindings, corresponding to statically unknown program values; (2) *conjunctions*, corresponding to multiple requirements; (3) *implications*, corresponding to hypotheses about program values; ending in a (4) *head* obligation that must hold under the preceding hypotheses.

CHC for `foreach` The constraint c_{for} has one Horn variable $\kappa : Int \to Int \to Int \to Prop$ representing the `while`-loop invariant over the counter `i`, upper bound `hi`, and lower bound `lo`. The first (head) conjunct $\kappa(lo, hi, lo)$ says the invariant must hold at loop entry (when `i = lo`). The second conjunct is the inductive step: if the invariant holds at `i` and the loop guard $i < hi$ is true, then we have two head obligations. (a) First, the closure's *precondition* $lo \leq i \wedge i < hi$ must hold, ensuring `f` is called within the specified range. (b) Second, the invariant must be preserved at $i + 1$, as mandated by the head $\kappa(i + 1, hi, lo)$.

CHC for `dot` Closures would be quite tiresome to use if programmers had to explicitly annotate them with their contracts. To relieve them of this tedium, Flux introduces a Horn variable $\kappa : Int \to Int \to Prop$ to represent the unknown input refinement for the closure `body`. The first conjunct of the constraint says that whenever the closure executes with an index `j`, which is assumed to satisfy the closure's input refinement $\kappa(j, n)$, the bound $j < n$ needed to safely access `x[j]` and `y[j]` must hold. The second conjunct is obtained from the *subtyping* obligation between the function type of the argument (`body`) and of the `foreach`'s parameter (`f`). Specifically, by contravariant input subtyping, the constraint says that since `foreach` calls the closure with v where $0 \leq v \wedge v < n$ (as it is called with $lo = 0$, $hi = n$), all such values must satisfy the input refinement $\kappa(v, n)$.

2.2 Foundational Solvers

The verification constraints (CHCs) distill Rust's stateful, temporal, and impure semantics into purely logical propositions — precisely the kind of thing that proof assistants were designed to prove. Except for one major challenge: proofs of CHC propositions must supply *explicit witnesses* for the existentially bound Horn variables, satisfying the head obligations on those variables. (Indeed, we tried proving some CHCs by hand — and with coding agents — but it was readily apparent that this direct route was utterly impractical.)

Flex addresses this challenge by reimagining two existing CHC-solving algorithms as *rewriting tactics* for the foundational setting.

Cyclic vs Acyclic Variables The Horn variables κ in CHCs come in two flavors. The variable κ may be *cyclic*, which means, roughly (we defer a precise definition to § 4.1), that it appears in the hypotheses of a clause where it also appears in the head; and otherwise we say it is *acyclic*. For

$$\exists \kappa : Int \to Int \to Int \to Prop.$$
$$\forall lo\ hi : Int.$$
$$\kappa(lo, hi, lo)$$
$$\wedge\ \forall i : Int.$$
$$\kappa(i, hi, lo) \ \to\ i < hi \ \to$$
$$lo \le i \wedge i < hi$$
$$\wedge\ \kappa(i+1, hi, lo)$$

$\xrightarrow{\texttt{fix}}$

$$\forall lo\ hi : Int.$$
$$\top$$
$$\wedge\ \forall i : Int.$$
$$lo \le i \wedge i \le hi \ \to\ i < hi \ \to$$
$$lo \le i \wedge i < hi$$
$$\wedge\ \top$$

$\xrightarrow{\texttt{grind}}$ No goals.

Fig. 3. (L) The CHC for `foreach`, (C) `fix` uses qualifiers to reduce the *cyclic* κ, replacing head occurrences with $\top$ and body occurrences with the solution, (R) `grind` then discharges the residual VC to verify `foreach`.

example, in the constraint c_{for} in Fig. 1, κ is cyclic as it appears in both the hypothesis $\kappa(i, hi, lo)$ and the head $\kappa(i+1, hi, lo)$ of the inductive clause. In contrast, the variable κ in c_{dot} is acyclic as it appears only in hypotheses in the first clause and only in head in the second.

Solving Acyclic Variables For an *acyclic* variable κ we can compute an exact closed form solution, perhaps in terms of *other* variables. The Fusion algorithm of [14] shows how to compute the *strongest* solution, informally by taking the *conjunction* of all the hypotheses that occur along the path to each head occurence of that κ, and then computing the *disjunction* over all the paths leading to head occurrences of the κ.

In Flex, the above algorithm is implemented in a tactic `zap` detailed in § 4.2, and shown in action in Fig. 2. On the left, we have the original CHC proposition. First, `zap` computes the strongest solution by traversing the CHC to the (single) head application of κ, collecting the hypotheses along the path, and existentially quantifying the universally quantified binders:

$$\kappa \ \doteq\ \lambda z, n. \exists v. 0 \le v \wedge v < n \wedge z = v$$

Next, the `zap` tactic traverses the CHC, *reducing* all the head applications of κ to $\top$ (*i.e.*, *True*) as they are guaranteed to hold by construction, and the body applications of κ with the computed solution. These reductions are highlighted in gray. The result is a horn variable free *verification condition* which is discharged by Lean's `grind` tactic.

Solving Cyclic Variables The Horn variable κ in c_{for} is cyclic: it appears as a hypothesis along the same path where it is the goal. Hence, we cannot apply the technique described above as it would loop indefinitely! Instead, Flex has a `fix` tactic that implements a form of abstract interpretation called *predicate abstraction* [21, 28]. This tactic proceeds in three steps.

Step 1: Qualifiers The `fix` tactic solves cyclic variables as *conjunctions of predicates* from a library of *qualifier* templates [62]. In Flex, a qualifier is any Lean predicate tagged with `@[qual]`, *e.g.*,

```
@[qual] def q_le (a b : Int) := a ≤ b
@[qual] def q_lt (a b : Int) := a < b
@[qual] def q_eq (a b : Int) := a = b
```

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

$$\kappa(z, hi, lo) \ \doteq\ \{z \le hi, hi \le z, z \le lo, lo \le z, \ldots z < hi, hi < z, z < lo, lo < z, \ldots\}$$

Step 3: Weaken till Fixpoint Finally, `fix` enters an iterative weakening loop where we use an *oracle* — which could be `grind` or any other theory specific decision procedure in Lean — to *discard* candidates that *cannot* be proven to hold at some CHC head. Once we have eliminated the impossible, whatever candidates remain, must be the truth. Hence, `fix` returns a solution comprising

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

$$
\begin{aligned}
&\exists \kappa : Int \rightarrow Int \rightarrow Int \rightarrow Int \rightarrow Prop \\
&\forall n : Int.(2 \leq n \ \wedge \ \mathtt{fib}(n) < \mathtt{MAX} \ \wedge \ n < \mathtt{MAX}) \rightarrow \\
&\quad \kappa(1, 2, 2, n) \\
&\quad \wedge \forall prev\ curr\ i : Int.\kappa(prev, curr, i, n) \ \rightarrow \\
&\qquad i \geq n \ \rightarrow \ curr = \mathtt{fib}(n) \\
&\qquad \wedge\ i < n \ \rightarrow \ prev + curr \leq \mathtt{MAX} \wedge \\
&\qquad\qquad \kappa(curr, prev + curr, i + 1, n)
\end{aligned}
$$

Fig. 4. (L) Triad of Constraints, Solutions and Certificates (§ 4.3); (R) CHC from `fib_loop` in Fig. 7.

the conjunction of all the candidates that remain standing. For our example, two candidates remain, yielding the solution $\kappa \doteq \lambda z, hi, lo.lo \leq z \wedge z \leq hi$. Like `zap`, `fix` uses the computed solution to reduce away κ by replacing head occurrences with $\top$ and body occurrences with the computed fixpoint, as shown in Fig. 3, yielding a plain VC that can be discharged by `grind`.

Certifying Reductions A key challenge that Flex addresses is that in the foundational setting, we cannot blithely rewrite the constraint with an alleged solution: instead, we must conclusively prove that the reduction is legitimate. Thus, whenever Flex's CHC solving tactics `zap` and `fix` reduce a constraint c to c' by computing the solution for a single variable κ, they traverse c and the solution to construct an explicit *certificate* that demonstrates that any proof of the reduced constraint c' yields a proof of the original c. To this end, we observe that the structure of constraints, solutions and hence, their certificates, form a triad summarized in Fig. 4 (and detailed in § 4.3). Their rhyming structure lets us mirror the traversal used to compute the solution to construct an explicit Lean proof term (using the matching constructs in Fig. 4) that justifies why it is sound to replace each head κ-application with $\top$ at that juncture in the constraint. These proofs are then chained together to show that any proof of the κ-free proposition is also a proof of the original CHC, yielding an end-to-end guarantee.

2.3 Foundational Verifiers

Our second contribution is to use the solvers to develop the first foundational CHC-based verifiers. First, a Floyd-Hoare style verifier [24, 35] for an imperative language Imp where the Horn variables let us synthesize *loop invariants* [6]. Second, a verifier for a core functional calculus extended with refinement types λ_{RK}, which uses Horn variables to infer unknown *refinements*. In each case, we formalize the source language's operational semantics and implement a CHC generator that is proved sound against the semantics. Flex can then discharge the generated CHCs, highlighting two key benefits: first that it can be used as a foundational "back-end" verifier for stateful or functional programs *in* Lean; second that it can reuse Lean's proof ecosystem to check specifications over arbitrary user-defined functions.

Loop Invariant Synthesis for Imp Fig. 5 shows our verifier for Imp in action. On the left, we first have a *specification* for `fib` as a plain Lean function, and below, a textbook imperative loop that computes the n^{th} Fibonacci number. We use Flex to prove the (partial) Floyd-Hoare triple that says that when the loop is executed from a state where the value of the variable `n` equals a with $a \geq 2$ then (if it exits) the value of `x` equals `fib`(a).

The proof of the triple is in two steps. First, the `generate` tactic reduces the assertion to the CHC shown on the top right. This CHC has a single cyclic Horn variable κ that represents the unknown loop invariant. The classical weakest-precondition based VC generation [18] yields a CHC with constraints that check (1) the invariant holds upon loop entry; (2) the invariant is inductive (re-established by the body) and (3) the specified postcondition holds on exit. Second,

```
@[grind] def fib (n : Int) : Int :=
  if n ≤ 1 then 1 else fib(n-1) + fib(n-2)
  termination_by n.toNat

@[qual] def q1 (v i:Int) := v = fib i
@[qual] def q2 (v i:Int) := v = fib (i-1)

theorem fibLoop (a : Int) :
  ⊨ (λs => s "n" = a ∧ a ≥ 2)
     <| prev := 1; x := 2; i := 2;
        while i < n do (
          next := prev + x ;
          prev := x ;
          x := next ;
          i := i + 1) |>
     (λs => s "x" = fib a)
  := by generate; fix; grind
```

$$
\begin{array}{l}
\exists \kappa : \mathit{Int} \to \mathit{Int} \to \mathit{Int} \to \mathit{Int} \to \mathit{Prop}. \\
\quad \forall n : \mathit{Int}. \\
\qquad n = a \;\to\; 2 \le a \;\to\; \kappa(n, 1, 2, 2) \\
\wedge\; \forall n\ p\ x\ i : \mathit{Int}. \\
\qquad \kappa(n, p, x, i) \;\to\; i < n \;\to\; \kappa(n, x, p + x, i + 1) \\
\wedge\; \forall n\ p\ x\ i : \mathit{Int}. \\
\qquad \kappa(n, p, x, i) \;\to\; n \le i \;\to\; x = \mathtt{fib}(a)
\end{array}
$$

$\xrightarrow{\texttt{fix}}$

$$
\begin{array}{l}
\quad \forall n : \mathit{Int}. \\
\qquad n = a \;\to\; 2 \le a \;\to\; \top \\
\wedge\; \forall n\ p\ x\ i : \mathit{Int}. \\
\qquad \ldots \;\to\; i < n \;\to\; \top \\
\wedge\; \forall n\ p\ x\ i : \mathit{Int}. \\
\qquad n = a \wedge i \le n \wedge p = \mathtt{fib}(i-1) \wedge x = \mathtt{fib}(i) \;\to \\
\qquad\quad n \le i \;\to\; x = \mathtt{fib}(a)
\end{array}
$$

Fig. 5. End-to-end Verification for Imp: (L) Specification of `fib` (top) and imperative implementation (bot), (R) Generated VC (top) and reduced constraint (bot).

Flex's Fix tactic reduces the CHC using predicate abstraction to synthesize a solution for the κ using the qualifiers shown above, yielding the constraint shown below, discharged by `grind`.

Refinement Synthesis for λ_{RK} Fig. 6 shows our verifier for λ_{RK} in action. On the left, we have a function `max` that returns the larger of its two inputs with a return type carrying an unknown refinement $\kappa(v, y, x)$. The expression `ex` applies `max` to 6 and 7. We use Flex to prove that `ex` can be typed as $\{v : \mathsf{Int} \mid v = 7\}$ under some typing environment K that assigns κ a concrete predicate.

The proof proceeds in two steps. First, the `generate` tactic runs λ_{RK}'s bidirectional type checker to emit the CHC shown on the top right. This CHC has a single *acyclic* Horn variable $\kappa : \mathit{Int} \to \mathit{Int} \to \mathit{Int} \to \mathit{Prop}$ representing the unknown return refinement. Type checking the body of `max` yields two head constraints: if $x \le y$ then the return value y must satisfy $\kappa(y, y, x)$, and symmetrically $\kappa(x, y, x)$ in the else branch. The call site `max 6 7` checked against $\{v : \mathsf{Int} \mid v = 7\}$ contributes the final clause: any v satisfying $\kappa(v, 7, 6)$ must equal 7. Second, Flex's Zap tactic reduces the CHC: since κ is acyclic, it computes its strongest solution, replaces each head occurrence with $\top$ and each body occurrence with the computed solution (highlighted in gray on the bottom right), yielding a plain VC that `grind` dispatches.

2.4 Expressive Refinement Typing

Our last contribution shows how embedding CHCs into Lean liberates refinement types from the limitations of SMT-solver theories, enabling their use for proving functional correctness properties of Rust using the Flux refinement type checker (§6).

Specifying Non-Overflow The left of Fig. 7 shows a Rust function that implements the same imperative fibonacci computation from Fig. 5. However, in the Rust setting, all the variables are fixed-sized (unsigned) integers `usize` and hence, Flux additionally requires us to prove that various arithmetic operations, notably `prev + curr`, do not overflow. The operation *will* overflow unless the `n` is sufficiently small, as stated by the additional precondition. Thus, when run on this code, Flux generates a CHC `FibLoopCHC` shown on the right in Fig. 4, that is similar to that in Fig. 5 but with an additional *assumption* that $\mathtt{fib}(n) < \mathtt{MAX}$ and a *head* that checks $\mathit{prev} + \mathit{curr} \le \mathtt{MAX}$.

```
max : x:Int -> y:Int -> {v:Int | κ(v,y,x)}
max ≐ λ x y ->
        let c = x ≤ y in
        if c then y else x

ex  ≐ let a = 6 in
      let b = 7 in
      max a b

example: ∃ K, ∅ ⊢K ex ⇐ {v:Int | v = 7}
  := by
     generate
     zap
     grind
```

$$
\begin{array}{l}
\exists \kappa : Int \to Int \to Int \to Prop. \\
\quad \forall x\ y : Int. \\
\qquad x \le y \to \kappa(y, y, x) \\
\quad \wedge\ \neg(x \le y) \to \kappa(x, y, x) \\
\wedge\ \forall v : Int. \\
\qquad \kappa(v, 7, 6) \to v = 7
\end{array}
$$

$$\xrightarrow{\text{zap}}$$

$$
\begin{array}{l}
\quad \forall x\ y : Int. \\
\qquad x \le y \to \top \\
\quad \wedge\ \neg(x \le y) \to \top \\
\wedge\ \forall v : Int. \\
\qquad (6 \le 7 \wedge v = 7) \vee (\neg 6 \le 7 \wedge v = 6) \to v = 7
\end{array}
$$

Fig. 6. End-to-end Verification for λ_{RK}: (L) Typing an λ_{RK} that computes the larger of two inputs, (R) CHC generated by typing.

```
fn fib_loop(n: usize) -> usize[fib(n)]
  requires 2 <= n && fib(n) < usize::MAX
{
    let mut prev = 1;
    let mut curr = 2;
    let mut i = 2;
    while i < n {
        let next = prev + curr;
        prev = curr;
        curr = next;
        i += 1;
    }
    return curr;
}
```

```
theorem fib_mono: ∀ (i j : Int),
  0 ≤ i → i ≤ j → fib i ≤ fib j
  := by
  intros i j hi hij
  induction j using fib_spec_fib.induct
  generalizing i with grind

def FibLoop_proof : FibLoopCHC := by
  unfold FibLoopCHC
  fix
  have _ : fib (i + 1) <= fib n := by
    apply fib_mono <;> grind
  grind
```

Fig. 7. Expressive Refinement Typing in Flux

Verifying Non-Overflow The right side of Fig. 7 shows how we can verify the code by proving `FibLoopCHC` in Lean: we need simply `unfold` and use the Flex tactic `fix`, which uses the qualifiers in Fig. 5 to solve for and reduce the κ, leaving behind the single goal

$$2 \le i < n, \mathtt{fib}(n) < \mathtt{MAX}, \ldots \vdash \mathtt{fib}(i-1) + \mathtt{fib}(i) \le \mathtt{MAX}$$

that we can discharge by proving and applying a lemma that `fib` is monotonic (`fib_mono`) in Lean. In the SMT based setting [44, 48] verification failures are opaque. The programmer would have to stare hard at the code to understand the root cause, then prove the monotonicity lemma using the spare (relative to Lean) affordances of the verifier, and finally instantiate the lemma by cluttering the source with assertions. In contrast, Lean's interactive context makes it easy to spot what is *missing*, and its ecosystem of theories and tactics makes it easy to then close the gap.

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	$::=$	φ	*κ-free atom*
		$\mid$	$\kappa(\overline{t})$	*horn application*
Constraints	c	$::=$	p	*head*
		$\mid$	$c \wedge c$	*conjunction*
		$\mid$	$p \to c$	*implication*
		$\mid$	$\forall x : b.\, c$	*quantification*
Closed Constraint	C	$::=$	$\exists \overline{\kappa}.c$	
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. 8. Grammar of the CHC fragment of Lean 4 `Expr` at sort *Prop*, together with the proof-term fragment emitted by our solvers.

3 Constraints and Proofs

Flex represents a system of *Constrained Horn Clauses* (CHCs) in a restricted fragment of Lean terms of type *Prop*. The syntactic structure preserves the information needed for Horn solving, while ensuring that key logical connectives like $\forall$, $\wedge$, $\to$ are native to Lean, thereby allowing us to implement the solver as an ordinary metaprogram over Lean's datatype *Expr*.

This representation provides two key benefits. First, it gives us *theories for free* as a Lean proposition can use formulas over arbitrary Lean theories to write specifications, allowing us to use arbitrary Lean definitions to synthesize invariants and use Lean proof automation — *e.g.* `omega`, `bv_decide`, `decide`, `grind` or lemmas [70] — to simplify verification, and, we do not need to maintain a deeply embedded AST for each theory, re-using *Expr* allows for inheriting theories of Lean, keeping the solver simple, but expressive. Second, it provides *soundness for free*, by letting us implement CHC solvers (§4) that emit proof terms which can then be checked by the kernel, guaranteeing the soundness of verification without trusting any code beyond the kernel itself.

As a running example throughout this section, consider a constraint that asserts $x \geq 0$ and an unknown refinement κ_1 relating x to its successor v:

$$c_0 \;\doteq\; \exists \kappa_1.\, \forall x.\, 0 \leq x \to (\forall v.\, v = x + 1 \to \kappa_1(v, x)) \wedge (\forall a.\, \kappa_1(a, x) \to 1 \leq a) \tag{1}$$

We will point back to c_0 as we introduce each piece of notation below.

3.1 Syntax and Semantics

Fig. 8 shows the syntax for CHCs, which follows a nested structure that preserves scoping information our solvers later use to recover the context of each Horn application (§ 4.2).

Sorts and Terms We assume a base family of *sorts* b which range over arbitrary Lean types like integers, bit-vectors, sets, maps *etc.* We write $x_1, x_2, x_3, \ldots$ for *variables* of sort b, and write $t_1, t_2, t_3, \ldots$ for arbitrary Lean terms of the appropriate sort. In c_0 above, x, v, and a are variables of sort *Int*, while $x + 1$ is a term of sort *Int* and $1 \leq a$ is a term of sort *Prop*. By convention, when a term is of sort *Prop*, we use the metavariable φ to denote it. Similarly, we use h to range over variables naming hypotheses, *i.e.*, of sort *Prop*. We assume terms include the trivial propositions $\top$ and $\bot$ (of sort *Prop*), which we will use to simplify constraints during solving (§4).

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 $\kappa(\bar{t})$. We assume variables κ do not appear inside arbitrary terms t. We will introduce such variables to represent *unknown* refinements, by existentially quantifying them in the constraints. For any constraint c, we write $\mathrm{KVars}(c)$ for the set of Horn variables occurring in c. In c_0, $\kappa_1 : \mathit{Int} \to \mathit{Int} \to \mathit{Prop}$ is a horn variable relating two integers.

Atoms An *atom* p is either a κ-free term t of sort *Prop*, or a *horn application* $\kappa(\bar{t})$. In c_0, $0 \le x$, $v = x + 1$, and $1 \le a$ are κ-free terms, while $\kappa_1(v, x)$ and $\kappa_1(a, x)$ are horn applications.

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. The running example c_0 is itself a closed constraint. In the sequel we abuse notation and write c for both open and closed constraints, as the intended meaning is always 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, the predicate $\sigma_1 \doteq \lambda v\, x.\ v \ge 1$ is an interpretation for κ_1, since $\sigma_1 : \mathit{Int} \to \mathit{Int} \to \mathit{Prop}$.

Satisfiability We *apply* an interpretation σ to a constraint c by replacing each κ-application $\kappa(\bar{t})$ with $\sigma(\bar{t})$; we write $c[\kappa := \sigma]$ for the result. Applying an interpretation for each κ in a closed constraint instantiates the existentials, leaving a κ-free proposition. A constraint c is *satisfiable* if there exist an interpretation for each κ such that applying them makes the constraint a valid proposition (*i.e.*, provable in LEAN). If an interpretation makes a constraint satisfiable we call it a *solution*. For example, applying σ_1 to c_0 (and β-reducing) replaces $\kappa_1(v, x)$ with $v \ge 1$ and $\kappa_1(a, x)$ with $a \ge 1$ leaving the following verification condition:

$$\forall x.\ 0 \le x \to (\forall v.\ v = x + 1 \to v \ge 1) \wedge (\forall a.\ a \ge 1 \to 1 \le a)$$

This VC is valid; hence σ_1 is a solution of κ_1 and c_0 is satisfiable.

Theories for free As terms and predicates are LEAN terms, they can use any theory expressible in LEAN. For example, arithmetic inequalities $x \le y$ over *Int* sorted variables x and y, mask equations over *BitVec n*, or equations over user-defined functions $v = \mathtt{fib}(i)$.

3.2 Structural Operations

To solve a constraint, the solvers in §4 need to know, for each Horn variable, the smallest enclosing scope and the hypotheses visible there; we now define the machinery to compute this.

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 $h\!:\!p$ (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 $c \uparrow \rho$ (Fig. 9) computes the *context* of a constraint c at prefix ρ, by accumulating the bindings and guards along ρ. The procedure $c \downarrow \rho$ (Fig. 9) returns the sub-constraint of c at the end of ρ. The *truncated context* $\Gamma \setminus \rho$ is the context obtained by dropping the prefix of Γ that matches 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, *i.e.*, $\mathrm{heads}(c) \doteq \{\kappa \mid \kappa(\bar{t}) = c \downarrow \rho \text{ for some head-prefix } \rho\}$. A

$\text{Scope} \;:\; (\mathcal{K} \times C) \to \mathit{Prefix}$		
$\text{Scope}(\kappa, \forall x{:}b.\, c)$	$\doteq$	$\mathsf{B} \cdot \text{Scope}(\kappa, c)$
$\text{Scope}(\kappa, c_1 \wedge c_2)$		
$\mid\ \kappa \in c_1,\ \kappa \notin c_2$	$\doteq$	$\mathsf{L} \cdot \text{Scope}(\kappa, c_1)$
$\mid\ \kappa \notin c_1,\ \kappa \in c_2$	$\doteq$	$\mathsf{R} \cdot \text{Scope}(\kappa, c_2)$
$\text{Scope}(\kappa, p \to c)$		
$\mid\ \kappa \notin p$	$\doteq$	$\mathsf{G} \cdot \text{Scope}(\kappa, c)$
$\text{Scope}(\kappa, c)$	$\doteq$	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. 9. $\text{Scope}(\kappa, c)$ 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.

goal $\Gamma \vdash p$ is a context Γ paired with a head p. For a head-prefix ρ, the goal of c at ρ is defined as $\text{Goal}(c, \rho) \doteq c\uparrow\rho \vdash c\downarrow\rho$. For example, in the running constraint c_0, we have:

$$\text{Goal}(c_0, \mathsf{B} \cdot \mathsf{G} \cdot \mathsf{L} \cdot \mathsf{B} \cdot \mathsf{G}) \;=\; x{:}\mathit{Int}; h_0{:}0 \le x; \mathsf{L}; \nu{:}\mathit{Int}; h_1{:}\nu = x + 1 \vdash \kappa_1(\nu, x)$$

Given a context Γ and term φ, we say the goal $\Gamma \vdash \varphi$ is *valid* if φ is derivable under the bindings and guards in Γ. A goal is a *κ-goal* if its head is $\kappa(\bar{t})$; an interpretation σ is a solution for such a goal if the goal $\Gamma \vdash \sigma(\bar{t})$ is valid. For example, the interpretation $\sigma_1 \doteq \lambda\nu\, x.\ \nu \ge 1$ is a solution for the goal above, since $\nu \ge 1$ is valid under the assumptions $0 \le x$ and $\nu = x + 1$.

Scopes and Well-formedness The procedure $\text{Scope}(\kappa, c)$, shown in Fig. 9 computes the *scope* of a variable κ in a constraint c. Informally, $\text{Scope}(\kappa, c)$ returns a prefix ρ corresponding to a path from the root of c to the *smallest sub-constraint* of c that contains *all* the applications of κ in c. The scope is more interesting once a constraint has more than one Horn variable, so consider extending c_0 with a second variable κ_2 and a sub-constraint nested under the guard $\kappa_1(a, x)$. For the constraint c_1 below, we get the scopes shown on the right ($\wedge$ is right associative).

$$c_1 \;\doteq\; \begin{array}{l} \exists\kappa_2, \kappa_1.\ \forall x.\ 0 \le x \to \\ \qquad \forall \nu.\ \nu = x + 1 \to \kappa_1(\nu, x) \\ \qquad \wedge\ \forall a.\ \kappa_1(a, x) \to \\ \qquad\quad 1 \le a \\ \qquad\quad \wedge\ \kappa_2(a + 1, a, x) \\ \qquad\quad \wedge\ \forall \nu.\ \kappa_2(\nu, a, x) \to 2 \le \nu \end{array} \qquad \begin{array}{lcl} \text{Scope}(\kappa_1, c_1) & \doteq & \mathsf{B} \cdot \mathsf{G} \\ \text{Scope}(\kappa_2, c_1) & \doteq & \mathsf{B} \cdot \mathsf{G} \cdot \mathsf{R} \cdot \mathsf{B} \cdot \mathsf{G} \cdot \mathsf{R} \end{array} \tag{2}$$

Well-formedness The solver in §4 assumes that each Horn variable's arguments are passed consistently with the binders in scope; we call this property well-formedness. Formally, let $\rho \doteq \text{Scope}(\kappa, c)$, $\Gamma \doteq c\uparrow\rho$, and $\text{dom}(\Gamma) \doteq x_0, \ldots, x_n$ the binders of Γ listed innermost first. We say that c is *well-formed* for κ if every application of κ in c has the form $\kappa(\ldots, x_0, \ldots, x_n)$, *i.e.*, its trailing arguments are exactly the binders $\text{dom}(\Gamma)$ in scope. Its remaining leading arguments are arbitrary terms, we write $\text{rank}(\kappa)$ for their number. A constraint c is *well-formed* if it is well-formed for all $\kappa \in \text{KVars}(c)$. It is straightforward to ensure well-formedness by simply *padding* the parameters of each κ. Thus, in the sequel, for simplicity, we assume that all constraints are well-formed. Constraint c_1 in Eq. (2) is well-formed as x is the only binder in the scope of κ_1, and each application of κ_1 has x as argument, while the binders in the scope of κ_2 are a and x, and each application of κ_2 has them as arguments.

3.3 Proof Terms

The solvers in §4 emit proofs to justify their rewrites. Fig. 8 shows the fragment of Lean's terms that the solvers emit as proofs. The fragment comprises exactly the introduction and elimination forms of the constraint logic (Fig. 8) and the disjunctions and existentials that arise in the solutions σ computed by our solvers. Specifically, the fragment contains terms t; λ-terms and applications for introducing and eliminating $\forall$ and implication; pairs $\langle q, q' \rangle$ and projections $q.1$, $q.2$ for conjunction, together with inl and inr for disjunction; existential witnesses $\langle t, q \rangle$, unpacking terms let $\langle x, h \rangle = q$ in q', and the constants rfl and $\langle\rangle$ for equality and $\top$. The typing of these terms is inherited from the Lean kernel, which re-checks every certificate our solvers emit. These are the standard notations for Lean's logical connectives, and the same core is shared by other type-theoretic proof assistants such as Rocq [69] and Agda [57], so the certifying solvers of §4 are not tied to Lean.

4 Solvers

Flex uses a three-step approach to solving Horn constraints. First, we *partition* the horn variables into a cut set and an acyclic set (§ 4.1). Second, we *eliminate* the acyclic variables by substituting them with their strongest solutions (§ 4.2). Third, we perform a *fixpoint* computation to find a conservative approximation for the remaining cyclic variables (§ 4.4). These steps are implemented in two Lean tactics — `zap` and `fix` — which *rewrite* the original horn constraint by eliminating the acyclic and cut variables respectively, leaving behind a κ-free VC which can be discharged using Lean tactics like `simp`, `omega` or SMT-inspired ones like `grind`. While the algorithms used in `zap` and `fix` are inspired by prior work, in the foundational setting they must also *certify* the soundness of the rewrites. Next, we describe how Flex implements partitioning (§ 4.1), and the two tactics, `zap` (§ 4.2) and `fix` (§ 4.4), in a certifying fashion, to obtain a foundational CHC solver.

4.1 Partitioning Horn Variables

Dependencies The *dependencies* of a constraint c is a subset of $\mathcal{K} \times \mathcal{K}$ defined by the function

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

We say κ' *depends on* κ in c if $(\kappa, \kappa') \in \mathsf{Dep}(c)$. Informally, this means that κ occurs in a guard p along some path to a head containing κ'. The *dependency graph* of a constraint c is the directed graph $\mathsf{Graph}(c)$ whose vertices are $\mathsf{KVars}(c)$ and which has an edge from κ to κ' if κ' depends on κ in c. Intuitively, if κ' depends on κ, then we need to solve for κ *before* we can solve for κ'.

Cycles, Cuts and Partitions A set of variables $\overline{\kappa} \subseteq \mathsf{KVars}(c)$ is a *cut set* for c if it corresponds to a *feedback vertex set* of $\mathsf{Graph}(c)$, *i.e.*, if removing the $\overline{\kappa}$ vertices and their incident edges renders $\mathsf{Graph}(c)$ acyclic [40]. We say that $\overline{\kappa_C}, \overline{\kappa_A}$ *partitions* the horn variables of a constraint c, if (1) $\overline{\kappa_C}$ and $\overline{\kappa_A}$ are disjoint, (2) $\overline{\kappa_C} \cup \overline{\kappa_A} = \mathsf{KVars}(c)$, (3) $\overline{\kappa_C}$ is a cut set for c. We say informally that $\overline{\kappa_A}$ is the set of *acyclic* variables. We define $\mathsf{Partition}(\overline{\kappa}, c)$ to be a function that computes the dependencies of c, builds a dependency graph, and computes a feedback vertex set to return a partition of the horn variables of c. The acyclic set $\overline{\kappa_A}$ is returned in *topological* order: each variable is bound outside the variables it depends on, so the innermost-first elimination of § 4.2 removes a variable's dependencies before the variable itself, and every computed solution mentions only cut variables. Computing a *minimum* feedback vertex set is NP-complete [40], so $\mathsf{Partition}(\overline{\kappa}, c)$ uses a greedy heuristic based on Tarjan's algorithm [68]; minimality affects only the residual CHC size, not soundness.

4.2 Solving Acyclic Variables

Top-level Algorithm (Zap**):** Fig. 10 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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 10. Zap eliminates acyclic horn variables (left) using Elim to compute solutions (right). The guarded first equation of Sol collapses sub-constraints without κ head applications to $\bot$, so the solution is κ-free, and all its free variables are among the $\overline{z}, \overline{x}$ bound in Elim.

$$
\begin{array}{l}
\exists \kappa_2, \kappa_1. \\
\forall x.\ 0 \le x \to \\
\quad \forall v.\ v = x + 1 \to \kappa_1(v, x) \\
\wedge\ \forall a.\ \kappa_1(a, x) \to \\
\quad 1 \le a \\
\quad \wedge\ \kappa_2(a + 1, a, x) \\
\quad \wedge\ \forall v.\ \kappa_2(v, a, x) \to \\
\quad\quad 2 \le v
\end{array}
\xrightarrow[\lambda z_0\, x.\, z_0 = x+1]{\text{reduce } \kappa_1}
\begin{array}{l}
\exists \kappa_2. \\
\forall x.\ 0 \le x \to \\
\quad \forall v.\ v = x + 1 \to \top \\
\wedge\ \forall a.\ a = x + 1 \to \\
\quad 1 \le a \\
\quad \wedge\ \kappa_2(a + 1, a, x) \\
\quad \wedge\ \forall v.\ \kappa_2(v, a, x) \to \\
\quad\quad 2 \le v
\end{array}
\xrightarrow[\lambda z_0\, a\, x.\, z_0 = a+1]{\text{reduce } \kappa_2}
\begin{array}{l}
\forall x.\ 0 \le x \to \\
\quad \forall v.\ v = x + 1 \to \top \\
\wedge\ \forall a.\ a = x + 1 \to \\
\quad 1 \le a \\
\quad \wedge\ \top \\
\quad \wedge\ \forall v.\ v = a + 1 \to \\
\quad\quad 2 \le v
\end{array}
$$

Fig. 11. Eliminating acyclic variables with Zap. (L) initial constraint from Eq. (2), (C) result of reducing κ_1, and (R) result of reducing κ_2 yielding a VC with no Horn variables.

acyclic variables have been removed, and a *proof term* q' that shows that the output constraint *entails* the original constraint. First, Zap partitions the horn variables $\overline{\kappa}$ into a cut set $\overline{\kappa_C}$ and an acyclic set $\overline{\kappa_A}$. Next, Zap invokes Zaps, which recurses over the acyclic prefix: it peels the outermost κ, recursively eliminates the inner variables to obtain a residual c_0, and invokes Zap1 to eliminate κ from c. Zaps then composes the proofs $q_2 : c_2 \to \exists \kappa.\, c_1$ and $q_1 : c_1 \to c$ into a proof $q_3 : c_2 \to \exists \kappa.\, c$ 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

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

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

Fig. 12. $\mathsf{Cert}^{\kappa}_{\sigma,\rho}$ 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 $\vee/\exists/\wedge$ structure of the solution σ.

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 (§ 4.3): it maps any proof h of c' to a proof of $\exists\kappa.\, c$, with σ as the witness. Fig. 11 illustrates this process on the constraint c_1 from Eq. (2).

Variable Elimination (Elim**):** The procedure $\mathsf{Elim}(\kappa, c)$, shown in Fig. 10, 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 $\top$ — as they are guaranteed to hold by virtue of how σ was computed — and all other non-head occurrences of $\kappa(\bar{t})$ are replaced by the solution $\sigma(\bar{t})$, yielding a constraint *without* κ.

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. 10, 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. 11 shows below each arrow, the solution computed for the κ reduced in the respective step. In the sub-constraint for κ_1 there is exactly one head application, with the (truncated) context $\nu\!:\!\mathit{Int}; \nu = x + 1$; the conjunct with the *uses* of κ_1 collapses to $\bot$. Thus, the computed solution is $\lambda z_0\, x.(\exists \nu.\ \nu = x + 1 \wedge z_0 = \nu) \vee \bot$ which we simplify to $\lambda z_0\, x.z_0 = x + 1$. As such, we can check that this procedure returns the *strongest solution* for κ in c, meaning, it returns a solution σ that entails *every* other valid solution for κ in c.

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 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.

4.3 Certification

In a foundational setting, one does not simply rewrite a constraint. The Lean kernel, ultimately, demands a proof of the original constraint, not the rewritten one. To this end, zap uses $\mathsf{Cert}^{\kappa}_{\sigma,\rho}$ to produce a term that *converts* any 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. 12) 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 original and reduced constraints diverge only when the original head is of the form $\kappa(\bar{t})$ but the reduced head is $\top$; in that case, q is vacuous. Other heads p are unchanged so we can directly reuse the corresponding proof term q.

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. Nav replays the truncated context against the structure of σ, whose $\vee/\exists$ shape mirrors the scoped sub-constraint. Each context entry selects the matching introduction form: a branch selects inl or inr, a bound variable supplies an existential witness, and a guard hypothesis supplies a conjunct. The descent bottoms out at the slot equalities $\bigwedge_i z_i = t_i$, using rfl to discharge them.

Example In the example in Fig. 11, κ_1 has the scope prefix $\rho \doteq \mathsf{B} \cdot \mathsf{G}$. Thus, at the head $\kappa_1(v, x)$, $\mathsf{Cert}^{\kappa}_{\sigma,\rho}$ 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:

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

Correctness The correctness of Zap is stated via two theorems. The first one, from [14], says that $\mathsf{Zap}(c)$ returns a constraint c', without any acyclic horn variables, that is logically equivalent to c.

Theorem 4.1 (Zap-Equivalence). *[14] If $(c', \cdot) = \mathsf{Zap}(c)$ then c is satisfiable iff c' is satisfiable.*

In the foundational setting, we also crucially depend on the following soundness result, which says that the proof returned by $\mathsf{Zap}(c)$ explicitly witnesses the correctness of the rewrite, *i.e.*, it allows us to convert any proof of c' into one of c:

Theorem 4.2 (Zap-Soundness). *If $(c', q) = \mathsf{Zap}(c)$ then $\vdash q : c' \to c$.*

4.4 Solving Cyclic Variables

Next, let us see how the `fix` tactic solves the cyclic variables by *predicate abstraction* [28], which works by solving each κ to a conjunction of predicates drawn from a library of user-supplied *qualifiers*. Hitherto, predicate abstraction has typically been carried out using SMT solvers [21, 62], where it is limited to whatever theories the SMT engine can handle efficiently. The foundational setting provides two significant benefits. First, we can use arbitrary Lean *predicates* as qualifiers,

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

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

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

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

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

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

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

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

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

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

Fig. 13. The Fix procedure for reducing cyclic horn variables via predicate abstraction.

and, correspondingly, second, we can use arbitrary Lean automation (including, of course, SMT-based tactics) to implement user-definable *oracles* that can allow synthesizing invariants over arbitrary theories. Next, we illustrate the above by describing how `fix` ticks.

Qualifiers, Instances and Candidates The predicate abstraction procedure PredAbs uses a set of qualifiers to compute a solution for each κ. Informally, qualifiers are predicate *templates*: PredAbs computes solutions by *conjoining* all *valid* instances of the 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 (1) ι is a map $[0, n] \to [0, m]$, where (2) for each $0 \le j \le n$, we have $b_j = b'_{\iota(j)}$. That is, an instance ι maps each parameter of the qualifier to a parameter of the κ of the same sort. A *candidate* is a pair of a qualifier q and an instance ι written as q_ι. We overload q_ι to also denote the predicate it induces by instantiating q along ι, namely $q_\iota \doteq \lambda z_0 \cdots z_m.q(z_{\iota(0)}, \ldots, z_{\iota(n)})$. 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 $\iota_1 \doteq \{0 \mapsto 0,\ 1 \mapsto 0\}$, $\iota_2 \doteq \{0 \mapsto 0,\ 1 \mapsto 1\}$, $\iota_3 \doteq \{0 \mapsto 1,\ 1 \mapsto 0\}$ and $\iota_4 \doteq \{0 \mapsto 1,\ 1 \mapsto 1\}$, yielding the candidate set containing the four candidates

$$(q_\le)_{\iota_1} \doteq \lambda z_0\, z_1.z_0 \le z_0 \quad (q_\le)_{\iota_2} \doteq \lambda z_0\, z_1.z_0 \le z_1 \quad (q_\le)_{\iota_3} \doteq \lambda z_0\, z_1.z_1 \le z_0 \quad (q_\le)_{\iota_4} \doteq \lambda z_0\, z_1.z_1 \le z_1$$

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 $\kappa(\overline{t})[\theta]$, by applying the pointwise conjunction of the candidate set $\theta(\kappa)$. We lift application to contexts $\Gamma[\theta]$ by applying to each hypothesis in Γ.

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

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.

Top-level Algorithm (Fix**):** Fig. 13 summarizes the top-level Fix algorithm, which takes as input a constraint c and a set of qualifiers Q and returns as output a pair (c', q) comprising a rewritten constraint c' where all the horn variables in c are removed, and a proof q that shows that the rewritten c' implies the original c. The procedure Fix is analogous to Zap: first, it invokes PredAbs on the constraint and qualifiers to obtain an assignment θ, this time for *all* the Horn variables simultaneously, as they are cyclic, and hence depend upon each other. 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 that $c' \rightarrow c$.

Predicate Abstraction (PredAbs**):** The right column in Fig. 13 shows how PredAbs takes as input a set of horn variables $\overline{\kappa}$, a constraint c, and a set of qualifiers Q, to compute an assignment that is valid for c. This assignment is computed in two steps. First, we use Init to compute an *initial assignment* that maps each κ to the full set of candidates obtained from all κ/q instances of each qualifier in Q. Second, we invoke Fixpoint to distill the full set of candidates down into those that can be proven valid for each head κ-application (which lets Fix subsequently reduce to $\top$).

Fixpoint Computation (Fixpoint**):** The procedure Fixpoint, summarized in Fig. 13, takes as input the constraint c and a candidate assignment θ, and iteratively invokes Weaken to whittle away from θ the candidates that *cannot be proven* valid. The procedure Weaken takes as input the constraint c, head-prefix ρ and assignment θ, and uses the Oracle to determine the *subset of candidates* of $\theta(\kappa)$, named I, that can indeed be proven valid at the head corresponding to ρ. If this subset is strictly smaller than $\theta(\kappa)$, *i.e.,* some whittling occurred, then Weaken returns the assignment where κ is updated to I; otherwise (if the subset is unchanged), Weaken returns $\bot$. Now Fixpoint checks if there is *any* head-prefix ρ under which the (current) assignment θ gets weakened to θ'. If so, Fixpoint recurses on θ'. Otherwise, θ is valid for c and hence, is returned as the assignment. Fixpoint terminates: the initial assignment is finite, and each successful Weaken strictly decreases the total number of candidates. Note that Weaken, and hence Fix, is parameterized by the procedure Oracle, which attempts to *prove* a goal, which lets the user plug in theory-specific oracles to synthesize solutions for CHCs over arbitrary Lean propositions.

Certification (Cert_θ**):** Finally, Fix invokes Cert_θ to justify the rewriting done by Red_θ (where head κ-applications are replaced with $\top$), by coughing up a *Proof* that shows that validity of the reduced constraint c' *implies* that of the original c. This procedure is nearly identical to the one for Zap discussed in § 4.3, in how it traverses the structure of c replaying in the proof from c'. The only difference is at the head applications $\kappa(\bar{t})$ where (instead of navigating the structure of the solution) we invoke the 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 4.3 (Fix-Soundness)**.** *If* Oracle *is sound and* $\mathsf{Fix}(c, Q) = (c', q)$*, then* $\vdash q : c' \rightarrow c$.

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 4.4 (Fix-Completeness [62])**.** *If* Oracle *is complete and* $\theta \doteq \mathsf{PredAbs}(\exists \overline{\kappa}.\, c, Q)$*, then* $\theta \preceq \theta'$ *for every* Q*-assignment* θ' *with* $\theta' \models c$.

$$
\begin{array}{lll}
\textbf{\textit{Variables}} & x & : \; \mathit{Var} \text{ (alias for } \mathit{String}) \\
\textbf{\textit{States}} & s & : \; \mathit{Var} \to \mathit{Int} \\
\textbf{\textit{Expressions}} & e & : \; \mathit{State} \to \mathit{Int} \\
\textbf{\textit{Guards}} & g & : \; \mathit{State} \to \mathit{Bool} \\
\textbf{\textit{Assertions}} & P & : \; \mathit{State} \to \mathit{Prop} \\
\textbf{\textit{Commands}} & c & ::= \mathsf{skip} \mid x := e \mid c_1; c_2 \mid \mathsf{if}\ g\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ g\ \mathsf{do}\ c
\end{array}
$$

Fig. 14. Syntax of Imp, shallowly embedded into Lean

5 Foundational Verifiers

A CHC-based program verifier works in two stages, both of which are traditionally trusted: a frontend *generator* that takes a program and generates a constraint whose validity implies the program's safety; a backend *solver* that then determines the validity of the constraint. The solver in §4 removes the backend from the trusted base by generating a kernel checkable proof of the constraint's validity. Next, we show there is no need to trust the frontend either, by mechanizing the constraint generation such that a proof from the solver suffices to verify the source program. To this end, we develop two verifiers. First, in § 5.1, we present a Floyd-Hoare style verifier [24, 35] for an imperative language Imp with mutable state, where the Horn variables let us synthesize *loop invariants* [7]. Second, we present functional calculus λ_{RK} with refinement types (§ 5.2), use them to develop a sound declarative type system (§ 5.3), and use that to implement an algorithmic generator (§ 5.4) that uses Horn variables to infer unknown *refinements*. In each case, we can then discharge the generated CHCs using the tactics from §4 to obtain an end-to-end foundational verifier.

5.1 A Verifier for Imp

To limber up, we implement a CHC based Floyd-Hoare style verifier for Imp summarized in Fig. 14. We shallowly embed Imp programs so states s are functions $\mathit{Var} \to \mathbb{Z}$, expressions e and guards g are state-indexed, and assertions P, Q are predicates $\mathit{State} \to \mathit{Prop}$. A (Floyd-Hoare) *triple* comprising a pre-condition P, command c and post-condition Q is *valid*, written $\models \{P\}\ c\ \{Q\}$, if every terminating run of c from a P-state ends in a Q-state.

Constraint Generator Let $V \doteq \{x_1, \ldots, x_n\}$ be a set of n ordered variables that occur in commands. The generator $\mathsf{VC}(P, c, Q)$ takes as input a triple P, c, Q (where c uses variables from V) and outputs a CHC. The implementation is the textbook *weakest precondition*-based VC-generation method [18], except in two places. First, we do not require explicitly provided invariants for loops: when the generator hits a while it introduces a Horn *variable* κ for the unknown invariant — a $|V|$-ary predicate — and *constrains* it to satisfy the usual initial, body and exit obligations:

$$
\begin{array}{rcll}
\mathsf{VC}(P, \mathsf{while}\ g\ \mathsf{do}\ c\ , Q) & \doteq & \exists\, \kappa : \mathit{Int}^{|V|} \to \mathit{Prop}. & \\
 & & \quad \forall s.\ P(s) \to [\![\kappa]\!](s) & \text{(initial)} \\
 & & \quad \wedge\ \mathsf{VC}(\lambda s.[\![\kappa]\!](s) \wedge g(s), c, [\![\kappa]\!]) & \text{(body)} \\
 & & \quad \wedge\ \forall s.\ [\![\kappa]\!](s) \to \neg g(s) \to Q(s) & \text{(exit)} \\
\text{where} \quad [\![\kappa]\!] & \doteq & \lambda s.\kappa(s(x_1), \ldots, s(x_n)) &
\end{array}
$$

The above constraints are not in the syntax from Fig. 8 as we are quantifying over states s and not base-sorted values. Fortunately, Lean's `simp` tactic suffices to reduce them to our grammar.

Soundness We prove in Lean (Theorem D.3) that whenever the CHC returned by $\mathsf{VC}(P, c, Q)$ is satisfiable, that the corresponding triple is valid.

$$x, y, v \in \mathit{Var} \qquad \kappa \in \mathit{Kvar} \qquad z \in \mathit{Int}$$

Base sort	$\mathit{Base} \ni b$	$::= \mathsf{Int} \mid \mathsf{Bool}$
Term		$t_b ::= x_b \mid z \mid \mathsf{true} \mid \mathsf{false} \mid t_{\mathsf{Int}} + t_{\mathsf{Int}}$
Formula		$\varphi ::= \top \mid t_b =_b t_b \mid t_{\mathsf{Int}} \leq t_{\mathsf{Int}} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \rightarrow \varphi \mid \forall x{:}b.\, \varphi$
Refinements		$r ::= \varphi \mid \kappa(\overline{t_b})$
Type		$\tau ::= \{\, v{:}b \mid r \,\} \mid x{:}\tau \rightarrow \tau$
Expression		$e ::= v \mid x \mid \oplus(\overline{e}) \mid e\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid (e : \tau)$
Values	$\mathit{Val} \ni v$	$::= z \mid \mathsf{true} \mid \mathsf{false} \mid \lambda x.\, e$
Type env.		$\Gamma ::= \cdot \mid \Gamma,\ x{:}\tau$

Fig. 15. Syntax of λ_{RK}.

Theorem 5.1 (VC-Generation soundness)**.** *If* $\mathsf{VC}(P, c, Q)$ *then* $\models \{P\}\ c\ \{Q\}$.

5.2 Syntax and Semantics of λ_{RK}

Fig. 15 summarizes the syntax of λ_{RK}, which extends the simply typed λ-calculus with refinement types [9, 38], in particular, with Horn variables κ that enable refinement inference via CHC solving.

Expressions and Evaluation The grammar follows a standard call-by-value language with arithmetic and boolean expressions. Let bindings are required because the type system enforces ANF (A-Normal Form [23]) to accommodate dependent function applications [38]. We give expressions a standard, substitution-based, big-step, call-by-value operational semantics, written $e \Downarrow v$ (e evaluates to v). Type annotations are erased at runtime—$(e : \tau)$ evaluates as e—so refinements play no part in evaluation and serve only for static checking.

Refinements We build refinement types on top of a *separate* first-order language, to eschew the circularities in the meta-theory (where types would depend on expressions, which would themselves contain types). This first-order language is stratified into intrinsically typed *terms* (t_b) and *formulas* (φ). A refinement r is either a first-order formula (φ) over integers and booleans, or a Horn application $\kappa(\overline{t_b})$ —highlighted in gray in Fig. 15—representing existentially quantified predicates over their arguments $\overline{t_b}$ that the solver must infer.

Types A type τ is either a *refined base type* $\{\, v{:}b \mid r \,\}$ or a *dependent arrow* $x{:}\tau \rightarrow \tau$, which names its argument x so the codomain may mention it. A refined base type restricts its sort $b \in \{\mathsf{Int}, \mathsf{Bool}\}$ to the values v satisfying the refinement r (which may be an unknown Horn application).

Next, we provide a semantics for refinements by *interpreting* them as Lean propositions, which provides a foundation for declarative typing (§ 5.3) and constraint generation (§ 5.4).

Interpreting Formulas We interpret each base type b as a Lean type, writing $[\![b]\!]$, where $[\![\mathsf{Int}]\!]$ (resp. $[\![\mathsf{Bool}]\!]$) denotes the Lean integers *Int* (resp. booleans *Bool*). A term t_b denotes an element of $[\![b]\!]$ under a closing substitution $\gamma : \mathit{Var} \rightarrow \mathit{Val}$ that maps its free variables to values, written $[\![t_b]\!]_\gamma$. A formula denotes a Lean proposition over values and we write $[\![\varphi]\!]_\gamma$ for its interpretation as a Lean *Prop*. The interpretation is the natural one mapping constructs to their Lean counterparts.

Interpreting Refinements To give meaning to a refinement r we must also interpret Horn applications $\kappa(\overline{t_b})$. We do this by quantifying at the meta level over a *K-assignment* which fixes the meaning of each κ as a Lean predicate. Crucially, this K will itself map the *syntactic* horn variables to existentially bound Lean predicates that the solver will then instantiate. Concretely,

a K-assignment has the following Lean type $Kvar \to List\,(\Sigma\, b : Base, [\![b]\!]) \to Prop$. Given a K-assignment, a Horn application is interpreted by looking up the predicate K assigns to κ and applying it to the interpreted arguments:

$$[\![\kappa(\overline{t_b})]\!]^K_\gamma \doteq K\ \kappa\ \overline{(b, [\![t_b]\!]_\gamma)}$$

Interpreting Types Finally, we interpret types as predicates on values. We write $v \in [\![\tau]\!]^K_\gamma$ to mean that the value v inhabits the interpretation of τ. A refined base type denotes the base values satisfying the (interpretation) of its refinement, and a dependent arrow denotes the lambdas that send every argument in the domain to a result in the codomain:

$$v \in [\![\{\ v{:}b \mid r\ \}]\!]^K_\gamma \doteq v \in [\![b]\!]\ \wedge\ [\![r]\!]^K_{\gamma[\,v \mapsto v]}$$
$$v \in [\![x{:}\tau_1 \to \tau_2]\!]^K_\gamma \doteq v = \lambda x.\, e\ \wedge\ \forall v_a.\, v_a \in [\![\tau_1]\!]^K_\gamma \Rightarrow \exists v_r.\, e[v_a/x] \Downarrow v_r\ \wedge\ v_r \in [\![\tau_2]\!]^K_{\gamma[x \mapsto v_a]}$$

5.3 Declarative Typing for λ_{RK}

We define a declarative typing judgment $\Gamma \vdash_K e : \tau$ which is mostly unremarkable except for being parameterized by a K-assignment. We highlight a couple of aspects of the system:

Typing Variables with Selfification When typing variables we strengthen its refinement by giving its *selfified* type [59], which crucially enables path-sensitive "occurrence" typing [71].

$$\frac{(x{:}\tau) \in \Gamma}{\Gamma \vdash_K x : \mathrm{self}(x, \tau)}\ \textsc{T-Var} \qquad \begin{array}{l}\mathrm{self}(x, \{\ v{:}b \mid p\ \}) \doteq \{\ v{:}b \mid p \wedge v = x\ \} \\ \mathrm{self}(x, y : s \to t) \doteq y : s \to t\end{array}$$

Typing Applications The typing judgment enforces ANF so that a function is always applied to a *variable*, which can be substituted into the dependent codomain without placing an arbitrary expression inside a refinement. (The alternative is to extend the language with *existential types* [41], which exchanges the restriction for more complex subtyping and metatheory.)

$$\frac{\Gamma \vdash_K e : x{:}\tau_1 \to \tau_2 \qquad \Gamma \vdash_K y : \tau_1}{\Gamma \vdash_K e\ y : \tau_2\,[y/x]}\ \textsc{T-App}$$

Subtyping Most rules in the declarative system are syntax directed; subtyping is the only place where refinements are actually compared, so it is the place where essentially all of the interesting checking happens. A subsumption rule lets an expression of type τ_1 be used at any supertype τ_2, deferring to a subtyping judgment $\Gamma \vdash_K \tau_1 <: \tau_2$. For arrows it is the standard rule, contravariant in the domain and covariant in the codomain; for two refined base types, subtyping holds when the first refinement implies the second under the assumptions in the context:

$$\frac{\forall \gamma, (\!|\Gamma|\!)^K_\gamma \to [\![r_1]\!]^K_\gamma \to [\![r_2]\!]^K_\gamma}{\Gamma \vdash_K \{\ v{:}b \mid r_1\ \} <: \{\ v{:}b \mid r_2\ \}}\ \textsc{S-Base} \qquad \frac{\Gamma \vdash_K \tau_1' <: \tau_1 \qquad \Gamma, x{:}\tau_1' \vdash_K \tau_2 <: \tau_2'}{\Gamma \vdash_K x{:}\tau_1 \to \tau_2 <: x{:}\tau_1' \to \tau_2'}\ \textsc{S-Fun}$$

Here $(\!|\Gamma|\!)^K_\gamma$ *extracts* the assumptions from Γ by conjoining the base refinements

$$(\!|\cdot|\!)^K_\gamma \doteq \top \qquad (\!|\Gamma, x{:}\{\ v{:}b \mid r\ \}|\!)^K_\gamma \doteq (\!|\Gamma|\!)^K_\gamma \wedge [\![r[x/v]]\!]^K_\gamma \qquad (\!|\Gamma, x{:}\tau_1 \to \tau_2|\!)^K_\gamma \doteq (\!|\Gamma|\!)^K_\gamma$$

Soundness We prove the declarative system sound: a well-typed program never gets stuck, evaluating to a value that inhabits the interpretation of its type.

Theorem 5.2 (Type soundness). *If $\cdot \vdash_K e : \tau$, then there exists a value v such that $e \Downarrow v$ and $v \in [\![\tau]\!]^K_\emptyset$.*

5.4 A Verifier for λ_{RK}

The declarative system assumes a K-assignment with the solutions for Horn variables that make a program safe. Now we describe a constraint generation algorithm that produces a CHC that can be discharged by our solver to find such K-assignment.

First, we define the syntax of constraints c as described in Fig. 8 instantiating atoms as formulas φ. Next, we define constraint generation as a *bidirectional* typechecking procedure implemented by three judgments: *synthesis* $\Gamma \vdash e \Rightarrow \tau \rightsquigarrow c$, *checking* $\Gamma \vdash e \Leftarrow \tau \rightsquigarrow c$, and *subtyping* $\Gamma \vdash s <: t \rightsquigarrow c$. Like the declarative system, most rules are syntax directed and accumulate subconstraints by conjoining them. For example, function application mirrors **T-App** producing a constraint for each subexpression and conjoining them:

$$\frac{\Gamma \vdash e \Rightarrow x{:}\tau_1 \to \tau_2 \rightsquigarrow c_1 \qquad \Gamma \vdash y \Leftarrow \tau_1 \rightsquigarrow c_2}{\Gamma \vdash e\ y \Rightarrow \tau_2[y/x] \rightsquigarrow c_1 \wedge c_2}\ \textsc{Syn-App}$$

The interesting case is again subtyping on base refinements, which produces a constraint requiring that all values of base type b satisfying r_1 also satisfy r_2:

$$\frac{}{\Gamma \vdash \{\, v{:}b \mid r_1 \,\} <: \{\, v{:}b \mid r_2 \,\} \rightsquigarrow\ \forall v{:}b.\, r_1 \to r_2}\ \textsc{S-Base}$$

Soundness The constraint generated by our procedure must imply the safety of the program. We formalize this guarantee by extending the interpretation of refinements to constraints as follows:

$$[\![\top]\!]^K_\gamma \doteq \top \qquad [\![\forall x{:}b.\, c']\!]^K_\gamma \doteq \forall v \in [\![b]\!].\, [\![c']\!]^K_{\gamma[x \mapsto v]}$$
$$[\![r_1 \to r_2]\!]^K_\gamma \doteq [\![r_1]\!]^K_\gamma \to [\![r_2]\!]^K_\gamma \qquad [\![c_1 \wedge c_2]\!]^K_\gamma \doteq [\![c_1]\!]^K_\gamma \wedge [\![c_2]\!]^K_\gamma$$

Then we prove two theorems. The first connects constraint generation to declarative typing.

Theorem 5.3 (Constraint Generation). *If* $\cdot \vdash e \Leftarrow \tau \rightsquigarrow c$ *and* $[\![c]\!]^K$ *is satisfiable, then* $\cdot \vdash_K e : \tau$.

Second, composing constraint generation soundness with declarative type soundness (Theorem 5.2) yields our end-to-end guarantee: if a program passes constraint generation and the resulting constraints are satisfiable, then the program evaluates to a value in its type's interpretation:

Theorem 5.4 (Verifier Soundness). *If* $\cdot \vdash e \Leftarrow \tau \rightsquigarrow c$ *and* $[\![c]\!]^K$ *then* $\exists v$ *s.t.* $e \Downarrow v$ *and* $v \in [\![\tau]\!]^K$.

6 Implementation and Evaluation

We implemented Flex in about 2700 lines of Lean, and extended Flux with a backend that emits constraints as Lean propositions in about 1800 lines of Rust. Next, we describe experiments, using a workstation running Ubuntu 24.04.4 LTS with an AMD Ryzen 7 8845HS processor (8 cores, 16 threads) and 27GB of RAM, to evaluate Flex on three questions

- **RQ1:** How *expressive* is the system compared to SMT? (§6.1)
- **RQ2:** How *efficient* is Zap over search-based tactics in Lean? (§6.2)
- **RQ3:** How *scalable*, *automated*, and *costly* is the Lean backend? (§6.3)

6.1 RQ1: Expressiveness

Through a suite of case studies, summarized in Table 1, we demonstrate how Flex expands the scope of provable specifications by providing Flux access to a foundational logic. The studies include `Sorting`: we verify the functional correctness of in-place insertion sort, quicksort, and merge sort, proving that each returns a sorted permutation of the input vector; `RingBuffer`: we verify memory safety of a ring-buffer-backed deque adapted from the Tock embedded OS kernel [64], where we prove that the implementation never reads from uninitialized memory, and that the Rust code implements a queue interface; `TickTock`: we verify various facts about modular arithmetic

Case study	Flux LoC	#CHCs	Spec LoC	Proof LoC
Sorting	316	15	96	620
RingBuffer	257	10	75	501
TickTock	15	5	—	302
HashTable	562	22	245	1361
All	1250	52	416	2784

Table 1. Case studies: lines of verified Rust (LoC), number of constraints (#CHCs), lines of manually written Lean specs (Spec LoC), and lines of proof (Proof LoC).

(a) The ring buffer (struct and impl)

```
#[refined_by(cap:int, hd:int, tl:int,
             init: Map<int, bool>)]
#[invariant(init_inv(self))]
struct RingBuffer<'a, T: Copy + 'a> {
    buf: FSlc<'a, T>[cap, init],
    head: usize[hd],
    tail: usize[tl],
}

impl<'a, T: Copy> RingBuffer<'a, T> {
  ...
  #[proven_externally]
  fn dequeue(&mut self) -> Option<T> {
    if self.head != self.tail {
      let v = self.buf.get(self.head);
      self.head = (self.head+1) %
                    self.buf.len();
      Some(v)
    } else {
      None
    }
  }
}
```

(b) Refinement-level specifications

```
fn rb_len(rb: RingBuffer) -> int {
  if rb.tl > rb.hd { rb.tl - rb.hd }
  else if rb.tl < rb.hd { rb.cap - rb.hd + rb.tl }
  else { 0 }
}

fn valid_idx(rb: RingBuffer, idx: int) -> bool {
  (idx + rb.cap - rb.hd) % rb.cap < rb_len(rb)
}

// Defined in Lean
fn init_inv(rb: RingBuffer) -> bool;
```

(c) Trusted wrapper over maybe uninitialized slices

```
#[opaque]
#[refined_by(len: int, init: Map<int,bool>)]
struct FSlc<'a,T>(&'a mut [MaybeUninit<T>]);

impl<'a, T: Copy> FSlc<'a, T> {
  #[trusted]
  fn get(self: &FSlc<T>[@n, @f],
         idx: usize{idx < n && map_get(f, idx)}) -> T;
}
```

Fig. 16. Ring buffer implementation adapted from Tock, verified to never read uninitialized memory. This requires a Lean invariant, `init_inv`, that quantifies over valid indices and is not expressible in Flux.

and bit-vectors needed to verify that the address segments calculated by the Tock embedded kernel implement process isolation [61]; `HashTable`: we verify the correctness of a hash table with chaining based collision resolution, by proving the Rust code implements a functional Lean specification.

Previously, to ensure predictable verification, Flux was restricted to quantifier- and recursion-free SMT-decidable specifications, which precluded expressing *any* of the case studies. For example, defining sortedness requires universally quantifying over the elements of an array, and the hash-table invariants quantify over bucket contents. Even when a property *is* expressible as it falls in the nominally decidable fragment — *e.g.* the bitvector facts from TickTock — the resulting formulas can make the SMT solver time out. This highlights a key benefit of Flex: the ability to fall back to interactive proof when automation falls short.

Example: `RingBuffer` Invariant Fig. 16 shows snippets of the fixed-capacity `RingBuffer` implementation of a double-ended queue (deque) adapted from the Tock codebase, and simplified for exposition. The structure maintains *head* and *tail* indices that delimit the range of slots holding

		Zap		Zap_grind			Zap_aesop		
Suite	#CHCs	hb	ms	#reduced	×hb	×ms	#reduced	×hb	×ms
`flux-medium`	78	268	24	48 (62%)	33×	23×	65 (83%)	99×	62×
`wave`	35	518	49	20 (57%)	22×	16×	29 (83%)	41×	25×
`flux-tests`	231	100	11	161 (70%)	12×	9×	217 (94%)	23×	16×
`liquid-fixpoint`	11	106	11	7 (64%)	14×	10×	10 (91%)	17×	12×
`flex-bench`	14	585	62	11 (79%)	251×	178×	11 (79%)	87×	44×
All	369	154	16	247 (67%)	18×	13×	332 (90%)	34×	22×

Table 2. RQ2: acyclic κ-variables reduction. #CHCs is the number of constraints in each suite; Zap can reduce all of them. For Zap, hb and ms are average heartbeats and wall-clock time (ms). For Zap_grind and Zap_aesop, #reduced is the number of constraints reduced, and ×hb/×ms report average cost relative to Zap.

	Zap (baseline)			Zap_grind/Zap ratio			Zap_aesop/Zap ratio		
Suite	depth	#consts	kerμs	depth	#consts	kerμs	depth	#consts	kerμs
flux-medium	87	47	5 834	1.0×	1.0×	1.1×	1.2×	1.4×	1.7×
wave	131	54	12 065	1.0×	1.0×	1.1×	1.1×	1.3×	1.4×
flux-tests	62	42	1 996	1.0×	1.0×	1.0×	1.1×	1.3×	1.4×
liquid-fixpoint	63	43	2 177	1.0×	1.0×	1.0×	1.1×	1.3×	1.5×
`flex-bench`	121	54	13 565	1.0×	1.0×	1.2×	1.0×	1.3×	1.4×
All	73	45	3 202	1.0×	1.0×	1.0×	1.1×	1.3×	1.5×

Table 3. Elimination proof-term shape (geomean per suite). Zap columns show geomean raw values; grind/aesop columns show geomean ratio vs. Zap over jointly-solved CHCs. depth = expression-tree depth; #consts = distinct constants; kerμs = kernel re-check time (μs).

valid data. As elements are pushed and popped at either end, these indices move through the fixed-size buffer and may eventually *wrap around* (necessitating careful modular arithmetic reasoning.) Reading a slot outside the valid range is undefined behavior as it may contain uninitialized data. We track exactly which slots are initialized with a map `init` and maintain an invariant saying all valid indices are initialized. We can define the length (`rb_len`) and what indices are valid (`valid_idx`) as ordinary Flux refinement functions that translate directly into Lean definitions. The invariant `init_inv` requires quantifying over valid indices, which was not expressible as a Flux refinement, but can now be, in Lean:

```
def init_inv (rb : RingBuffer) : Prop :=
  ∀ i : Int, 0 ≤ i ∧ i < rb.cap → valid_idx rb i → rb.init i
```

Annotating the dequeue method with `proven_externally` generates a Lean constraint in which this definition is available to prove two key facts: that at the call to `FSlc::get` the precondition is satisfied, and that dequeue preserves `init_inv`.

The Lean backend does not eliminate all manual work. The Fix algorithm finds solutions automatically for simple cyclic κ-variables, but complex invariants (e.g., in `merge` and `quicksort_range`) must still be supplied by hand. Sometimes adding the right qualifiers is sufficient; other times, even with the correct invariant provided, manual Lean proof is needed to show it is maintained.

6.2 RQ2: Efficiency of Zap

Zap (§ 4.2) eliminates an acyclic κ-variable by instantiating the existential with the strongest solution for κ, and constructing a proof term that justifies replacing head occurrences of κ with $\top$. We demonstrate that deterministic proof construction makes Zap faster *and* robuster than search-based tactics, and hence, crucial in practice.

Experiment design. To evaluate Zap we implemented two variants, Zap_grind and Zap_aesop, which use the same Sol procedure to compute the strongest solution, but replace the $\mathsf{Cert}^{\kappa}_{\sigma,\rho}$ proof-term construction procedure by calls to `grind` and `aesop`, respectively, each run with default settings. We ran each tactic on constraints with one or more acyclic κ-variables, to measure the cost of reduction using Zap (§ 4.2). Our benchmarks are divided into five suites: `flux-tests`: the full Flux positive regression test suite; `liquid-fixpoint`: LiquidFixpoint's regression tests, the SMT-based solver used by many refinement type systems, including Flux; `wave`: a verified WebAssembly runtime previously ported to Flux [39, 45]; `flux-medium`: medium-size benchmarks including implementations of vector-manipulating algorithms and programs from the Flux guide and tutorials; `flex-bench`: the case studies from § 6.1.

Results. Table 2 summarizes the results across a total of 369 constraints. For each suite the table reports three metrics: the number of constraints successfully reduced and, over those, the average wall-clock time (ms) and heartbeats — Lean's deterministic proxy for proof effort. Zap can successfully reduce *all* constraints with a modest running time (max 62 ms). The search-based alternatives fare considerably worse. Zap_grind reduces only 247 of the 369 constraints (67%), and on the constraints it can reduce, it consumes 18× more heartbeats and it is 13× slower on average; the performance is worst on `flex-bench` (11s per CHC), which has the largest constraints. Zap_aesop fares better in coverage, successfully reducing 90% of the constraints, but it is 22 times slower on average than Zap. Furthermore, Proof terms produced by aesop are on average 10% deeper and 50% *slower* for the kernel to check, shown in Table 3.

6.3 RQ3: Scalability and Cost

SMT solvers are a marvelous feat of engineering. Replacing them with a *foundational* constraint solver in Lean raises a couple of questions: does it *scale* to the constraints that verifiers generate in practice, and how much automation does it *trade off*? To answer these questions we gave Flex free rein, running the full solver pipeline described in §4 — including Zap and Fix — on all of our benchmarks. For Fix, we instantiate the oracle that prunes instance candidates with a combination of `grind` and `aesop`. This leaves a κ-free VC, whose remaining goals we then attempt to close with the same `grind`/`aesop` combination. We raise Lean's default heartbeat, from 200,000 to 5,000,000, so that the search-based tactics have ample room to succeed. We exclude `flex-bench` from this experiment, as its case studies were chosen precisely because they require reasoning beyond what an SMT solver can reasonably automate.

Results. Flex automatically discharges 95.7% of all constraints across the benchmark suite (Table 4). Failures concentrate on constraints with cyclic κ-variables, where the success rate drops to 82.3% due to addressable limitations in the current implementation of `fix`. The 37 failures decompose into three categories. **(i) Recursion/Heartbeat limits:** 21 failures are caused by exceeding the default recursion depth limit during predicate abstraction, and a further 4 by exceeding the heartbeat limit; both parameters are tunable, and raising them can let additional constraints pass automatically at the cost of longer running times. **(ii) Missing qualifiers:** 4 failures occur because Flux discovers some qualifiers by scraping the constraint, and we have not yet ported this inference to the Lean backend; adding the missing qualifier manually allows our tactic to close these constraints. Even when all needed qualifiers are present, predicate abstraction can still fail to find the correct invariant,

Benchmark	#CHCs	Success	#Cyclic	Success (cyclic)	LF time	Flex Time
`flux-medium`	148	89.2%	60	47 (78.3%)	4.2s	9.5m
`wave`	77	96.1%	10	7 (70.0%)	2.8s	4.0m
`flux-tests`	608	97.3%	80	67 (83.8%)	20.9s	7.4m
`liquid-fixpoint`	47	95.7%	14	14 (100.0%)	3.1s	58.0s
Total	880	95.7%	164	135 (82.3%)	31s	21.8m

Table 4. Automation coverage of the full solver pipeline (Zap followed by Fix with a `grind/aesop` oracle), run with no human input. #CHCs is the number of constraints in the suite; Success is the fraction discharged, and Success (cyclic) the fraction discharged among those constraints with a cyclic κ-variable. LF time and Flex Time are the total wall-clock time to solve the entire suite with Liquid Fixpoint (SMT) and Flex, respectively.

depending on the oracle it relies on. **(iii) Concrete Goal closure:** 8 failures have no κ-variables at all, and simply come down to `grind/aesop` being unable to close the goal automatically; adding annotations to relevant theorems can enable Flex to handle more complex goals. This failure mode can also arise when Fix finds the correct invariant, as the verification condition remaining after κ-elimination must still be discharged by `grind/aesop`.

This demonstrates that our solver together with standard tactics can handle a variety of constraints automatically. Crucially, failures are *recoverable*: a user can fall back to an interactive Lean proof rather than being blocked with only an opaque SMT failure to go on. Moreover, improvements to proof automation in Lean would automatically transfer.

The expected cost is speed: Table 4 shows that the Lean backend is roughly 2 orders of magnitude slower than the LiquidFixpoint SMT-based backend Flux currently uses. For a foundational system whose proofs are machine-checked, this is the current price to pay. We expect that incorporating known optimizations from LiquidFixpoint into Flex will shrink the performance gap.

7 Related Work

Flex relates to a vast literature on program-verification; these are the lines closest to ours.

SMT-based Verifiers Our work is inspired by program logic [24, 35] based verifiers, in particular, those that emit a *constraint* that must be then proven to verify the program [18]. Typically these constraints are (horn variable free) VCs that are discharged by SMT solvers [15, 22, 48, 50]. Prusti [2] and Verus [44] implement such Floyd-Hoare style verifiers for Rust. F* generalizes the approach to the higher order setting using Dijkstra monads [67], while Stainless [30] uses refinement types. In contrast to the above, LiquidHaskell [73], Flux [45] and Thrust [58] use CHC, which generalize VCs with existentially quantified variables that delegate the synthesis of refinements or invariants to an SMT-based CHC solver. However, instead of relying upon incomplete, SMT heuristics [49], that can be opaque and hard to use [78], Flex provides the full arsenal of interactive proof, when required, and provides foundational guarantees about the satisfiability of the CHC.

Proving or Certifying VCs Work on proof-carrying code showed how to avoid trusting SMT solvers by modifying decision procedures to emit certificates [54]. This idea was extended to model checkers [32], and refinement types [13], and several modern theorem provers emit independently checkable certificates [3, 4, 8, 19, 52]. A dual approach is to use foundational, interactive proof assistants to discharge VCs. For example, Jahob [77] delegates VCs to a variety of backends including Isabelle. Similarly the Creusot Rust verifier [16] emits VCs in the Why3 format, after which they can be discharged using either SMT or interactive proof [20]. The above techniques

apply to plain VCs: our work shows how to represent such provers as *oracles* and thus, lift such proofs to obtain certificates for CHC satisfiability.

Transpiling to/from Interactive Provers The above techniques use verification obligations as the medium of communication between implementation (*e.g.* C, Java, Rust, Haskell) and verification (*e.g.* SMT or interactive proof). An alternative is to *transpile* the implementation *into* the prover's language, and then prove properties about the transpiled code, as done by [37] and [65] which translate Scala and Haskell into Isabelle or Rocq respectively, or [5, 34] which translate Rust into Lean. In general, though, the impedance mismatch between the languages can make such translation challenging. Of course, one can also go in the opposite direction, *i.e.* to *extract* the executable implementation from source written directly in a proof assistant, as done in systems like [1, 63] for C, or [25] for Rust or [27] for Dafny, which deeply embed program logic and refinement type based verifiers for the corresponding languages inside Rocq or Lean, after which the prover's extraction mechanism can be used to get executable code. Flex is complementary to the above, in that we provide a foundational CHC solver, that can be used either to build end-to-end verifiers (§5) *or* to verify expressive constraints generated by (trusted) compiler plugins like Flux (§6). Indeed, it would be interesting to explore using Flex to build a CHC-based verifier on top of Lean's `MVCGen` monad or the Loom system.

CHC Solvers Flex uses [14] to solve acyclic variables, and predicate abstraction [21] to compute (overapproximate) solutions for cyclic ones. In contrast, Spacer [42, 72], Eldarica [36], and SeaHorn [29] solve CHCs by inferring invariants using IC3 [10] or Lazy Abstraction and Craig Interpolation [33, 51], which do not require qualifiers, but which may diverge. It would be interesting to explore ways to extend Flex with iterative refinement techniques to reduce the reliance on qualifiers.

Mechanizing Refinement Types Our mechanization of λ_{RK} is influenced by earlier mechanizations including simple refinement types [47], the work on System FR [30], Refined Featherweight Java [66] and most closely, the polymorphic refinement calculus of [9]. Unlike the prior work, we use a big-step semantics, and more importantly, mechanize refinement *inference* using Horn variables, algorithmic constraint generation and solving.

References

[1] Andrew W. Appel, Lennart Beringer, Robert Dockins, Aquinas Hobor, Xavier Leroy, and Gordon Stewart. 2014. *Program Logics for Certified Compilers.* Cambridge University Press, New York, NY, USA.

[2] 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

[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] Karthikeyan Bhargavan, Maxime Buyse, Lucas Franceschino, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. 2024. hax: Verifying Security-Critical Rust Software Using Multiple Provers. In *Verified Software. Theories, Tools and Experiments: 16th International Conference, VSTTE 2024, Prague, Czech Republic, October 14–15, 2024, Revised Selected Papers* (Prague, Czech Republic). Springer-Verlag, Berlin, Heidelberg, 96–119. doi:10.1007/978-3-031-86695-1_7

[6] 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

[7] Nikolaj S. Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. 2013. On Solving Universally Quantified Horn Clauses. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7935).* Springer, 105–125. doi:10.1007/978-3-642-38856-9_8

[8] 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

[9] Michael H. Borkowski, Niki Vazou, and Ranjit Jhala. 2024. Mechanizing Refinement Types. *Proc. ACM Program. Lang.* 8, POPL, Article 70 (Jan. 2024), 30 pages. doi:10.1145/3632912

[10] 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

[11] Mauro Bringolf, Dominik Winterer, and Zhendong Su. 2023. Finding and Understanding Incompleteness Bugs in SMT Solvers. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 43, 10 pages. doi:10.1145/3551349.3560435

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

[13] 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

[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] 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

[17] David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *J. ACM* 52, 3 (2005), 365–473. doi:10.1145/1066100.1066102

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

[19] 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

[20] 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

[21] 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

[22] 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

[23] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations. *SIGPLAN Not.* 28, 6 (June 1993), 237–247. doi:10.1145/173262.155113

[24] 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.

[25] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. *Proc. ACM Program. Lang.* 8, PLDI, Article 192 (June 2024), 25 pages. doi:10.1145/3656422

[26] Catarina Gamboa, Abigail Reese, Alcides Fonseca, and Jonathan Aldrich. 2025. Usability Barriers for Liquid Types. *Proc. ACM Program. Lang.* 9, PLDI, Article 224 (June 2025), 26 pages. doi:10.1145/3729327

[27] 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

[28] 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.

[29] 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

[30] 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

[31] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jay R. Lorch, Bryan Parno, Justine Stephenson, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California, USA) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/2815400.2815428

[32] 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.

[33] 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

[34] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6, ICFP, Article 116 (Aug. 2022), 31 pages. doi:10.1145/3547647

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

[36] 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

[37] Lars Hupel and Viktor Kuncak. 2016. Translating Scala Programs to Isabelle/HOL. In *Proceedings of the 8th International Joint Conference on Automated Reasoning - Volume 9706*. Springer-Verlag, Berlin, Heidelberg, 568–577. doi:10.1007/978-3-319-40229-1_38

[38] Ranjit Jhala and Niki Vazou. 2021. Refinement Types: A Tutorial. *Found. Trends Program. Lang.* 6, 3–4 (Oct. 2021), 159–317. doi:10.1561/2500000032

[39] 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.

[40] 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

[41] Kenneth Knowles and Cormac Flanagan. 2009. Compositional reasoning and decidable checking for dependent contract types. In *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification* (Savannah, GA, USA) *(PLPV '09)*. Association for Computing Machinery, New York, NY, USA, 27–38. doi:10.1145/1481848.1481853

[42] 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

[43] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the 30th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 438–454. doi:10.1145/3694715.3695952

[44] 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

[45] 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

[46] Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. 2021. STORM: Refinement Types for Secure Web Applications. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 441–459. https://www.usenix.org/conference/osdi21/presentation/lehmann

[47] Nico Lehmann and Éric Tanter. 2016. Formalizing Simple Refinement Types in Coq. In *2nd International Workshop on Coq for Programming Languages (CoqPL'16)*. St. Petersburg, FL, USA.

[48] 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.

[49] 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

[50] 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.

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

[52] 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.

[53] Eric Mugnier, Yuanyuan Zhou, Ranjit Jhala, and Michael Coblenz. 2025. On the Impact of Formal Verification on Software Development. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 403 (Oct. 2025), 27 pages. doi:10.1145/3763181

[54] 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.

[55] Charles Gregory Nelson. 1980. *Techniques for Program Verification.* Ph. D. Dissertation. Stanford University, Stanford, CA, USA.

[56] Tobias Nipkow and Gerwin Klein. 2014. *Concrete Semantics with Isabelle/HOL*. Springer. http://concrete-semantics.org

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

[58] 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

[59] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 2004. Dynamic Typing with Dependent Types. In *Exploring New Frontiers of Theoretical Informatics*, Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell (Eds.). Springer US, Boston, MA, 437–450.

[60] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2026. *Logical Foundations*. Software Foundations, Vol. 1. Electronic textbook. https://softwarefoundations.cis.upenn.edu/lf-current/

[61] Vivien Rindisbacher, Evan Johnson, Nico Lehmann, Tyler Potyondy, Pat Pannuto, Stefan Savage, Deian Stefan, and Ranjit Jhala. 2025. TickTock: Verified Isolation in a Production Embedded OS. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 786–801. doi:10.1145/3731569.3764856

[62] 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

[63] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 158–174. doi:10.1145/3453483.3454036

[64] 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.

[65] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is Reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018)*. Association for Computing Machinery, 152–164. doi:10.1145/3167092

[66] Ke Sun, Di Wang, Sheng Chen, Meng Wang, and Dan Hao. 2024. Formalizing, Mechanizing, and Verifying Class-Based Refinement Types. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 39:1–39:30. doi:10.4230/LIPIcs.ECOOP.2024.39

[67] 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

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

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

[70] The mathlib Community. 2020. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*. ACM, New Orleans, LA, USA. doi:10.1145/3372885.3373824

[71] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of typed scheme. *SIGPLAN Not.* 43, 1 (Jan. 2008), 395–406. doi:10.1145/1328897.1328486

[72] 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

[73] 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

[74] Dominik Winterer and Zhendong Su. 2024. Validating SMT Solvers for Correctness and Performance via Grammar-Based Enumeration. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 355 (Oct. 2024), 24 pages. doi:10.1145/3689795

[75] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the unusual effectiveness of type-aware operator mutations for testing SMT solvers. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 193 (Nov. 2020), 25 pages. doi:10.1145/3428261

[76] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 718–730. doi:10.1145/3385412.3385985

[77] 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

[78] 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

[79] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 1789–1806.

A Zap: Soundness

This appendix proves the correctness of the Zap algorithm of § 4.2, whose two results are *Soundness* (Theorem 4.2) and *Equivalence* (Theorem 4.1).

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 (§ 4.1) 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. Since guards are atoms (Fig. 8), 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. 9). 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. 12).
- 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. 17 fixes the typing judgment $\Gamma \vdash q : \varphi$ for the proof terms of Fig. 8: 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. 17. Typing of the proof-term fragment of Fig. 8.

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

Fig. 18. The Sol procedure, reproduced from Fig. 10.

These rules are not the Lean kernel itself but our model of it, restricted to this fragment and following the kernel formalization of Carneiro [12]; 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. 10) computes the solution using Sol as in Fig. 18, with the slot names $\overline{z}$ fresh for the binders of c:

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

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:*

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

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

Proof.

(a) Immediate from the guarded equation of Fig. 18: $\kappa \notin \mathrm{heads}(c^*)$ (§ 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. 18,

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

- 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 $\mathrm{Graph}(c)$ (§ 4.1), 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.

Lemma A.2 (Reduction)**.** $\mathrm{Red}(\kappa, \sigma, c)$ *(Fig. 10) 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. 10 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. 10 eliminates the acyclic variables innermost first: writing $\overline{\kappa_A} = [\kappa_1, \ldots, \kappa_n]$ in elimination order, $c_0 = c$ and $c_i = \mathrm{Red}(\kappa_i, \sigma_i, c_{i-1})$, with σ_i synthesized on the residual c_{i-1}. The partition orders the acyclic set topologically (§ 4.1): 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 4.2) for Zap. It is stated under the typing relation, shown in Fig. 17 for the fragment of Lean proof terms in Fig. 8.

$\mathsf{Cert}^{\kappa}_{\sigma,\rho}$ 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. 12) 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. 12), 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. 12) 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. 12) 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. 12 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. 10) 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 4.2, 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. 17 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 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, § 4.3). *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. 10 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 (Equivalence, § 4.3)**.** *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 § 4.4, whose two results are *Soundness* (Theorem 4.3) and *Completeness* (Theorem 4.4).

Overview Soundness is again a typing result. Completeness is the argument of Rondon et al. [62], 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. 17 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 (§ 4.4) 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. 13 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 § 4.4: 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. 13).

(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 $\forall$, $\rightarrow$, $\wedge$, 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. 13) 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, § 4.4). *If* $\mathsf{Fix}(c, Q) = (c', q)$ *then* c' *is a VC and* $\vdash q : c' \rightarrow c$, *and* q *is closed.*

Proof. By Fig. 13, $c' = \mathsf{Red}^*(\overline{\kappa_C}, c)$ 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. 17) rebinds the prefix, yielding $\exists\overline{\kappa_C}.\, c = c$, and $\rightarrow$-introduction over h gives $q : c' \rightarrow c$. The judgment holds under the empty context, so q is closed. □

Completeness assumes *(Oracle-Completeness)*, as scoped in § 4.4.

Theorem B.5 (Completeness, § 4.4). *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). □

$$x, y, v \in \mathit{Var} \qquad \kappa \in \mathit{Kvar} \qquad z \in \mathit{Int}$$

Base sort	$\mathit{Base} \ni b$	$::= \mathsf{Int} \mid \mathsf{Bool}$
Term	t_b	$::= x_b \mid z \mid \mathsf{true} \mid \mathsf{false} \mid t_{\mathsf{Int}} + t_{\mathsf{Int}} \mid \neg\, t_{\mathsf{Bool}} \mid t_{\mathsf{Bool}} \wedge t_{\mathsf{Bool}}$
Formula	φ	$::= \top \mid \bot \mid t_b =_b t_b \mid t_{\mathsf{Int}} \leq t_{\mathsf{Int}} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$ $\mid\ \varphi \rightarrow \varphi \mid \forall x{:}b.\, \varphi \mid \exists x{:}b.\, \varphi$
Refinements	r	$::= \varphi \mid \kappa(\overline{t_b})$
Type	τ	$::= \{\, v{:}b \mid r \,\} \mid x{:}\tau \rightarrow \tau$
Primitives	$\oplus$	$::= + \mid \leq \mid \neg \mid \wedge$
Expression	e	$::= v \mid x \mid \oplus(\overline{e}) \mid e\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid (e : \tau)$
Values	$\mathit{Val} \ni v$	$::= z \mid \mathsf{true} \mid \mathsf{false} \mid \lambda x.\, e$
Type env.	Γ	$::= \cdot \mid \Gamma,\, x{:}\tau$

Fig. 19. Full syntax of λ_{RK}, extending Fig. 15 with the term and formula constructs elided in the body.

C Full Rules for λ_{RK}

This appendix gives the complete definitions and rules for λ_{RK}, expanding § 5.2. We state them with named binders for readability; the mechanization is locally nameless, with ν and every quantifier a de Bruijn index and each binding rule carrying an explicit freshness side condition. The two agree up to this change of convention, so we omit the freshness bookkeeping below.

C.1 Syntax and Semantics

Fig. 19 summarizes the full syntax of λ_{RK}, which extends the simply typed λ-calculus with refinement types [9, 38], in particular, with Horn variables κ that enable refinement inference via CHC solving.

Expressions and Evaluation The grammar follows a standard call-by-value language with arithmetic and boolean expressions. Let bindings are required because the type system enforces ANF (A-Normal Form [23]) to accommodate dependent function applications [38]. We give expressions a standard, substitution-based, big-step, call-by-value operational semantics, written $e \Downarrow v$ (e evaluates to v). The full rules appear in Fig. 20, where u ranges over the boolean values $\{\mathsf{true}, \mathsf{false}\}$. Type annotations are erased at runtime—$(e : \tau)$ evaluates as e—so refinements play no part in evaluation and serve only for static checking.

Lemma C.1 (Determinism)**.** *If $e \Downarrow v_1$ and $e \Downarrow v_2$ then $v_1 = v_2$.*

Refinements We build refinement types on top of a *separate* first-order language, to eschew the circularities in the meta-theory (where types would depend on expressions, which would themselves contain types.) This first-order language is stratified into intrinsically typed *terms* (t_b) and *formulas* (φ). A refinement r is either a first-order formula (φ) over integers and booleans, or a Horn application $\kappa(\overline{t_b})$ —highlighted in gray in Fig. 19—representing existentially quantified predicates over their arguments $\overline{t_b}$ that the solver must infer.

Types A type τ is either a *refined base type* $\{\, v{:}b \mid r \,\}$ or a *dependent arrow* $x{:}\tau \rightarrow \tau$, which names its argument x so the codomain may mention it. A refined base type restricts its sort $b \in \{\mathsf{Int}, \mathsf{Bool}\}$ to the values v satisfying the refinement r (which may be an unknown Horn application).

Next, we provide a semantics for refinements by *interpreting* them as Lean propositions, which provides a foundation for declarative typing (§ C.2) and constraint generation (§ C.3).

$$\frac{}{v \Downarrow v}\ \textbf{E-Val} \qquad \frac{e \Downarrow v}{(e : \tau) \Downarrow v}\ \textbf{E-Ann} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2[v_1/x] \Downarrow v_2}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow v_2}\ \textbf{E-Let}$$

$$\frac{e_1 \Downarrow \lambda x.\, e \qquad e_2 \Downarrow v_a \qquad e[v_a/x] \Downarrow v}{e_1\ e_2 \Downarrow v}\ \textbf{E-App} \qquad \frac{e_1 \Downarrow z_1 \qquad e_2 \Downarrow z_2}{e_1 + e_2 \Downarrow z_1 + z_2}\ \textbf{E-Add}$$

$$\frac{e_1 \Downarrow z_1 \qquad e_2 \Downarrow z_2}{e_1 \le e_2 \Downarrow (z_1 \le z_2)}\ \textbf{E-Leq} \qquad \frac{e \Downarrow u}{\neg e \Downarrow \neg u}\ \textbf{E-Not} \qquad \frac{e_1 \Downarrow u_1 \qquad e_2 \Downarrow u_2}{e_1 \wedge e_2 \Downarrow u_1 \wedge u_2}\ \textbf{E-And}$$

$$\frac{e_0 \Downarrow \mathsf{true} \qquad e_1 \Downarrow v}{\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v}\ \textbf{E-IfT} \qquad \frac{e_0 \Downarrow \mathsf{false} \qquad e_2 \Downarrow v}{\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v}\ \textbf{E-IfF}$$

Fig. 20. Big-step operational semantics of λ_{RK}.

Interpreting Formulas We interpret each base type b as a Lean type, writing $[\![b]\!]$, where $[\![\mathsf{Int}]\!]$ denotes the Lean integers *Int* and $[\![\mathsf{Bool}]\!]$ denotes the Lean booleans *Bool*. A term t_b denotes an element of $[\![b]\!]$ under a closing substitution $\gamma : \mathit{Var} \to \mathit{Val}$ that maps its free variables to values, written $[\![t_b]\!]_\gamma$. A formula denotes a Lean proposition over values and we write $[\![\varphi]\!]_\gamma$ for its interpretation as a Lean *Prop*. The interpretation is the natural one that maps constructs to Lean counterparts:

$$\begin{aligned}
[\![x]\!]_\gamma &\doteq \gamma(x) & [\![t_1 =_b t_2]\!]_\gamma &\doteq [\![t_1]\!]_\gamma = [\![t_2]\!]_\gamma \\
[\![z]\!]_\gamma &\doteq z & [\![t_1 \le t_2]\!]_\gamma &\doteq [\![t_1]\!]_\gamma \le [\![t_2]\!]_\gamma \\
[\![\mathsf{true}]\!]_\gamma,\ [\![\mathsf{false}]\!]_\gamma &\doteq \mathsf{true},\ \mathsf{false} & [\![\neg\varphi]\!]_\gamma &\doteq \neg\, [\![\varphi]\!]_\gamma \\
[\![t_1 + t_2]\!]_\gamma &\doteq [\![t_1]\!]_\gamma + [\![t_2]\!]_\gamma & [\![\varphi_1 \wedge \varphi_2]\!]_\gamma &\doteq [\![\varphi_1]\!]_\gamma \wedge [\![\varphi_2]\!]_\gamma \\
[\![\neg\, t]\!]_\gamma &\doteq \neg\, [\![t]\!]_\gamma & [\![\varphi_1 \vee \varphi_2]\!]_\gamma &\doteq [\![\varphi_1]\!]_\gamma \vee [\![\varphi_2]\!]_\gamma \\
[\![t_1 \wedge t_2]\!]_\gamma &\doteq [\![t_1]\!]_\gamma \wedge [\![t_2]\!]_\gamma & [\![\varphi_1 \to \varphi_2]\!]_\gamma &\doteq [\![\varphi_1]\!]_\gamma \to [\![\varphi_2]\!]_\gamma \\
[\![\top]\!]_\gamma &\doteq \top & [\![\forall x{:}b.\, \varphi]\!]_\gamma &\doteq \forall v \in [\![b]\!].\, [\![\varphi]\!]_{\gamma[x \mapsto v]} \\
[\![\bot]\!]_\gamma &\doteq \bot & [\![\exists x{:}b.\, \varphi]\!]_\gamma &\doteq \exists v \in [\![b]\!].\, [\![\varphi]\!]_{\gamma[x \mapsto v]}
\end{aligned}$$

Formulas are κ-free, so their interpretation does not consult the K-assignment; only refinements do.

Interpreting Refinements To give meaning to a refinement r we must also interpret Horn applications $\kappa(\overline{t_b})$. We do this by quantifying at the meta level over a *K-assignment* which fixes the meaning of each κ as a Lean predicate. Crucially, this K will itself map the *syntactic* horn variables to existentially bound Lean predicates that that solver will then instantiate. Concretely, a K-assignment has the following Lean type $\mathit{Kvar} \to \mathit{List}\,(\Sigma\, b : \mathit{Base}, [\![b]\!]) \to \mathit{Prop}$. Given a K-assignment, a Horn application is interpreted by looking up the predicate K assigns to κ and applying it to the interpreted arguments:

$$[\![\kappa(\overline{t_b})]\!]^K_\gamma \doteq K\ \kappa\ \overline{(b, [\![t_b]\!]_\gamma)}$$

Interpreting Types Finally, we interpret types as predicates on values. We write $v \in [\![\tau]\!]^K_\gamma$ to mean that the value v inhabits the interpretation of τ. A refined base type denotes the base values satisfying the (interpretation) of its refinement, and a dependent arrow denotes the lambdas that

send every argument in the domain to a result in the codomain:

$$v \in [\![\{\, \nu{:}b \mid r \,\}]\!]^K_\gamma \doteq v \in [\![b]\!] \;\wedge\; [\![r]\!]^K_{\gamma[\,\nu \mapsto v]}$$
$$v \in [\![x{:}\tau_1 \to \tau_2]\!]^K_\gamma \doteq v = \lambda x.\, e \;\wedge\; \forall v_a.\, v_a \in [\![\tau_1]\!]^K_\gamma \Rightarrow \exists v_r.\, e[v_a/x] \Downarrow v_r \;\wedge\; v_r \in [\![\tau_2]\!]^K_{\gamma[x \mapsto v_a]}$$

C.2 Declarative Typing

We define a declarative typing judgment $\Gamma \vdash_K e : \tau$ which is mostly unremarkable except for being parameterized by a K-assignment. The full rules are given in Fig. 21, with subtyping in Fig. 22. We highlight a couple of aspects of the system:

Typing Variables with Selfification When typing variables we strengthen its refinement by giving its *selfified* type [59], which crucially enables path-sensitive "occurrence" typing [71].

$$\frac{(x{:}\tau) \in \Gamma}{\Gamma \vdash_K x : \mathrm{self}(x, \tau)}\ \textsc{T-Var} \qquad \begin{aligned} \mathrm{self}(x, \{\, \nu{:}b \mid r \,\}) &\doteq \{\, \nu{:}b \mid \nu = x \,\} \\ \mathrm{self}(x, y : s \to t) &\doteq y : s \to t \end{aligned}$$

The mechanization synthesizes the bare singleton $\{\, \nu{:}b \mid \nu = x \,\}$ rather than $\{\, \nu{:}b \mid r \wedge \nu = x \,\}$, dropping r so that synthesized types stay κ-free; no information is lost, since r remains recorded against x in the typing context and is recovered from there for subtyping.

Typing Applications The typing judgment enforces ANF so that a function is always applied to a *variable*, which can be substituted into the dependent codomain without placing an arbitrary expression inside a refinement. (The alternative is to extend the language with *existential types* [41], which exchanges the restriction for more complex subtyping and metatheory.)

$$\frac{\Gamma \vdash_K e : x{:}\tau_1 \to \tau_2 \qquad \Gamma \vdash_K y : \tau_1}{\Gamma \vdash_K e\, y : \tau_2\,[y/x]}\ \textsc{T-App}$$

Subtyping Most rules in the declarative system are syntax directed; subtyping is the only place where refinements are actually compared, so it is the place where essentially all of the interesting checking happens. A subsumption rule lets an expression of type τ_1 be used at any supertype τ_2, deferring to a subtyping judgment $\Gamma \vdash_K \tau_1 <: \tau_2$. For arrows it is the standard rule, contravariant in the domain and covariant in the codomain; for two refined base types, subtyping holds when the first refinement implies the second under the assumptions in the context:

$$\frac{\forall \gamma, (\!|\Gamma|\!)^K_\gamma \to [\![r_1]\!]^K_\gamma \to [\![r_2]\!]^K_\gamma}{\Gamma \vdash_K \{\, \nu{:}b \mid r_1 \,\} <: \{\, \nu{:}b \mid r_2 \,\}}\ \textsc{S-Base} \qquad \frac{\Gamma \vdash_K \tau_1' <: \tau_1 \qquad \Gamma, x{:}\tau_1' \vdash_K \tau_2 <: \tau_2'}{\Gamma \vdash_K x{:}\tau_1 \to \tau_2 <: x{:}\tau_1' \to \tau_2'}\ \textsc{S-Fun}$$

Here $(\!|\Gamma|\!)^K_\gamma$ *extracts* the assumptions from Γ by conjoining the base refinements

$$\begin{aligned} (\!|\cdot|\!)^K_\gamma &\doteq \top \\ (\!|\Gamma, x{:}\{\, \nu{:}b \mid r \,\}|\!)^K_\gamma &\doteq (\!|\Gamma|\!)^K_\gamma \wedge [\![r[x/\nu]]\!]^K_\gamma \\ (\!|\Gamma, x{:}\tau_1 \to \tau_2|\!)^K_\gamma &\doteq (\!|\Gamma|\!)^K_\gamma \end{aligned}$$

Extraction also gives us *entailment* of a refinement implication in a context, stated semantically so that it applies uniformly whether the refinements are formulas or Horn applications:

$$\Gamma \models_K \forall \nu{:}b.\, r_1 \to r_2 \;\doteq\; \forall \gamma.\, (\!|\Gamma|\!)^K_\gamma \to \forall v \in [\![b]\!].\, [\![r_1]\!]^K_{\gamma[\,\nu \mapsto v]} \to [\![r_2]\!]^K_{\gamma[\,\nu \mapsto v]}$$

This is exactly the premise of **S-Base** in Fig. 22.

Soundness We prove the declarative system sound: a well-typed program never gets stuck, evaluating to a value that inhabits the interpretation of its type. The proof factors through two lemmas: subtyping is semantic inclusion of type denotations, and a fundamental lemma that evaluates every well-typed term into its denotation under a closing substitution.

$$\frac{(x{:}\tau) \in \Gamma}{\Gamma \vdash_K x : \mathrm{self}(x, \tau)}\ \textbf{T-Var} \qquad \frac{}{\Gamma \vdash_K z : \{\ v{:}\mathsf{Int} \mid v = z\ \}}\ \textbf{T-Int}$$

$$\frac{u \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma \vdash_K u : \{\ v{:}\mathsf{Bool} \mid v = u\ \}}\ \textbf{T-Bool} \qquad \frac{x \notin \mathrm{dom}(\Gamma) \qquad \Gamma,\ x{:}\tau_1 \vdash_K e : \tau_2}{\Gamma \vdash_K \lambda x.\, e : x{:}\tau_1 \to \tau_2}\ \textbf{T-Lam}$$

$$\frac{\Gamma \vdash_K e : x{:}\tau_1 \to \tau_2 \qquad \Gamma \vdash_K y : \tau_1}{\Gamma \vdash_K e\ y : \tau_2\ [y/x]}\ \textbf{T-App}$$

$$\frac{\Gamma \vdash_K e_1 : s \qquad x \notin \mathrm{dom}(\Gamma) \cup \mathrm{fv}(\tau) \qquad \Gamma,\ x{:}s \vdash_K e_2 : \tau}{\Gamma \vdash_K \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}\ \textbf{T-Let} \qquad \frac{\Gamma \vdash_K e : \tau}{\Gamma \vdash_K (e : \tau) : \tau}\ \textbf{T-Ann}$$

$$\frac{\Gamma \vdash_K e : s \qquad \Gamma \vdash_K s <: \tau}{\Gamma \vdash_K e : \tau}\ \textbf{T-Sub} \qquad \frac{(x{:}\{\ v{:}\mathsf{Int} \mid r_1\ \}) \in \Gamma \qquad (y{:}\{\ v{:}\mathsf{Int} \mid r_2\ \}) \in \Gamma}{\Gamma \vdash_K x + y : \{\ v{:}\mathsf{Int} \mid v = x + y\ \}}\ \textbf{T-Add}$$

$$\frac{(x{:}\{\ v{:}\mathsf{Int} \mid r_1\ \}) \in \Gamma \qquad (y{:}\{\ v{:}\mathsf{Int} \mid r_2\ \}) \in \Gamma}{\Gamma \vdash_K x \le y : \{\ v{:}\mathsf{Bool} \mid (v = \mathsf{true} \to x \le y) \wedge (x \le y \to v = \mathsf{true})\ \}}\ \textbf{T-Leq}$$

$$\frac{(x{:}\{\ v{:}\mathsf{Bool} \mid r\ \}) \in \Gamma}{\Gamma \vdash_K \neg x : \{\ v{:}\mathsf{Bool} \mid v = \neg x\ \}}\ \textbf{T-Not} \qquad \frac{(x{:}\{\ v{:}\mathsf{Bool} \mid r_1\ \}) \in \Gamma \qquad (y{:}\{\ v{:}\mathsf{Bool} \mid r_2\ \}) \in \Gamma}{\Gamma \vdash_K x \wedge y : \{\ v{:}\mathsf{Bool} \mid v = x \wedge y\ \}}\ \textbf{T-And}$$

$$\frac{\begin{array}{c}(x{:}\{\ v{:}\mathsf{Bool} \mid r\ \}) \in \Gamma \qquad y \notin \mathrm{dom}(\Gamma) \\ \Gamma,\ y{:}\{\ v{:}\mathsf{Bool} \mid x = \mathsf{true}\ \} \vdash_K e_1 : \tau \\ \Gamma,\ y{:}\{\ v{:}\mathsf{Bool} \mid x = \mathsf{false}\ \} \vdash_K e_2 : \tau\end{array}}{\Gamma \vdash_K \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau}\ \textbf{T-If}$$

Fig. 21. Declarative typing for λ_{RK}. The primitive rules (**T-Add**, **T-Leq**, **T-Not**, **T-And**) and **T-If** require variable arguments, enforcing ANF; **T-If** records the path condition $x = \mathsf{true}$ (resp. $x = \mathsf{false}$) in each branch via a fresh binding y whose refinement mentions x but not v, leaving the scrutinee's own refinement r in scope.

$$\frac{\Gamma \models_K \forall v{:}b.\ r_1 \to r_2}{\Gamma \vdash_K \{\ v{:}b \mid r_1\ \} <: \{\ v{:}b \mid r_2\ \}}\ \textbf{S-Base} \qquad \frac{\Gamma \vdash_K \tau_1' <: \tau_1 \qquad \Gamma, x{:}\tau_1' \vdash_K \tau_2 <: \tau_2'}{\Gamma \vdash_K x{:}\tau_1 \to \tau_2 <: x{:}\tau_1' \to \tau_2'}\ \textbf{S-Fun}$$

Fig. 22. Declarative subtyping for λ_{RK}.

Lemma C.2 (Subtyping is Semantic Inclusion)**.** *If* $\Gamma \vdash_K s <: t$*, then for every* γ *such that* $(\!|\Gamma|\!)^K_\gamma$ *and every value* v*,* $v \in [\![s]\!]^K_\gamma$ *implies* $v \in [\![t]\!]^K_\gamma$*.*

Lemma C.3 (Fundamental Lemma)**.** *Suppose* $\Gamma \vdash_K e : \tau$ *and* γ *maps each variable bound in* Γ *to a closed value with* $\gamma(x) \in [\![\tau_x]\!]^K_\gamma$ *for every* $(x{:}\tau_x) \in \Gamma$*. Then there exists a value* v *such that* $e\,\gamma \Downarrow v$ *and* $v \in [\![\tau]\!]^K_\gamma$*, where* $e\,\gamma$ *applies* γ *as a substitution to* e*.*

Instantiating the fundamental lemma with the empty context and empty substitution yields type soundness, restating Theorem 5.2.

Theorem C.4 (Type soundness). *If* $\cdot \vdash_K e : \tau$, *then there exists a value* v *such that* $e \Downarrow v$ *and* $v \in [\![\tau]\!]^K_\emptyset$.

C.3 Algorithmic Constraint Generation

The declarative system assumes a K-assignment with the solutions for Horn variables that make a program safe. Now we describe a constraint generation algorithm that produces a CHC that can be discharged by our solver to find such K-assignment.

First, we define the syntax of constraints c as described in Fig. 8 instantiating atoms as formulas φ. Next, we define constraint generation as a *bidirectional* typechecking procedure implemented by three judgments: *synthesis* $\Gamma \vdash e \Rightarrow \tau \rightsquigarrow c$, *checking* $\Gamma \vdash e \Leftarrow \tau \rightsquigarrow c$, and *subtyping* $\Gamma \vdash s <: t \rightsquigarrow c$. The full rules are given in Fig. 23 to 25. Like the declarative system, most rules are syntax directed and accumulate subconstraints by conjoining them. For example, function application mirrors **T-App** producing a constraint for each subexpression and conjoining them:

$$\frac{\Gamma \vdash e \Rightarrow x{:}\tau_1 \to \tau_2 \rightsquigarrow c_1 \qquad \Gamma \vdash y \Leftarrow \tau_1 \rightsquigarrow c_2}{\Gamma \vdash e\ y \Rightarrow \tau_2[y/x] \rightsquigarrow c_1 \wedge c_2}\ \textsc{Syn-App}$$

The interesting case is again subtyping on base refinements, which produces a constraint requiring that all values of base type b satisfying r_1 also satisfy r_2:

$$\frac{}{\Gamma \vdash \{\, v{:}b \mid r_1 \,\} <: \{\, v{:}b \mid r_2 \,\} \rightsquigarrow\ \forall v{:}b.\, r_1 \to r_2}\ \textsc{Sub-Base}$$

Implication Binders Rules that move under a binder wrap the subconstraint of the body in an *implication binder* $\lceil x{:}\tau \rceil$, which quantifies over the values of a base-typed binding and assumes its refinement, and is the identity for function-typed bindings:

$$\lceil x{:}\{\, v{:}b \mid r \,\} \rceil\ c \doteq \forall x{:}b.\, r[x/v] \to c$$
$$\lceil x{:}(y{:}\tau_1 \to \tau_2) \rceil\ c \doteq c$$

Soundness The constraint generated by our procedure must imply the safety of the program. We formalize this guarantee by extending the interpretation of refinements to constraints as follows:

$$[\![\top]\!]^K_\gamma \doteq \top$$
$$[\![r_1 \to r_2]\!]^K_\gamma \doteq [\![r_1]\!]^K_\gamma \to [\![r_2]\!]^K_\gamma$$
$$[\![\forall x{:}b.\, c']\!]^K_\gamma \doteq \forall v \in [\![b]\!].\, [\![c']\!]^K_{\gamma[x \mapsto v]}$$
$$[\![c_1 \wedge c_2]\!]^K_\gamma \doteq [\![c_1]\!]^K_\gamma \wedge [\![c_2]\!]^K_\gamma$$

and we write $\Gamma \models_K c$ for $\forall \gamma.\ (\!|\Gamma|\!)^K_\gamma \to [\![c]\!]^K_\gamma$. The generation judgments are sound for the declarative system in arbitrary contexts:

Lemma C.5 (Subtyping Generation). *If* $\Gamma \vdash s <: t \rightsquigarrow c$ *and* $\Gamma \models_K c$, *then* $\Gamma \vdash_K s <: t$.

Lemma C.6 (Generation Soundness). *If* $\Gamma \vdash e \Rightarrow \tau \rightsquigarrow c$ *and* $\Gamma \models_K c$, *then* $\Gamma \vdash_K e : \tau$; *and likewise if* $\Gamma \vdash e \Leftarrow \tau \rightsquigarrow c$ *and* $\Gamma \models_K c$, *then* $\Gamma \vdash_K e : \tau$.

Specializing Lemma C.6 to the empty context restates Theorem 5.3:

Theorem C.7 (Constraint Generation). *If* $\cdot \vdash e \Leftarrow \tau \rightsquigarrow c$ *and* $[\![c]\!]^K_\cdot$ *is satisfiable, then* $\cdot \vdash_K e : \tau$.

Second, composing constraint generation soundness with declarative type soundness (Theorem C.4) yields our end-to-end guarantee, restating Theorem 5.4: if a program passes constraint generation and the resulting constraints are satisfiable, then the program evaluates to a value in its type's interpretation:

$$\frac{(x{:}\tau) \in \Gamma}{\Gamma \vdash x \Rightarrow \mathrm{self}(x,\tau) \rightsquigarrow \top}\ \textsc{Syn-Var} \qquad \frac{}{\Gamma \vdash z \Rightarrow \{\ v{:}\mathsf{Int} \mid v = z\ \} \rightsquigarrow \top}\ \textsc{Syn-Int}$$

$$\frac{u \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma \vdash u \Rightarrow \{\ v{:}\mathsf{Bool} \mid v = u\ \} \rightsquigarrow \top}\ \textsc{Syn-Bool} \qquad \frac{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow c}{\Gamma \vdash (e : \tau) \Rightarrow \tau \rightsquigarrow c}\ \textsc{Syn-Ann}$$

$$\frac{\Gamma \vdash e \Rightarrow x{:}\tau_1 \to \tau_2 \rightsquigarrow c_1 \qquad \Gamma \vdash y \Leftarrow \tau_1 \rightsquigarrow c_2}{\Gamma \vdash e\ y \Rightarrow \tau_2[y/x] \rightsquigarrow c_1 \wedge c_2}\ \textsc{Syn-App}$$

$$\frac{(x{:}\{\ v{:}\mathsf{Int} \mid r_1\ \}) \in \Gamma \qquad (y{:}\{\ v{:}\mathsf{Int} \mid r_2\ \}) \in \Gamma}{\Gamma \vdash x + y \Rightarrow \{\ v{:}\mathsf{Int} \mid v = x + y\ \} \rightsquigarrow \top}\ \textsc{Syn-Add}$$

$$\frac{(x{:}\{\ v{:}\mathsf{Int} \mid r_1\ \}) \in \Gamma \qquad (y{:}\{\ v{:}\mathsf{Int} \mid r_2\ \}) \in \Gamma}{\Gamma \vdash x \le y \Rightarrow \{\ v{:}\mathsf{Bool} \mid (v = \mathsf{true} \to x \le y) \wedge (x \le y \to v = \mathsf{true})\ \} \rightsquigarrow \top}\ \textsc{Syn-Leq}$$

$$\frac{(x{:}\{\ v{:}\mathsf{Bool} \mid r\ \}) \in \Gamma}{\Gamma \vdash \neg x \Rightarrow \{\ v{:}\mathsf{Bool} \mid v = \neg x\ \} \rightsquigarrow \top}\ \textsc{Syn-Not}$$

$$\frac{(x{:}\{\ v{:}\mathsf{Bool} \mid r_1\ \}) \in \Gamma \qquad (y{:}\{\ v{:}\mathsf{Bool} \mid r_2\ \}) \in \Gamma}{\Gamma \vdash x \wedge y \Rightarrow \{\ v{:}\mathsf{Bool} \mid v = x \wedge y\ \} \rightsquigarrow \top}\ \textsc{Syn-And}$$

Fig. 23. Constraint synthesis for λ_{RK}.

$$\frac{x \notin \mathrm{dom}(\Gamma) \qquad \Gamma,\ x{:}\tau_1 \vdash e \Leftarrow \tau_2 \rightsquigarrow c}{\Gamma \vdash \lambda x.\, e \Leftarrow x{:}\tau_1 \to \tau_2 \rightsquigarrow \lceil x{:}\tau_1 \rceil\, c}\ \textsc{Chk-Lam}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow s \rightsquigarrow c_1 \qquad x \notin \mathrm{dom}(\Gamma) \cup \mathrm{fv}(\tau) \qquad \Gamma,\ x{:}s \vdash e_2 \Leftarrow \tau \rightsquigarrow c_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Leftarrow \tau \rightsquigarrow c_1 \wedge \lceil x{:}s \rceil\, c_2}\ \textsc{Chk-Let}$$

$$\frac{\begin{array}{c}(x{:}\{\ v{:}\mathsf{Bool} \mid r\ \}) \in \Gamma \qquad y \notin \mathrm{dom}(\Gamma) \\ \Gamma,\ y{:}\{\ v{:}\mathsf{Bool} \mid x = \mathsf{true}\ \} \vdash e_1 \Leftarrow \tau \rightsquigarrow c_1 \\ \Gamma,\ y{:}\{\ v{:}\mathsf{Bool} \mid x = \mathsf{false}\ \} \vdash e_2 \Leftarrow \tau \rightsquigarrow c_2\end{array}}{\Gamma \vdash \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Leftarrow \tau \rightsquigarrow \lceil y{:}\{\ v{:}\mathsf{Bool} \mid x = \mathsf{true}\ \} \rceil\, c_1 \wedge \lceil y{:}\{\ v{:}\mathsf{Bool} \mid x = \mathsf{false}\ \} \rceil\, c_2}\ \textsc{Chk-If}$$

$$\frac{e \text{ is not a } \lambda\text{-, let-, or if-expression} \qquad \Gamma \vdash e \Rightarrow s \rightsquigarrow c_1 \qquad \Gamma \vdash s <: \tau \rightsquigarrow c_2}{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow c_1 \wedge c_2}\ \textsc{Chk-Syn}$$

Fig. 24. Constraint checking for λ_{RK}.

Theorem C.8 (Verifier Soundness). *If* $\cdot \vdash e \Leftarrow \tau \rightsquigarrow c$ *and* $[\![c]\!]^K_{\cdot}$ *then* $\exists v$ *s.t.* $e \Downarrow v$ *and* $v \in [\![\tau]\!]^K_{\cdot}$.

$$\frac{}{\Gamma \vdash \{\, v{:}b \mid r_1 \,\} <: \{\, v{:}b \mid r_2 \,\} \rightsquigarrow\ \forall v{:}b.\, r_1 \rightarrow r_2}\ \textsc{Sub-Base}$$

$$\frac{\Gamma \vdash \tau_1' <: \tau_1 \rightsquigarrow c_1 \qquad \Gamma,\, x{:}\tau_1' \vdash \tau_2 <: \tau_2' \rightsquigarrow c_2}{\Gamma \vdash x{:}\tau_1 \rightarrow \tau_2 <: x{:}\tau_1' \rightarrow \tau_2' \rightsquigarrow c_1 \wedge \lceil x{:}\tau_1' \rceil\, c_2}\ \textsc{Sub-Fun}$$

Fig. 25. Constraint-emitting subtyping for λ_{RK}.

$$\frac{}{\langle \mathsf{skip},\, s\rangle \Downarrow s}\ \textbf{E-Skip} \qquad \frac{}{\langle x := e,\, s\rangle \Downarrow s[x \mapsto e(s)]}\ \textbf{E-Asgn}$$

$$\frac{\langle c_1,\, s_1\rangle \Downarrow s_2 \qquad \langle c_2,\, s_2\rangle \Downarrow s_3}{\langle c_1; c_2,\, s_1\rangle \Downarrow s_3}\ \textbf{E-Seq} \qquad \frac{g(s_1) \qquad \langle c_1,\, s_1\rangle \Downarrow s_2}{\langle \mathsf{if}\ g\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2,\, s_1\rangle \Downarrow s_2}\ \textbf{E-IfT}$$

$$\frac{\neg g(s_1) \qquad \langle c_2,\, s_1\rangle \Downarrow s_2}{\langle \mathsf{if}\ g\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2,\, s_1\rangle \Downarrow s_2}\ \textbf{E-IfF} \qquad \frac{\neg g(s)}{\langle \mathsf{while}\ g\ \mathsf{do}\ c\ ,\, s\rangle \Downarrow s}\ \textbf{E-WhF}$$

$$\frac{g(s_1) \qquad \langle c,\, s_1\rangle \Downarrow s_2 \qquad \langle \mathsf{while}\ g\ \mathsf{do}\ c\ ,\, s_2\rangle \Downarrow s_3}{\langle \mathsf{while}\ g\ \mathsf{do}\ c\ ,\, s_1\rangle \Downarrow s_3}\ \textbf{E-WhT}$$

Fig. 26. Big-step operational semantics of Imp.

D Full Rules for Imp

Commands, Assertions and Triples We represent Imp programs with a *shallow* embedding where states s are functions $Var \rightarrow \mathbb{Z}$, expressions e and guards g are state-indexed, and assertions P, Q are predicates $State \rightarrow Prop$. A (Floyd-Hoare) *triple* comprising a pre-condition P, command c and post-condition Q is *valid*, written $\models \{P\}\ c\ \{Q\}$, if every terminating run of c from a P-state ends in a Q-state.

Operational Semantics The state update $s[x \mapsto v]$ rebinds x to v:

$$s[x \mapsto v] \ \doteq\ \lambda y.\mathsf{if}\ y = x\ \mathsf{then}\ v\ \mathsf{else}\ s(y)$$

Fig. 26 gives the standard big-step evaluation judgment $\langle c,\, s_1\rangle \Downarrow s_2$ (command c takes state s_1 to state s_2), which makes triple validity formal:

$$\models \{P\}\ c\ \{Q\} \ \doteq\ \forall s_1\, s_2.\ \langle c,\, s_1\rangle \Downarrow s_2 \rightarrow P(s_1) \rightarrow Q(s_2)$$

Floyd-Hoare Proof Rules Fig. 27 lists the standard proof rules [24, 35], stated directly as lemmas about valid triples.

Lemma D.1 (Hoare Rules)**.** *Every rule in Fig. 27 is valid: if the premises hold, so does the conclusion.*

Constraint Generator Let $V \doteq \{x_1, \ldots, x_n\}$ be a set of n ordered variables that occur in commands. The generator $\mathrm{VC}(P, c, Q)$ takes as input a triple P, c, Q (where c uses variables from V) and outputs a CHC. The implementation is the textbook *weakest precondition*-based VC-generation method [18], except in two places. First, we do not require explicitly provided invariants for loops: when the generator hits a while it introduces a Horn *variable* κ for the unknown invariant — a $|V|$-ary

$$\frac{}{\models \{P\}\ \mathsf{skip}\ \{P\}}\ \textbf{H-Skip} \qquad \frac{}{\models \{\lambda s.Q(s[x \mapsto e(s)])\}\ x := e\ \{Q\}}\ \textbf{H-Asgn}$$

$$\frac{\models \{P\}\ c_1\ \{R\} \qquad \models \{R\}\ c_2\ \{Q\}}{\models \{P\}\ c_1; c_2\ \{Q\}}\ \textbf{H-Seq}$$

$$\frac{\models \{\lambda s.P(s) \wedge g(s)\}\ c_1\ \{Q\} \qquad \models \{\lambda s.P(s) \wedge \neg g(s)\}\ c_2\ \{Q\}}{\models \{P\}\ \mathsf{if}\ g\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \{Q\}}\ \textbf{H-If}$$

$$\frac{\forall s.\ P(s) \to I(s) \qquad \models \{\lambda s.I(s) \wedge g(s)\}\ c\ \{I\} \qquad \forall s.\ I(s) \wedge \neg g(s) \to Q(s)}{\models \{P\}\ \mathsf{while}\ g\ \mathsf{do}\ c\ \ \{Q\}}\ \textbf{H-While}$$

$$\frac{\models \{P'\}\ c\ \{Q\} \qquad \forall s.\ P(s) \to P'(s)}{\models \{P\}\ c\ \{Q\}}\ \textbf{H-ConsPre}$$

$$\frac{\models \{P\}\ c\ \{Q'\} \qquad \forall s.\ Q'(s) \to Q(s)}{\models \{P\}\ c\ \{Q\}}\ \textbf{H-ConsPost}$$

Fig. 27. Derived Floyd-Hoare proof rules for Imp.

predicate — and *constrains* it to satisfy the usual initial, body and exit obligations:

$$\begin{aligned}
\mathsf{VC}(P, \mathsf{while}\ g\ \mathsf{do}\ c\ , Q) \quad \doteq \quad & \exists\, \kappa : \mathit{Int}^{|V|} \to \mathit{Prop}. \\
& \quad \forall s.\ P(s) \to [\![\kappa]\!](s) && \text{(initial)} \\
& \quad \wedge\ \mathsf{VC}(\lambda s.[\![\kappa]\!](s) \wedge g(s), c, [\![\kappa]\!]) && \text{(body)} \\
& \quad \wedge\ \forall s.\ [\![\kappa]\!](s) \to \neg g(s) \to Q(s) && \text{(exit)} \\
\text{where} \quad [\![\kappa]\!] \quad \doteq \quad & \lambda s.\kappa(s(x_1), \ldots, s(x_n))
\end{aligned}$$

The full definition appears in Fig. 28. The generator is in continuation-passing weakest-precondition style: $\mathsf{WP}_V(c,\ Q,\ k)$ computes the weakest precondition w of c with respect to Q, emits proof obligations along the way, and hands w to the continuation k; the top level $\mathsf{VC}_V(P,\ c,\ Q)$ instantiates k with the requirement that P imply the computed precondition. Only loops introduce a Horn variable; sequencing computes intermediate assertions by backward substitution. The scope V makes the invariants' arity explicit and grows through sequencing: the second command sees the variables $\mathsf{asg}(c_1)$ assigned by the first. Invariants additionally receive the values of a fixed list $C = t_1, \ldots, t_m$ of integer-valued specification terms harvested from P and Q.

The above constraints are not in the syntax from Fig. 8 as we are quantifying over states s and not base-sorted values. Fortunately, Lean's `simp` tactic suffices to reduce them to our grammar.

Soundness We prove in Lean that whenever the CHC returned by $\mathsf{VC}_V(P,\ c,\ Q)$ is satisfiable, that the corresponding triple is valid. The proof factors through a generic lemma about the generator: any satisfied $\mathsf{WP}_V(c,\ Q,\ k)$ yields a genuine precondition.

Lemma D.2 (Generator Soundness)**.** *If* $\mathsf{WP}_V(c,\ Q,\ k)$*, then there is an assertion* w *such that* $\models \{w\}\ c\ \{Q\}$ *and* $k(w)$*.*

Theorem D.3 (VC-Generation soundness)**.** *For any scope* V *and spec terms* C*, if* $\mathsf{VC}_V(P,\ c,\ Q)$ *then* $\models \{P\}\ c\ \{Q\}$*.*

$$
\begin{aligned}
\mathrm{WP}_V(\mathsf{skip},\, Q,\, k) &\doteq k(Q)\\
\mathrm{WP}_V(x := e,\, Q,\, k) &\doteq k(\lambda s.Q(s[x \mapsto e(s)]))\\
\mathrm{WP}_V(c_1; c_2,\, Q,\, k) &\doteq \mathrm{WP}_{V'}(c_2,\, Q,\, \lambda w.\mathrm{WP}_V(c_1,\, w,\, k))\\
\mathrm{WP}_V(\mathsf{if}\ g\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2,\, Q,\, k) &\doteq \mathrm{WP}_V(c_1,\, Q,\, \lambda w_1.\mathrm{WP}_V(c_2,\, Q,\, \lambda w_2.k(w_g)))\\
\mathrm{WP}_V(\mathsf{while}\ g\ \mathsf{do}\ c\ ,\, Q,\, k) &\doteq\ \exists \kappa : \mathit{Int}^{m+|V|} \to \mathit{Prop}.\\
&\qquad \mathrm{WP}_{V''}(c,\, [\![\kappa]\!]_V,\, \lambda w.\forall s.\, [\![\kappa]\!]_V(s) \wedge g(s) \to w(s)) \quad \text{(preserve)}\\
&\quad \wedge \forall s.\, [\![\kappa]\!]_V(s) \wedge \neg g(s) \to Q(s) \quad \text{(exit)}\\
&\quad \wedge k([\![\kappa]\!]_V) \quad \text{(continue)}
\end{aligned}
$$

$$\mathrm{VC}_V(P,\, c,\, Q) \doteq \mathrm{WP}_V(c,\, Q,\, \lambda w.\forall s.\, P(s) \to w(s))$$

where $V' \doteq V \cup \mathrm{asg}(c_1)$, $\quad V'' \doteq V \cup \mathrm{asg}(c)$, $\quad w_g \doteq \lambda s.(g(s) \to w_1(s)) \wedge (\neg g(s) \to w_2(s))$

$[\![\kappa]\!]_V \ \doteq\ \lambda s.\kappa(t_1(s), \ldots, t_m(s),\, s(x_1), \ldots, s(x_n))$ for $V = x_1, \ldots, x_n$ and $C = t_1, \ldots, t_m$

$\mathrm{asg}(\mathsf{skip}) \doteq \emptyset \qquad \mathrm{asg}(x := e) \doteq \{x\} \qquad \mathrm{asg}(\mathsf{while}\ g\ \mathsf{do}\ c\ ) \doteq \mathrm{asg}(c)$

$\mathrm{asg}(c_1; c_2) \doteq \mathrm{asg}(c_1) \cup \mathrm{asg}(c_2) \qquad \mathrm{asg}(\mathsf{if}\ g\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \doteq \mathrm{asg}(c_1) \cup \mathrm{asg}(c_2)$

Fig. 28. The full constraint generator for Imp, in continuation-passing weakest-precondition style; the spec-term list C is fixed throughout. $\mathrm{asg}(c)$ collects the assignment targets of c in order of first appearance, and $V \cup V'$ appends the new variables of V' to V in that order.