Constrained Horn clauses

Finding the relations that make a proof go through

CMPUT 664 · Formal Verification · November 16, 2026

We know how to check an invariant

For a loop proof, an invariant must:

1. Hold when the loop is first reached.
2. Be preserved by every iteration.
3. Establish the required property when the loop exits.

In Dafny and Lean, we supplied the invariant and justified these obligations.

Can a solver find the invariant as well?

Leave the relation unknown

Find a condition $R(n, x, y)$ that holds **whenever the loop guard is tested**.

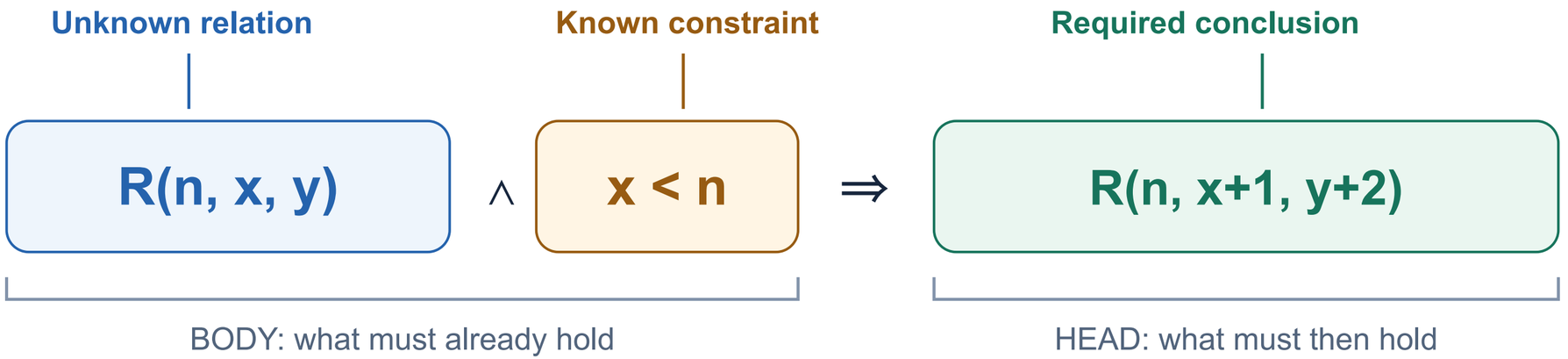
```
1 x, y = 0, 0              # (1) Initialize; n is a nonnegative integer
2 while x < n:             # unknown invariant R(n, x, y)
3     x += 1               # (2) One iteration
4     y += 2
5 assert y == 2 * n        # (3) Check the result
```

Code	Requirement on R
(1) Initialize: `x, y = 0, 0`	For every $n \geq 0$, include $(n, 0, 0)$ in R.
(2) Iterate: `x += 1; y += 2`	From (n, x, y) in R with $x < n$, include $(n, x + 1, y + 2)$ in R.
(3) Assert: `y == 2*n`	Any state in R that exits ($x \geq n$) must satisfy $y = 2n$.

The verifier generates clauses from the code and assertion. The solver searches for an R satisfying them.

What is a constrained Horn clause?

A CHC is an implication with a restricted **body** and **head**:



- **Horn:** unnegated relation premises imply one relation fact, or false.
- **Constrained:** the body combines relation premises with known theory conditions, such as $x < n$.

The rule must hold for **all values of its variables**.

CHCs are still solver queries

The question is **what we ask the solver to find**.

Verification task	What is supplied	What the solver searches for
Check an invariant	A particular invariant formula	A counterexample to a proof obligation
Bounded model checking	A program and an execution bound	A failing execution within that bound
CHC solving	Rules with unknown relations	Interpretations of those relations satisfying every rule

An SMT script can express the CHCs. A **Horn-solving engine** specializes in finding predicates within this recursive rule structure.

Why the Horn structure helps

For this comparison, p, q, r are Boolean facts, and p is required to be true.

Horn rule	**Non-Horn rule**
$p \implies q$	$p \implies q \vee r$
q must be true	At least one of q, r must be true, but neither is individually forced
Required consequences can be propagated	Choices must be made consistently with the other clauses

For the Horn rule, the smallest model is $\{p, q\}$.

For the non-Horn rule, both $\{p, q\}$ and $\{p, r\}$ are models, but their common facts $\{p\}$ are not a model.

Horn start/step rules have a least set of consequences. This supports fixed-point computation.

The computational payoff

For **finite propositional formulas**, the restriction changes the complexity:

Problem	Computation
Horn-SAT	Forward propagation suffices; solvable in **linear time**
General SAT	**NP-complete**; disjunctive choices can interact across clauses

For CHCs, the relations may range over infinitely many integer states. **CHC solving is still undecidable in general**, even with linear integer arithmetic.

The Horn structure still supports specialized **fixed-point and inductive-inference algorithms**, with SMT handling theory queries.

Why many verification problems fit

Many proof obligations have this shape:

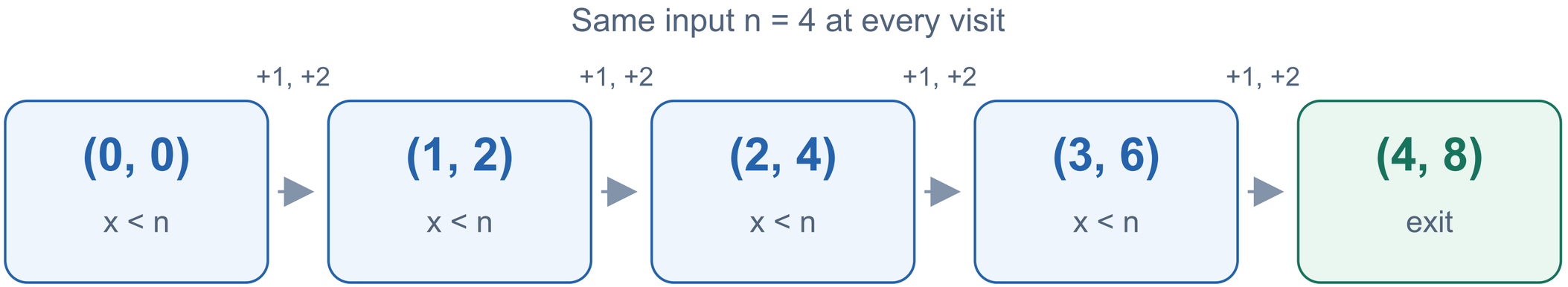
Established assertions, together with program semantics, must establish another assertion.

Program feature	Relation to infer	How the obligations connect it
Loop	An invariant at the loop head	Initialization establishes it; the body preserves it
Procedure	An input/output summary	A caller combines its state assertion with the callee's summary
Stateful system	A relation over persistent state	Every permitted operation must preserve it
Refinement type	A predicate on values	Computations must produce values satisfying the predicate

Loops and recursion create **cycles between these obligations**. A CHC solver can infer the related predicates together.

Watch the relationship, not just the values

For our loop with $n = 4$, record (x, y) at the loop head, including the final guard check.



Every displayed state satisfies y = 2x

The same relationship, $y = 2x$, survives every iteration, whatever n is.

A relation represents a set of states

Write $R(n, x, y)$ for membership in a proposed set of loop-head states.

A concrete execution	**A relation used in a proof**
One input and one sequence of states	A set covering every permitted input and iteration
For $n = 4$: $(0,0), (1,2), \ldots, (4,8)$	Perhaps $0 \le x \le n \ \wedge \ y = 2x$

We will constrain R before choosing its formula.

Where do the clauses come from?

Tool developers implement **translation rules derived from the language semantics**. The verifier applies those rules to the input program.

```
1 # A: before the assignment
2 x += 1
3 # B: after the assignment
```

Introduce unknown relations $R_A(n, x, y)$ and $R_B(n, x, y)$ at these locations. The assignment generates:

$$R_A(n, x, y) \implies R_B(n, x + 1, y).$$

The code fixes the update to x and leaves n, y unchanged. The solver finds formulas for R_A and R_B.

Control flow connects the relations. A loop returns to the same location, creating recursive clauses.

The invariant obligations form a CHC problem

The code fixes the guards and updates. **Only the definition of R is unknown.**

$$n \geq 0 \implies R(n, 0, 0)$$

$$R(n, x, y) \wedge x < n \implies R(n, x + 1, y + 2)$$

$$R(n, x, y) \wedge x \geq n \wedge y \neq 2n \implies \text{false}$$

Every rule must hold for **all** values of its variables.

The step uses R on both sides because the body returns to the loop head. The last rule encodes the assertion.

Solver input: these clauses. **Desired output:** a formula for R that makes them all true.

A solution defines the invariant

After `sat`, we can request a **model defining the unknown relation**. One possible model is:

$$R(n, x, y) \quad \equiv \quad \underbrace{0 \leq x \leq n}_{\text{bounds}} \quad \wedge \quad \underbrace{y = 2x}_{\text{preserved equality}} \quad .$$

Substitute this formula for **every occurrence** of R in the three clauses.

If all three implications are valid, this interpretation is a **model of the clauses** and supplies an invariant proving the assertion.

Three local checks cover every iteration

For $R(n, x, y) \equiv 0 \le x \le n \wedge y = 2x$:

Obligation	Why it holds
Start	$n \ge 0$ gives $0 \le 0 \le n$ and $0 = 2 \cdot 0$.
Step	$x < n$ gives $x + 1 \le n$ over integers; $y + 2 = 2(x + 1)$.
Exclude errors	$x \le n$ and exit condition $x \ge n$ give $x = n$, so $y = 2n$.

Induction turns closure under one step into coverage of all finite executions.

A failing execution rules out every solution

Change the update, keeping the **required assertion unchanged**:

```
1 x, y = 0, 0
2 while x < n:
3     x += 1
4     y += 3                  # changed update
5 assert y == 2 * n            # unchanged requirement
```

For $n = 2$, the start and step clauses force:

$$R(2,0,0), \qquad R(2,1,3), \qquad R(2,2,6).$$

The program produces $y = 6$. The assertion still requires $y = 2n = 4$.

The unchanged error clause therefore applies:

$$R(2,2,6) \wedge 2 \geq 2 \wedge 6 \neq 2 \cdot 2 \implies \text{false}.$$

No interpretation of R can satisfy all the clauses.

What the solver result tells us

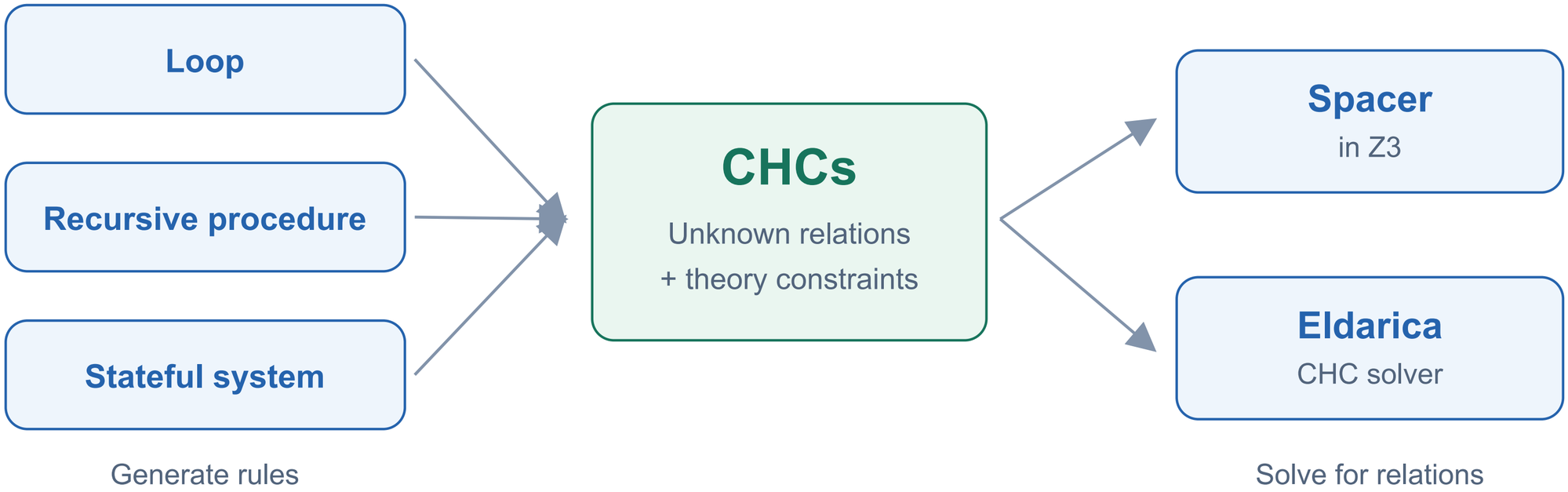
For our query asserting **all three safety clauses**:

Status	Evidence a supporting solver can provide	Meaning
`sat`	A **model defining the unknown relations**, such as a formula for R	The model supplies an invariant proving safety
`unsat`	A **refutation** deriving false; a verifier may reconstruct an error trace	No model exists; the exact encoding admits a failing execution
`unknown` / timeout	No conclusive model or refutation	The question remains unresolved

`check-sat` returns the status. Model/proof output must be requested and supported by the interface.

An SMT verification counterexample is itself a **model of a query encoding a violation**.

A common interface for verification



Verifier developers implement a translation from the language semantics. The tool generates clauses for each input program and reuses an engine for inferring invariants and summaries.

What makes this interface flexible?

The relations can describe different things. **The obligations connecting them must still fit the Horn form.**

Encoding choice	Examples
What a tuple records	A program state, input/output values, two executions, or security labels
What a rule summarizes	One instruction, a whole block, a procedure call, or a system operation
How data is represented	Mathematical integers, machine bit-vectors, arrays, or abstract values
What counts as an error	A bad access, a broken contract, unequal outputs, or a forbidden data flow

The encoding may describe concrete behavior or a **sound abstraction**. Each clause must conclude with **one relation fact or false**.

The encoding must preserve the connection to the program and its required property. Representing something as a relation is only the starting point.

Application: infer a procedure summary

```
1 count(n):                  // called with n >= 0
2     if n = 0 then return 0;
3     else return 1 + count(n - 1);
```

Let $S(n, r)$ relate an input n to a possible return value r.

A summary describes the result without expanding every recursive call.

What rules should S satisfy?

Recursive code becomes recursive rules

$$\text{true} \implies S(0,0)$$

$$n > 0 \land S(n-1,r) \implies S(n,r+1)$$

$$S(n,r) \land n \geq 0 \land r \neq n \implies \text{false}$$

One solution is $S(n,r) \equiv n \geq 0 \land r = n$.

The relation summarizes **every recursion depth**. These clauses establish a property of returned results.

A caller composes summaries

```
1 assume n >= 0;
2 a := count(n);          // first call: S(n,a)
3 b := count(a);          // second call: S(a,b)
4 assert b = n;
```

$$n \geq 0 \wedge S(n, a) \wedge S(a, b) \wedge b \neq n \implies \text{false}.$$

The same inferred summary is used twice. Its solution gives $a = n$ and $b = a$.

Two calls require two relation premises. Both use S, so $k = 2$ even though there is only one summary predicate.

The general clause form

$$\underbrace{\varphi \wedge P_1(\bar{t}_1) \wedge \cdots \wedge P_k(\bar{t}_k)}_{\text{body}} \quad \Longrightarrow \quad \underbrace{H}_{\text{head}}$$

φ is a known theory constraint. P_i is an unknown relation with arguments $\bar{t}_i$. There are k relation premises, possibly zero.

Part	**Loop step ($k = 1$)**	**Two calls ($k = 2$)**
φ	$x < n$	$n \geq 0 \wedge b \neq n$
Relation premises	$R(n, x, y)$	$S(n, a)$ and $S(a, b)$
H	$R(n, x + 1, y + 2)$	false

The premises may reuse the **same relation symbol with different arguments**.

Application: prove array accesses stay in bounds

```
1 assume n >= 0;              // a has length n
2 i := 0;
3 while i < n do
4     a[i] := 0;
5     i := i + 1;
```

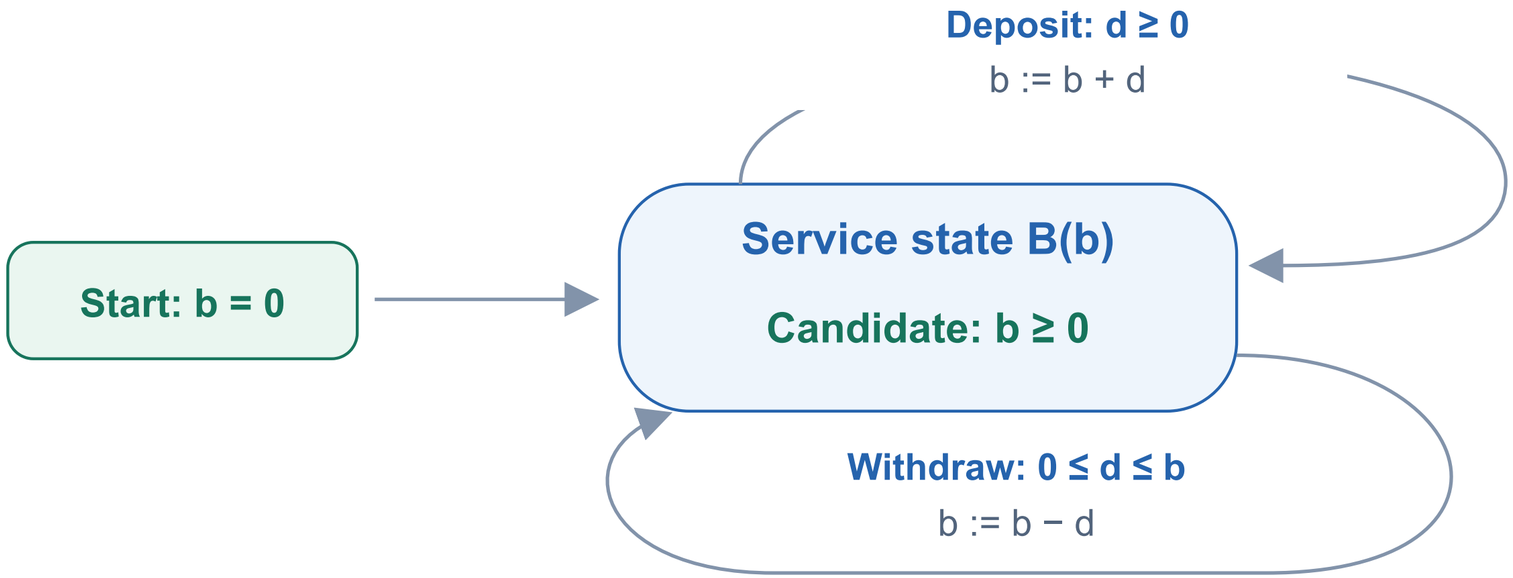
At the loop head, $i = n$ is allowed. **At the access, it is not.**

$$H(n, i) \wedge i < n \implies A(n, i)$$
$$A(n, i) \wedge (i < 0 \vee i \geq n) \implies \text{false}$$

H: states at the loop head. A: states at the access. Start and step rules constrain H as before.

Application: preserve a property across requests

A simplified balance service starts at $b = 0$. Deposits and withdrawals may repeat.



Infer a relation $B(b)$ preserved by **every permitted operation**. Exclude $B(b) \wedge b < 0$.

Removing the withdrawal guard $d \leq b$ permits an error.

Application: compare two implementations

Repeated addition	**Direct calculation**
$x := 0;\ y := 0$	$z := 2n$
While $x < n$: $x := x + 1;\ y := y + 2$	
Return y	Return z

With equal inputs $n \geq 0$, track both computations and forbid **different returned values**.

The relation $y = 2x \wedge z = 2n \wedge 0 \leq x \leq n$ connects their states.

At loop exit, $y = z$. This proves agreement of results; termination is a separate obligation.

Application: infer a refinement type

A **refinement type** adds a logical condition to an ordinary type.

```
1 def magnitude(x):                # x is a mathematical integer
2     if x >= 0: return x
3     return -x
```

Let $K(v)$ describe possible returned values. A caller requires a nonnegative result.

Code / requirement	**Constraint on K**
Nonnegative branch returns x	$x \geq 0 \implies K(x)$
Negative branch returns $-x$	$x < 0 \implies K(-x)$
Caller requires a nonnegative value	$K(v) \wedge v < 0 \implies \text{false}$

A solution $K(v) \equiv v \geq 0$ gives the return type $\{v : \text{Int} \mid v \geq 0\}$.

The unknown relation is now a type refinement.

Application: track sensitive data

```
1 def on_location_update(location):
2     cache.last = location       # sensitive source
3 def on_upload_click():
4     send(cache.last)             # public sink
```

Track a label: 1 means sensitive. $C(s)$ describes possible labels in the cache; $U(s)$ describes possible upload labels.

Code / requirement	Abstract rule
Cache the location	$\text{true} \implies C(1)$
Upload the cached value	$C(s) \implies U(s)$
Forbid sensitive uploads	$U(1) \implies \text{false}$

A location update followed by an upload forces the forbidden fact $U(1)$.

The analysis tracks sensitivity, rather than the exact location value.

Different analyses, the same logical interface

Task	What the encoding captures	Example system
Code safety	Control flow, memory models, assertions	**SeaHorn**
Smart-contract safety	Persistent state across transactions	**Solidity SMTChecker**
Refinement-type inference	Logical conditions on values	**Liquid Types**
Information-flow analysis	Abstract data flow through code and heap	**HornDroid**
Verification with borrowing	Value relations exploiting ownership	**RustHorn**

The work is shared: **design the encoding for the domain; reuse inference over recursive relations**.

Memory models, abstractions, and proof rules remain important parts of these systems.

What formulas can the solver search for?

CHCs specify the obligations. The solving algorithm chooses a language and a search strategy for defining R.

Technique	**Candidate definitions of R**
Given predicate pool	Conjunctions or Boolean combinations of the supplied predicates
Templates	Fixed formula shapes with unknown parameters, such as $an + bx + cy \leq d$
Predicate / lemma generation	New formulas constructed during solving within a supported theory language

For our example, a useful language is **linear-arithmetic formulas over n, x, y.**

A predicate-abstraction solver can **grow its pool** using interpolation; the pool need not be fixed in advance.

How can a solver discover the relation?

Our returned model can be expressed using three linear-arithmetic predicates:

$$x - n \leq 0, \qquad y - 2x \leq 0, \qquad 2x - y \leq 0.$$

Their conjunction gives $R(n, x, y) \equiv x \leq n \wedge y = 2x$.

A **Spacer-style** method learns lemmas during the search:

1. Explore whether an error can be reached.
2. Generalize failed attempts into new theory lemmas using SMT reasoning.
3. Check whether the learned constraints form an inductive solution.

The predicates need not come from a fixed supplied list. A real error instead gives a derivation of false.

A useful interface does not make verification easy

- **Inference can fail to finish.** CHC satisfiability is undecidable in general, even over linear integer arithmetic.
- **The search space matters.** A restricted language or incomplete inference method can miss a valid invariant.
- **The encoding matters.** Integers, machine arithmetic, arrays, and heaps require appropriate semantics. Abstraction can introduce false alarms.
- **Safety does not imply termination.** An error may be unreachable because execution never returns.

`unknown` or a timeout means the requested proof has **not been established**.

Try writing the rules

```
1 assume n >= 0;
2 x := 0;
3 while x < n do
4     x := x + 2;
5 assert x <= n + 1;
```

Use $R(n, x)$ for loop-head states.

Write a start rule, a step rule, and an error-exclusion rule.

Can you find a solution without recording that x is even?

What CHCs let us ask

Find relations that make all the proof obligations hold.

What we already studied	What CHCs add
Hoare logic and verification conditions	Leave intermediate assertions unknown and constrain them
Model checking and reachable states	Search for an inductive safe set
SMT and background theories	Combine arithmetic constraints with recursive relations

The unknowns can be loop invariants, procedure summaries, type refinements, or facts in a security analysis.