Constrained Horn Clauses for Verification and Synthesis

Grigory Fedyukovich

Aug 4, IARCS

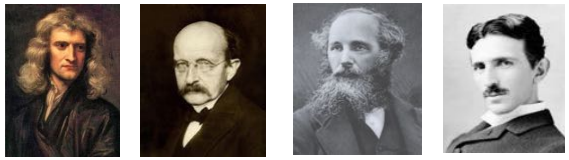
Automated Reasoning about Software

Logic-based verification and synthesis

- A user provides a program and a desired specification
  - *E.g., program never writes outside of allocated memory*
- A tool automatically constructs a model of the program
  - *Program = formula*
- Use decision procedures to reason about formulas
  - *Thus, deriving properties about programs*

Inspired by methods in applied science

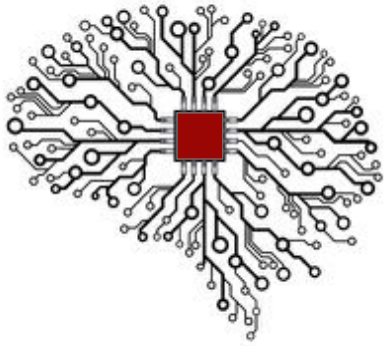
- E.g., physicists in 17-19th centuries



Automated Reasoning Today

Automated verification with SMT solvers

- Scale to large industrial applications



Constrained Horn Clauses (CHC):

- Symbolic representation of programs
- Safety specifications (assertions)

Verification = CHC translation + CHC solving

- E.g., *SeaHorn + Spacer*, *JayHorn + Eldarica, or RustHorn + HoIce*

“Guess-and-check” invariant generation for CHC solving:

- Iteratively construct proofs using SMT solvers

Successful techniques are based on simple ideas

- Property Directed Reachability / IC3
- Machine Learning / Syntax-Guided methods

SAT: Boolean Satisfiability

Given a Boolean expression, using "and" ($\wedge$), "or" ($\vee$) and "not" ($\neg$),

- Is there an assignment of $true$ and $false$ to the variables that makes the expression equal $true$?

Example:

- $(x \vee \neg y) \wedge (x \vee z \vee w) \wedge \neg z \wedge (w \vee y)$
- Solution: $x = y = w = true, z = false$

DPLL (Davis-Putnam-Logemman-Loveland, '60)

- Smart enumeration of all possible SAT assignments

Many heuristics used by modern tools

- Allow solving instances with millions of variables

SMT: Satisfiability Modulo Theory

Satisfiability of Boolean formulas over atoms in a theory

- E.g., $x > 2 * y - 1 \wedge y < 4$
- A solution: $x = y = 0$

Extends syntax of Boolean formulas with functions and predicates

- +, -, *div, select, store, bvadd*, etc.

Existing solvers support many theories useful for program verification

- Equality and Uninterpreted Functions: $x = y \implies f(x) = f(y)$
- Real/Integer Linear Arithmetic: $x > 2 * y - 1 \wedge y < 4$
- Unbounded Arrays: $A = store(B, i, select(C, j))$
- Bitvectors (a.k.a. machine integers): $y = x \texttt{ >> } 4 \wedge z = y \texttt{ \& } x$
- Floating point: $0.1 * x = 3.6 * y$

Solving based on SAT

- As well as multiple heuristics for the theory reasoning

CHC: Constrained Horn Clauses

Formula in first order logic:

$$\varphi \wedge p_1(V) \wedge \ldots \wedge p_k(V) \implies H$$

- where A is a constraint language
  (e.g., (non-)linear arithmetic, arrays, bit-vectors, etc.)
- φ is a constraint in A
- $p_1 \ldots p_k$ are uninterpreted relation symbols
- each $p_i(V)$ is an application of the predicate to variables
- H is either some application $p_i(V)$ or $false$

System of CHCs

- Only one CHC with $H = false$
- Has a solution if there exists an interpretation for each p_i
  making each CHC valid

CHC Solvers

IC3/PDR

[Hoder, Bjorner, SAT'12]

[Cimatti, Griggio, CAV' 12]

[McMillan, CAV'14]

[Komuravelli, Gurfinkel, Chaki, Clarke, CAV'14]

CEGAR

[Unno, Terauchi, TACAS'15]

[Hojjat, Ruemmer, FMCAD'18]

[Vazou, Seidel, Jhala, etc, ICFP'14]

[Dietsch, Heizmann, Hoenicke, Nutz, Podelski, HCVS/PERR'19]

Abstract Interpretation

[Kafle, Gallagher, Morales, CAV'16]

[Bakhirkin, Monniaux, SAS'17]

CEGIS/SyGuS

[Beyene, Popeea, Rybalchenko, CAV'13]

[Fedyukovich, Prabhu, Madhukar, Gupta, FMCAD'18]

ML

[Champion, Chiba, Kobayashi, TACAS'18]

[Zhu, Magill, Jagannathan, PLDI'18]

Abduction

[Dillig, Dillig, Li, McMillan, OOPSLA'13]

The rest of the talk

[all papers co-authored by Fedyukovich]

FMCAD 2017:

Sampling Invariants from Frequency Distributions

CAV 2019:

Quantified Invariants via Syntax-Guided Synthesis

TACAS 2021:

Bridging Arrays and ADTs in Recursive Proofs

CP 2019:

Lemma Synthesis for Automating Induction over Algebraic Data Types

PLDI 2021:

Specification Synthesis with Constrained Horn Clauses

Example

Program in C

```
int j, m, N = nondetInt();
int *A = nondetArray(N);
int i = 0;
```

```
while (i < N) {
  if (m < A[i]) m = A[i];
  i++;
}
```

```
assume(0 ≤ j < N);
assert(m ≥ A[j]);
```

CHC-encoding

Symbolic execution (via *static single assignment* transformation)

Uninterpeted predicate

$$i = 0 \implies \boldsymbol{inv}(A, i, m, N)$$

$$\boldsymbol{inv}(A, i, m, N) \wedge i < N \wedge m' = ite(m < A[i], A[i], m) \wedge i' = i + 1 \implies \boldsymbol{inv}(A, i', m', N)$$

$$\boldsymbol{inv}(A, i, m, N) \wedge i \geq N \wedge 0 \leq j < N \wedge \neg(m \geq A[j]) \implies \bot$$

Verification Conditions as CHCs

Compact representation of a loop

$$\begin{aligned} \mathrm{INIT}(V) &\implies Inv(V) \\ Inv(V) \wedge \mathrm{TR}(V, V') &\implies Inv(V') \\ Inv(V') \wedge \mathrm{BAD}(V') &\implies \bot \end{aligned}$$

Getting finite traces

- Unroll the loop some k times
- Evaluate a so-called Bounded Model Checking (BMC) formula

$$\mathrm{INIT}(V) \wedge \underbrace{\mathrm{TR}(V, V') \wedge \mathrm{TR}(V', V'') \wedge \ldots \wedge \mathrm{TR}(V^{(k-1)}, V^{(k)})}_{k} \wedge \mathrm{BAD}(V^{(k)})$$

  - here, each $V^{(i)}$ is a fresh copy of V

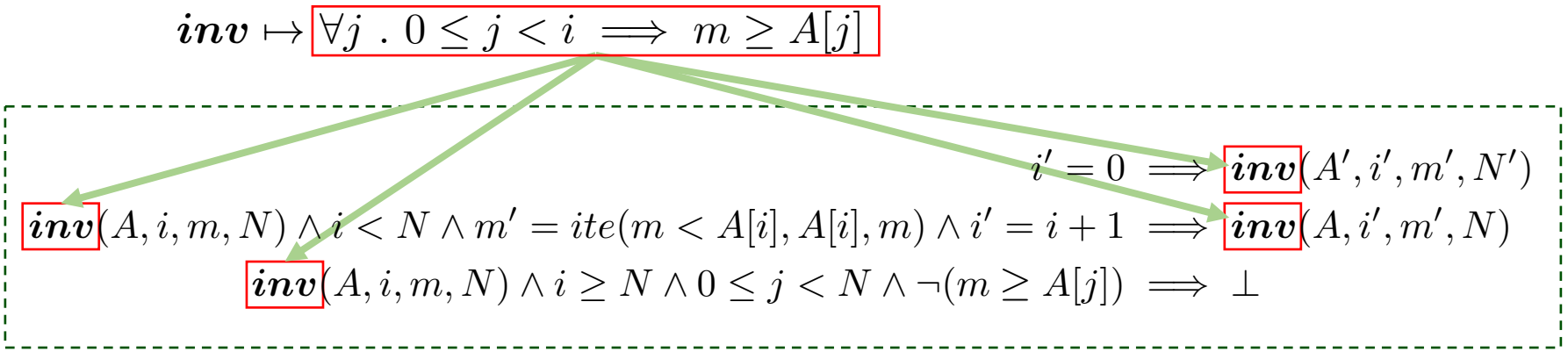
- Increase k for finding more bugs
- But to prove that there are no bugs, *we need inductive invariants*

Solutions for CHCs

Inductive invariant:

- Describes all initial states
- If it describes a state from where a transition starts, then it describes a state where the transition ends
- Describes no bad states

Example



Our approach: FreqHorn

[Fedyukovich et al, FMCAD'17]

High-level view:

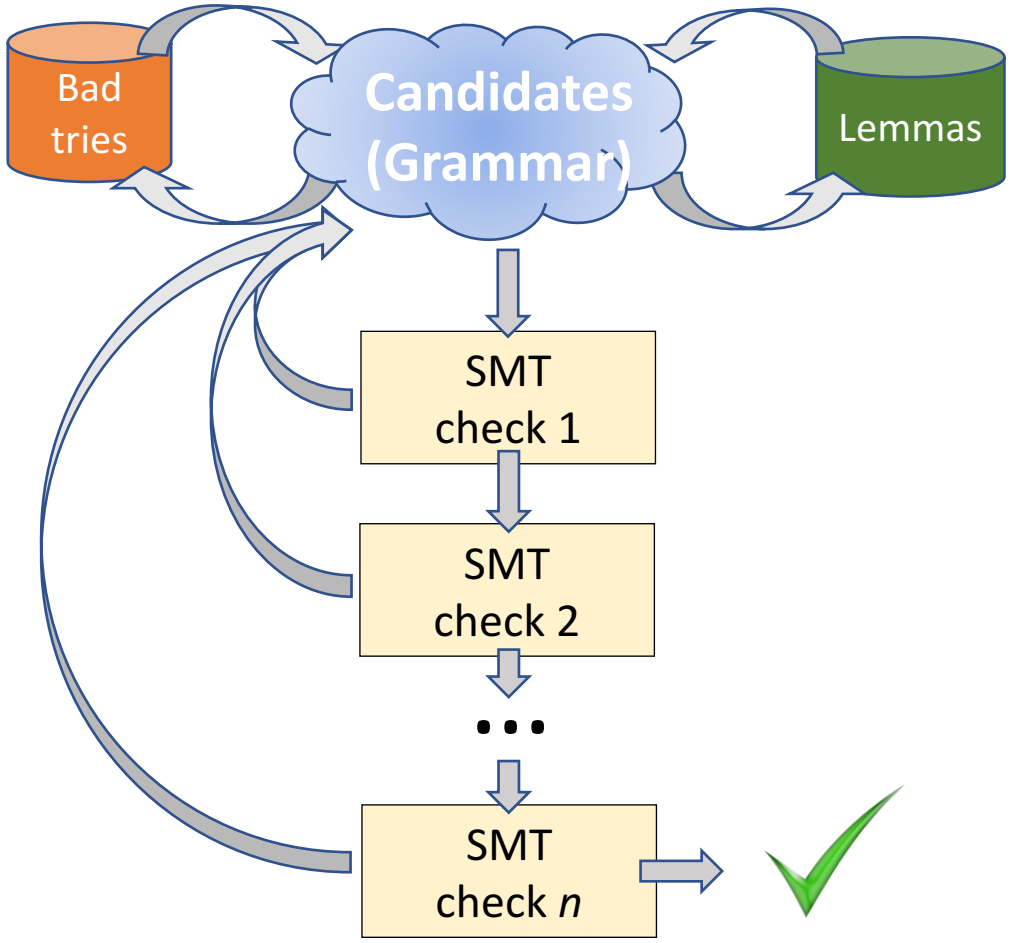
- Loop between a candidate generator and an SMT-solver

Candidate generator

- **Syntax-Guided Synthesis** (**SyGuS**)
- Learning from positive / negative candidates

SMT-based decision maker

- Off-the-shelf SMT solvers (for termination, non-termination, safety, etc…)



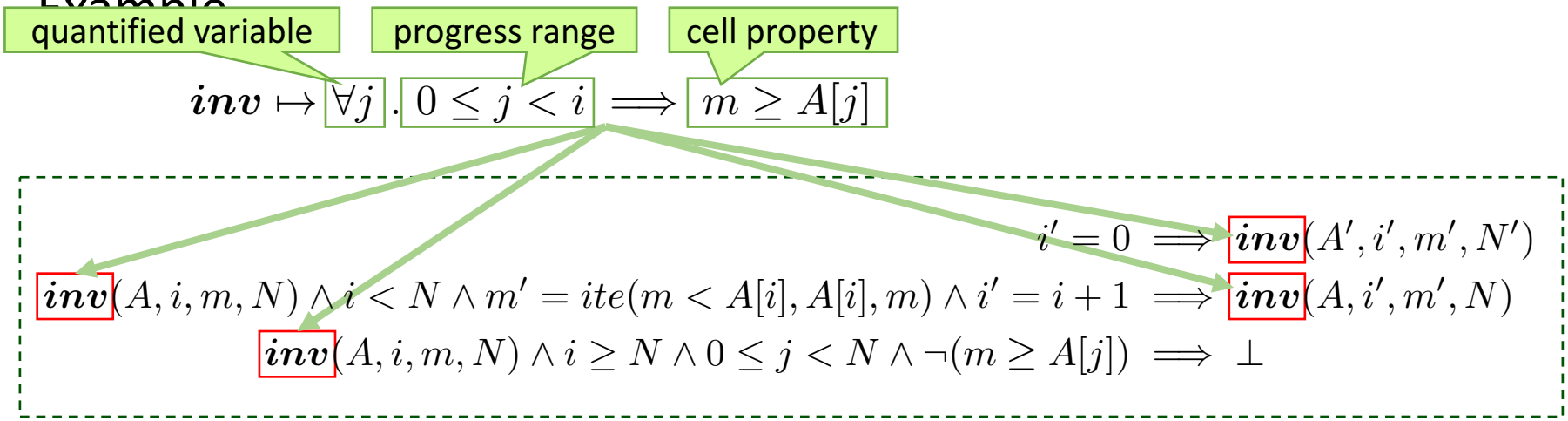
Quantified Solutions for CHCs

[Fedyukovich et al., CAV 2019]

Inductive invariant:

- Describes all initial states
- If it describes a state from where a transition starts, then it describes a state where the transition ends
- Describes no bad states

Example



Obtaining Quantified Invariants

[Fedyukovich et al., CAV 2019]

$$\forall \vec{q} \;.\; progressRange(\vec{q}, counters) \implies cellProperty(\vec{q}, vars)$$

Identify $counters \subseteq vars$

- Variables and the direction of their change

Introduce fresh variables $\vec{q}$ to be quantified

- One per each counter
- Progress range based on termination conditions of loops and initial values of counters

SyGuS-based sampling of cell properties

- Using automatically generated formal grammars
- Similar to the case of numeric invariants

Relational Verification with CHCs

[Mordvinov et al., LPAR'2017, FMCAD'19]

Need for multiple invariants

- Solutions are often inexpressible in the constraint language

Program transformation should help

- Simplifies the verification condition
- Preserves semantics

Can be made directly on the level of CHCs

- Using a product-transformation

Applications:

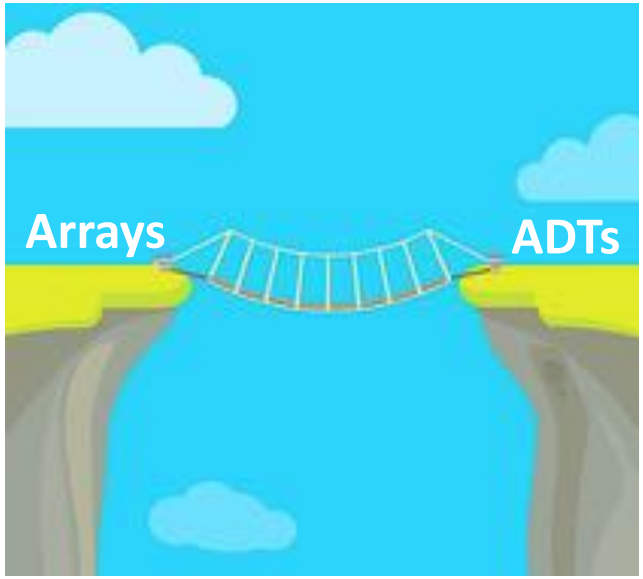
- Information-flow Checking
- Equivalence Checking

Application: Bridging arrays and ADT

[Fedyukovich et al., TACAS'2021]

Data structures in programs

- Arrays
- Algebraic Data Types (ADT)



Automated reasoning is hard

- Arrays need (universal) *quantifiers*
- ADTs need *recursion*

Relational verification is even harder

- **Goal**: prove that an ADT-implementation and an Array-implementation of the same interface are equivalent
- **Solution**: synthesize relational invariants which are recursive and quantified

Example

Two implementations of a stack

```
class ListStack:
  def init():
    xs = nil

  def push(in):
    xs = cons(in, xs)

  def pop():
    assert xs != nil
    out = xs.head
    xs = xs.tail
    return out
```

```
class ArrStack:
  def init():
    n = 0
    a = [...]

  def push(in):
    a[n] = in
    n = n + 1

  def pop():
    assert n > 0
    n = n - 1
    return a[n]
```

Example

Two implementations of a stack

$$\boldsymbol{R}(xs, n, a) = \begin{cases} n = 0 & \text{if } xs = \texttt{nil} \\ n > 0 \wedge y = a[n-1] \wedge \boldsymbol{R}(ys, n-1, a) & \text{if } xs = \texttt{cons}(y, ys) \end{cases}$$

generalize

$$\boldsymbol{R}(xs, n, a) = \begin{cases} \_(n, a) & \text{if } xs = \texttt{nil} \\ \_(y, ys, n, a, \_) \wedge \boldsymbol{R}(ys, \_) & \text{if } xs = \texttt{cons}(y, ys) \end{cases}$$

$$\boldsymbol{R}(xs, cs) = \begin{cases} \_(cs) & \text{if } xs = \texttt{nil} \\ \exists\ cs_r\ .\ \_(y, ys, cs, cs_r) \wedge \boldsymbol{R}(ys, cs_r) & \text{if } xs = \texttt{cons}(y, ys) \end{cases}$$

```
return out
```

Key Ideas

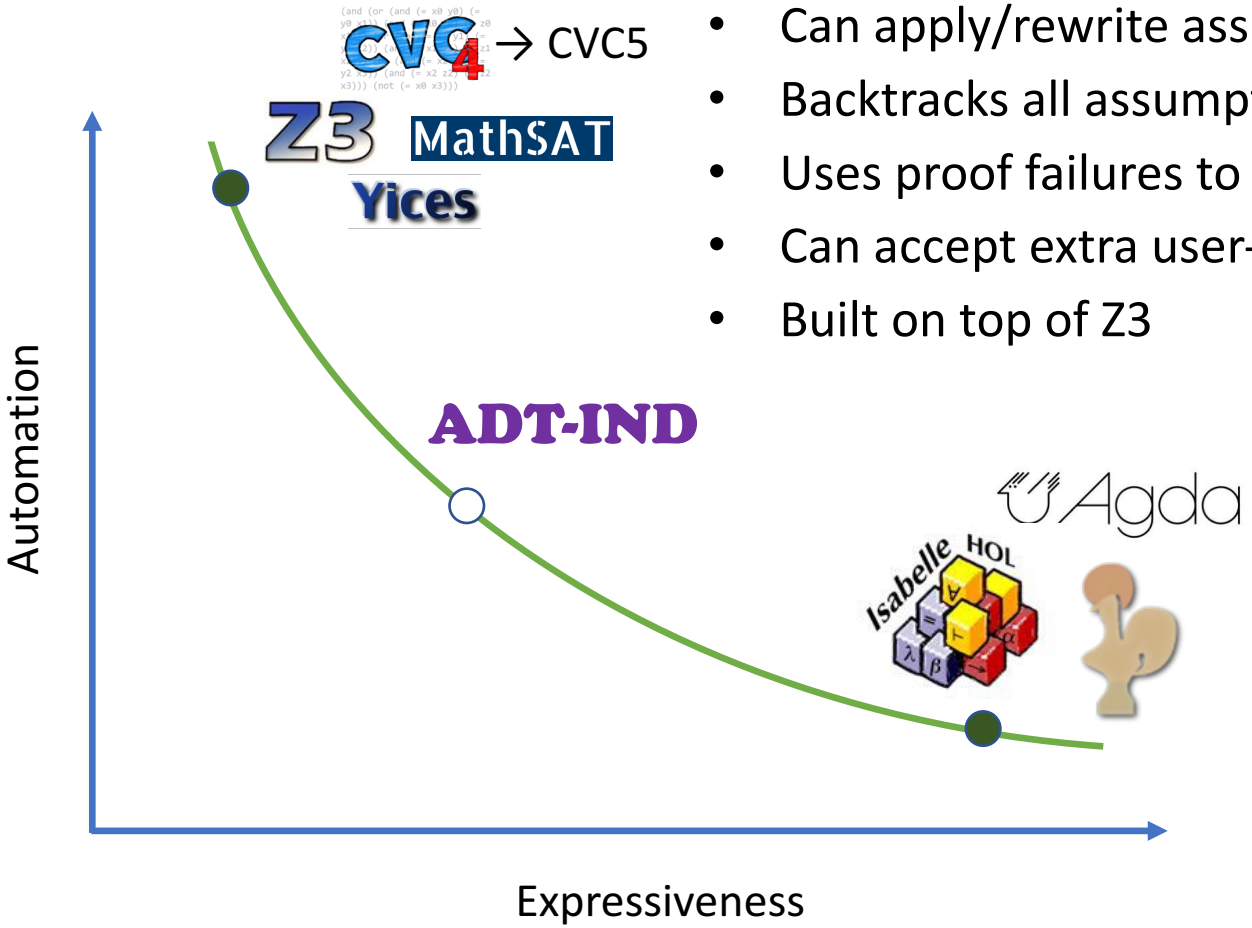
- Invariants describe the correspondence between programs in terms of the user-visible *input* and *output* variables
  - coincidence of preconditions
  - equivalence of outputs
  - the initiation and the consecution constraints
- Solutions for CHC use the recursive template
- CHC solver instantiates holes gradually
- CHC solver distinguishes:
  - *Producers*, i.e., that make the ADT “larger”
  - *Consumers*, i.e., that make the ADT “smaller”
  - *Noops*, i.e., that do not change the ADT

$$\boldsymbol{R}(xs, cs) = \begin{cases} \_(cs) & \text{if } xs = \texttt{nil} \\ \exists\ cs_r\ .\ \_(y, ys, cs, cs_r) \wedge \boldsymbol{R}(ys, cs_r) & \text{if } xs = \texttt{cons}(y, ys) \end{cases}$$

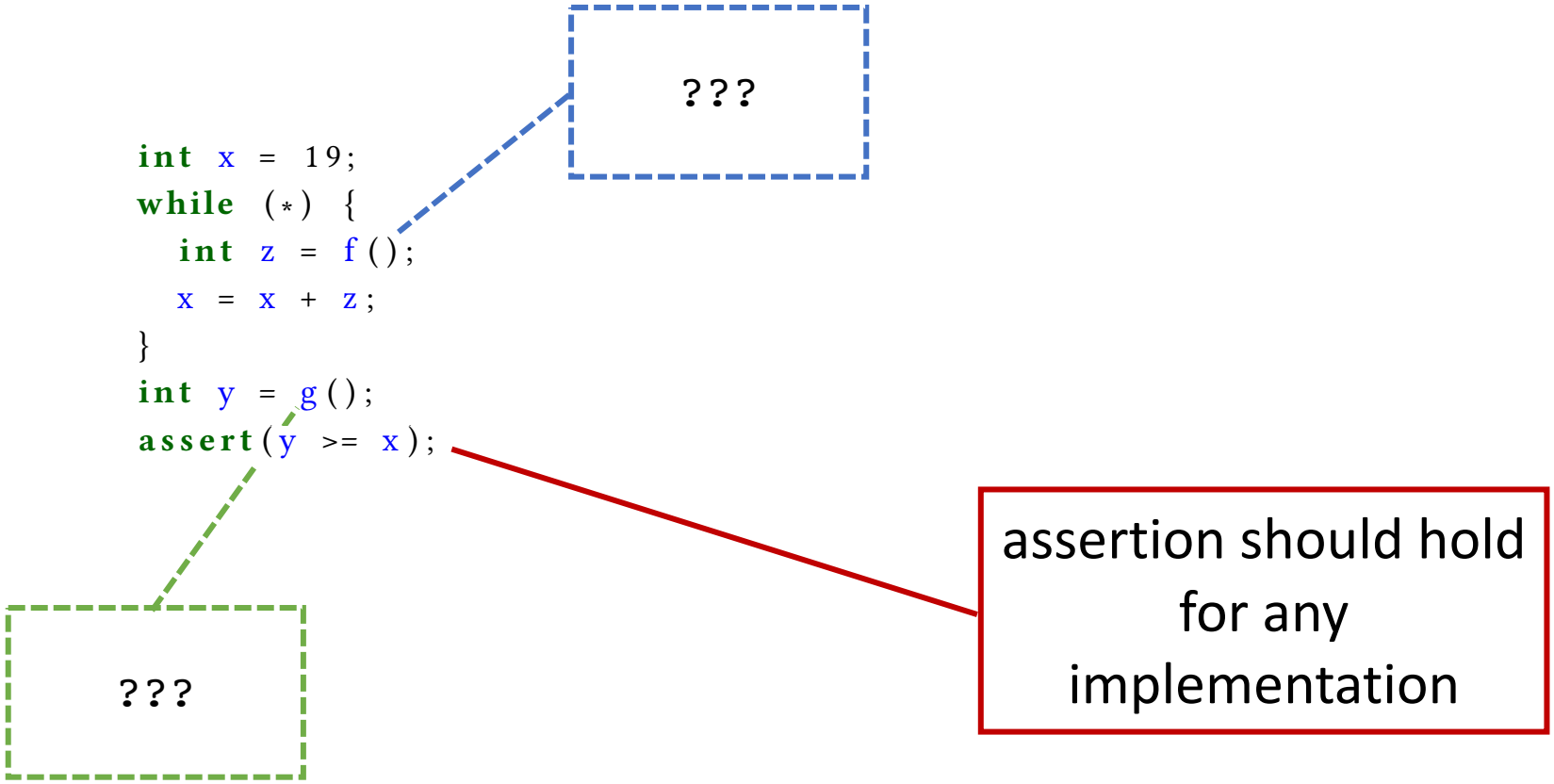
The **ADT-IND** prover

[Yang et al., CP'2019]

- Proves the validity of universally-quantified formulas by induction
- Can apply/rewrite assumptions
- Backtracks all assumptions fail
- Uses proof failures to synthesize lemmas
- Can accept extra user-lemmas
- Built on top of Z3

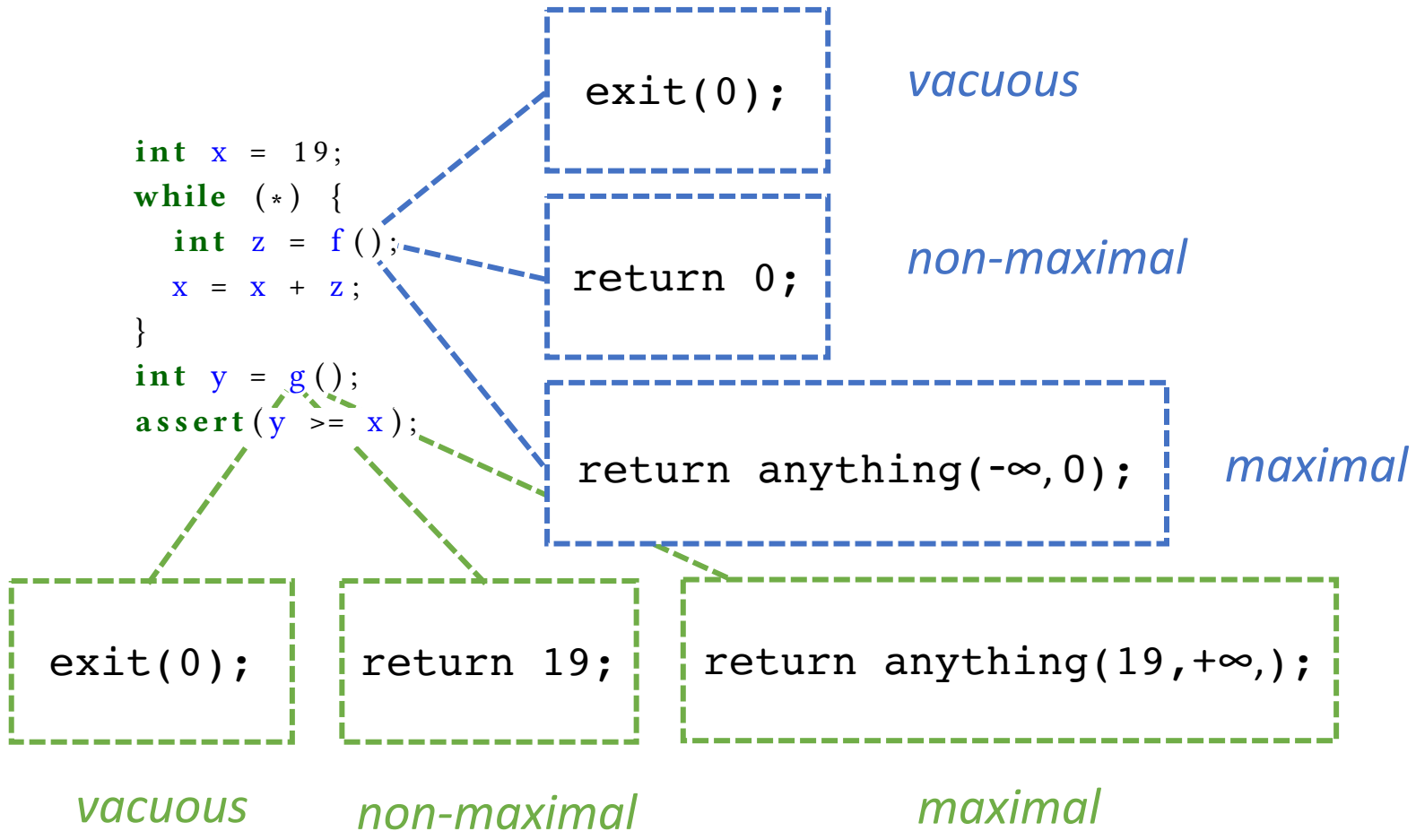


Specification Synthesis



Specification Synthesis

Possible implementations



Specification Synthesis as CHC Problem

[Prabhu et al., PLDI'2021]

```
int x = 19;
while (*) {
  int z = f();
  x = x + z;
}
int y = g();
assert(y >= x);
```

$$x = 19 \implies \boldsymbol{inv}(x)$$
$$\boldsymbol{inv}(x) \wedge \boldsymbol{f}(z) \wedge x' = x + z \implies \boldsymbol{inv}(x')$$
$$\boldsymbol{inv}(x) \wedge \boldsymbol{g}(y) \wedge \neg(y \geq x) \implies \bot$$

Vacuous solutions

- Simply make the bodies of CHCs unsatisfiable
- Existing CHC solvers can easily discover them

Non-vacuous solutions

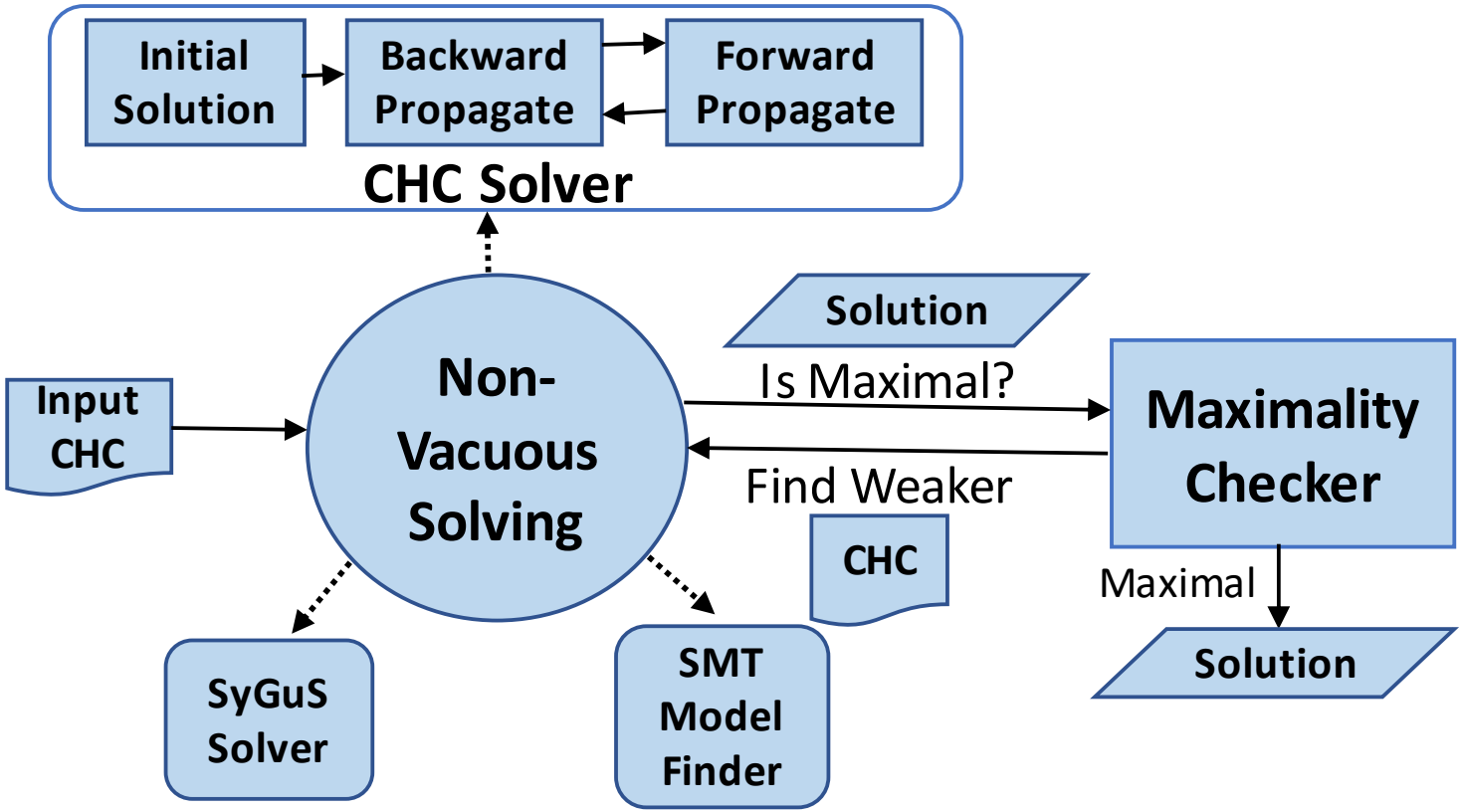
- Bodies of CHCs (and solutions) are satisfiable
- Require a CHC solver to do an extra SMT check for each solution

Maximal solutions

- Any weakening of solutions leads to assertion violation
- Require a CHC solver to check for assertion violations in a loop

HornSpec: Big idea

[Prabhu et al., PLDI'2021]



Evaluation

The tool

- FreqHorn https://github.com/grigoryfedyukovich/aeval/tree/rnd
- Built on top of the Z3 SMT solver
- Fully automated workflow
- Parallelized using Message Passing Interface

Comparable with

- Spacer, Eldarica, HOICE, MCMC, ICE, etc
- > 500 public benchmarks

Strong points of FreqHorn

- Quantified Invariants over Arrays
- ADT support
- Forward/Backward propagation using Quantifier Elimination
- Maximal Specification Synthesis

UNIVERSITY of WASHINGTON

PRINCETON UNIVERSITY

TATA CONSULTANCY SERVICES

Conclusion and Future Work

- Safety verification
- Relational Verification
- Specification Synthesis

Solving Constrained Horn Clauses

- Automatic parallelization
- Security verification
- Termination Analysis

- Quantified Specification Synthesis
- Performance-aware Synthesis
- Termination-aware Synthesis