Quantifiers on Demand

Arie Gurfinkel, Sharon Shoham, and Yakir Vizel

ATVA 2018
October 9, 2018

Constrained Horn Clauses (CHC)

A Constrained Horn Clause (CHC) is a FOL formula of the form

$$\forall V \cdot (\varphi \wedge p_1[X_1] \wedge \cdots \wedge p_n[X_n]) \rightarrow h[X]$$

where

- $\mathcal{T}$ is a background theory (e.g., Linear Arithmetic, Arrays, Bit-Vectors, or combinations of the above)
- V are variables, and X_i are terms over V
- φ is a constraint in the background theory $\mathcal{T}$
- $p_1, \ldots, p_n, h$ are n-ary predicates
- $p_i[X]$ is an application of a predicate to first-order terms

UNIVERSITY OF WATERLOO

CHC Satisfiability

A $\mathcal{T}$-**model** of a set of a CHCs Π is an extension of the model M of $\mathcal{T}$ with a first-order interpretation of each predicate p_i that makes all clauses in Π true in M

A set of clauses is **satisfiable** if and only if it has a model

- This is the usual FOL satisfiability

A $\mathcal{T}$**-solution** of a set of CHCs Π is a substitution σ from predicates p_i to $\mathcal{T}$-formulas such that $\Pi\sigma$ is $\mathcal{T}$-valid

In the context of program verification

- a program satisfies a property iff corresponding CHCs are satisfiable
- solutions are inductive invariants
- refutation proofs are counterexample traces

Procedures for Solving CHC(T)

Predicate abstraction by lifting Model Checking to HORN

- QARMC, Eldarica, …

Maximal Inductive Subset from a finite Candidate space (Houdini)

- TACAS'18: hoice, FreqHorn

Machine Learning

- PLDI'18: sample, ML to guess predicates, DT to guess combinations

Abstract Interpretation (Poly, intervals, boxes, arrays…)

- Approximate least model by an abstract domain (SeaHorn, …)

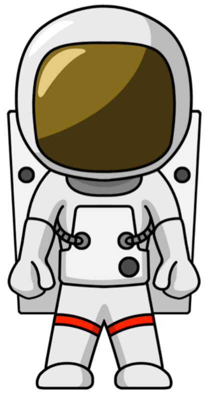
Interpolation-based Model Checking

- Duality, QARMC, …

SMT-based Unbounded Model Checking (IC3/PDR)

- Spacer, Implicit Predicate Abstraction

Spacer: Solving SMT-constrained CHC



Spacer: a solver for SMT-constrained Horn Clauses

- now the default (and only) CHC solver in Z3
  - https://github.com/Z3Prover/z3
  - dev branch at https://github.com/agurfinkel/z3

Supported SMT-Theories

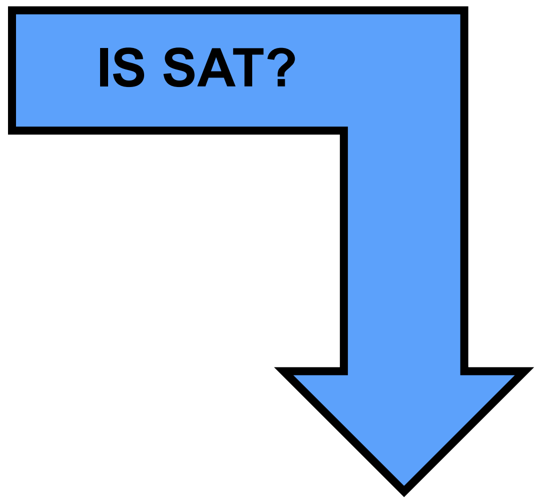
- Linear Real and Integer Arithmetic
- Quantifier-free theory of arrays
- ***Universally quantified theory of arrays + arithmetic (this talk!)***
- Best-effort support for many other SMT-theories
  - data-structures, bit-vectors, non-linear arithmetic

Support for Non-Linear CHC

- for procedure summaries in inter-procedural verification conditions
- for compositional reasoning: abstraction, assume-guarantee, thread modular, etc.

Program Verification with HORN(LIA)

```
z = x; i = 0;
assume (y > 0);
while (i < y) {
  z = z + 1;
  i = i + 1;
}
assert(z == x + y);
```




```
z = x & i = 0 & y > 0                          → Inv(x, y, z, i)
Inv(x, y, z, i) & i < y & z1=z+1 & i1=i+1      → Inv(x, y, z1, i1)
Inv(x, y, z, i) & i >= y & z != x+y            → false
```

In SMT-LIB

```
(set-logic HORN)

;; Inv(x, y, z, i)
(declare-fun Inv ( Int Int Int Int) Bool)

(assert
 (forall ( (A Int) (B Int) (C Int) (D Int))
        (=> (and (> B 0) (= C A) (= D 0))
           (Inv A B C D)))
 )
(assert
 (forall ( (A Int) (B Int) (C Int) (D Int) (C1 Int) (D1 Int) )
        (=>
         (and (Inv A B C D) (< D B) (= C1 (+ C 1)) (= D1 (+ D
1)))
         (Inv A B C1 D1)
         )
        )
 )
(assert
 (forall ( (A Int) (B Int) (C Int) (D Int))
        (=> (and (Inv A B C D) (>= D B) (not (= C (+ A B))))
           false
           )
        )
 )

(check-sat)
(get-model)
```

```
$ z3 add-by-one.smt2
sat
(model
  (define-fun Inv ((x!0 Int) (x!1 Int) (x!2 Int) (x!3 Int)) Bool
    (and (<= (+ x!2 (* (- 1) x!0) (* (- 1) x!3)) 0)
         (<= (+ x!2 (* (- 1) x!0) (* (- 1) x!1)) 0)
         (<= (+ x!0 x!3 (* (- 1) x!2)) 0)))
)
```

```
Inv(x, y, z, i)

    z  = x + i

    z <= x + y
```

HORN(ALIA): Arrays + LIA

```
int A[N];
for (int i = 0; i < N; ++i)
   A[i] = 0;
int j = nd();
assume(0 <= j < N);
assert(A[j] == 0);
```

IS SAT?

```
Inv(A, N, 0)
Inv(A, N, i) & i < N → Inv(A[i := 0], N, i+1)
Inv(A, N, i) & i >= N & 0 <= j < N & A[j] != 0 → false
```

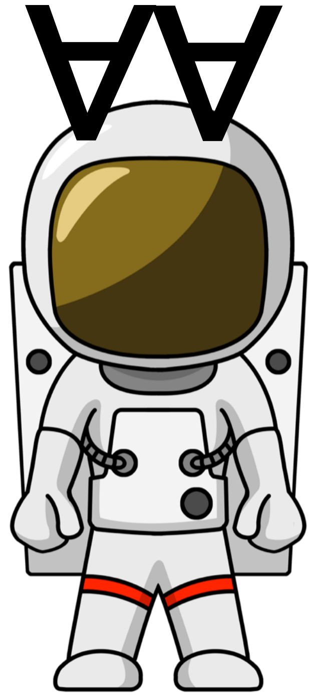
In SMT-LIB

```
(set-logic HORN)

;; Inv(A, N, i)
(declare-fun Inv ( (Array Int Int) Int Int ) Bool)

(assert
 (forall ( (A (Array Int Int)) (N Int) (C Int)) (Inv A N 0)))

(assert
 (forall ( (A (Array Int Int)) (N Int) (i Int) )
        (=>
         (and (Inv A N i) (< i N) )
         (Inv (store A i 0) N (+ i 1))
         )
        )
 )
(assert
 (forall ( (A (Array Int Int)) (N Int) (i Int) (j Int))
        (=> (and (Inv A N i )
                (>= i N) (<= 0 j) (< j N) (not (= (select A
j) 0)))
          false
          )
        )
 )

(check-sat)
(get-model)
```

```
$ z3 -t:100 array-zero.smt2
canceled
unknown
```

Inv(A, N, i)

∀ 0 <= j < i < N → A[j] = 0

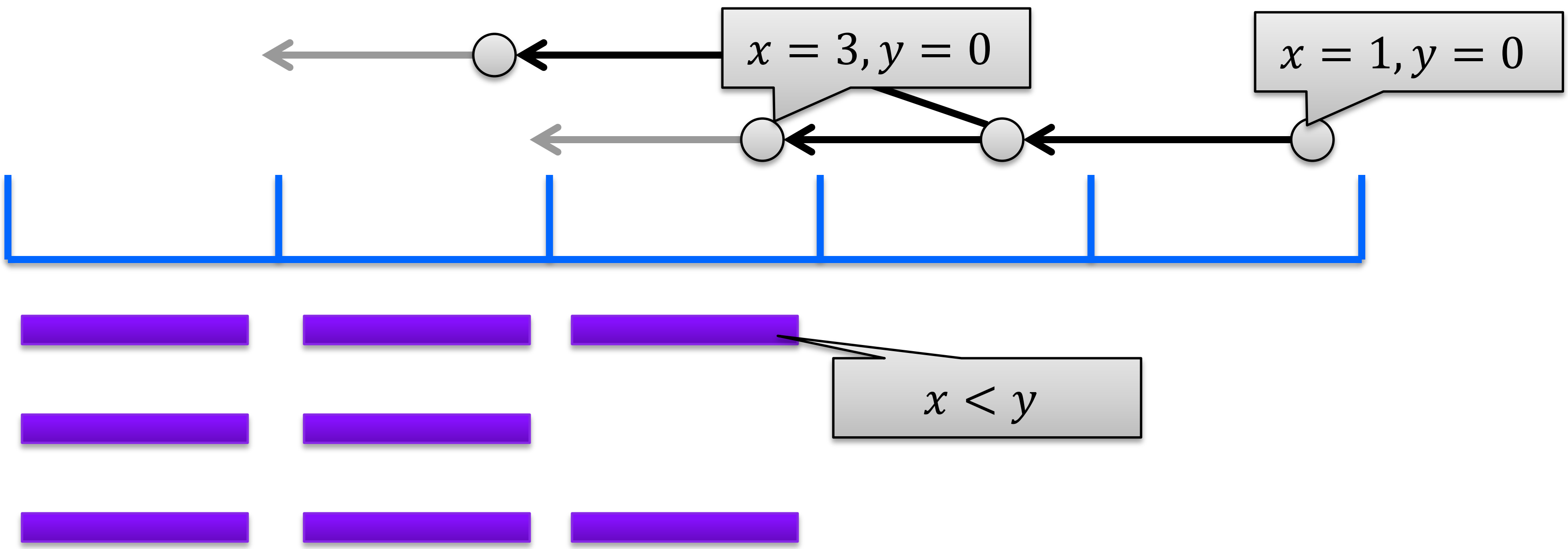


Extends Spacer with reasoning about quantified solutions

QUIC3: QUANTIFIED IC3

IC3/PDR In Pictures: MkSafe

MkSafe



Predecessor

$$\text{find } M \text{ s.t. } M \models F_i \wedge Tr \wedge m'$$

$$\text{find } m \text{ s.t. } (M \models m) \wedge (m \implies \exists V' \cdot Tr \wedge m')$$

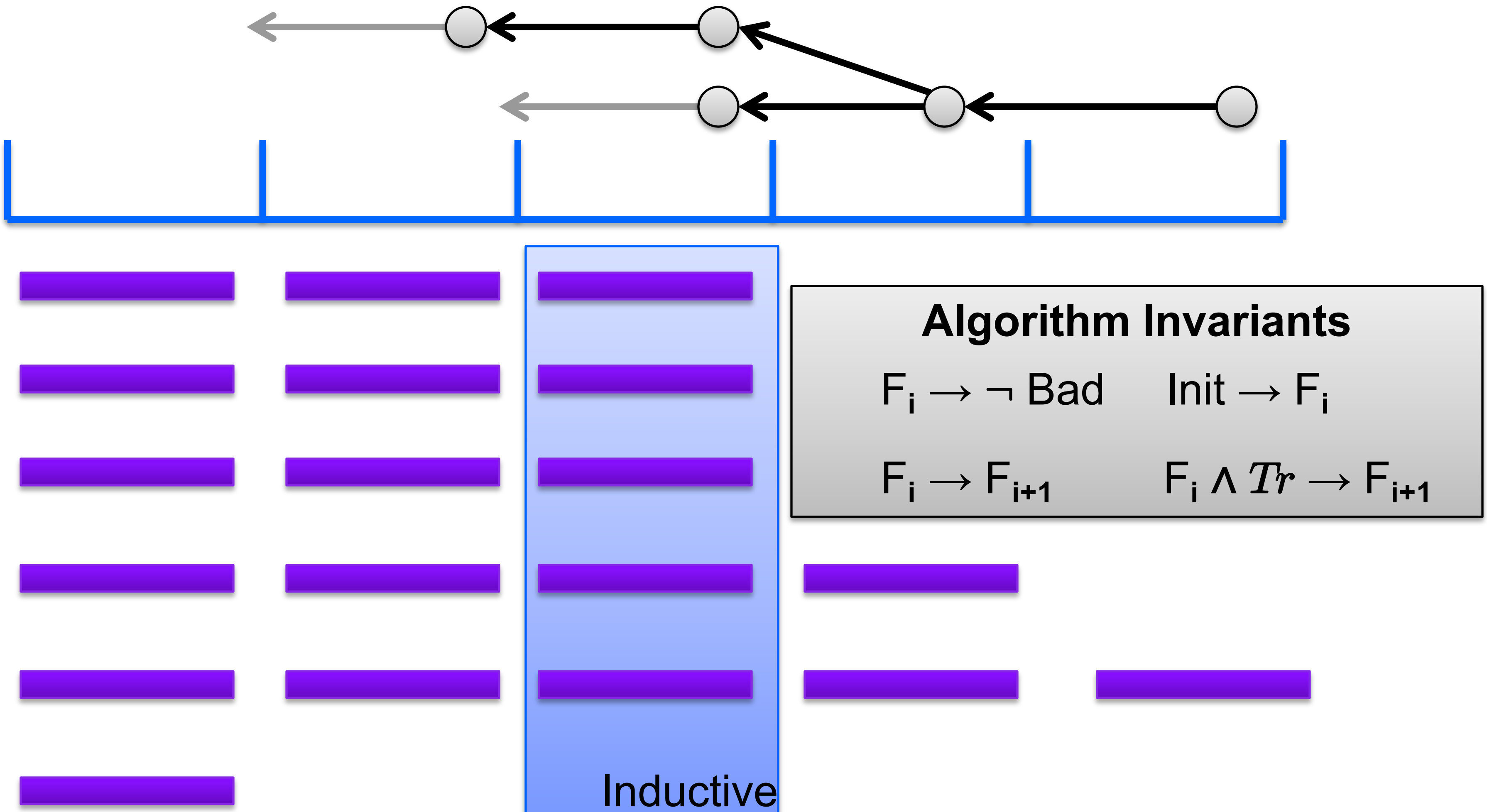
NewLemma

$$\text{find } \ell \text{ s.t. } (F_i \wedge Tr \implies \ell') \wedge (\ell \implies \neg m)$$

UNIVERSITY OF WATERLOO

Push

IC3/PDR in Pictures: Push



$$\text{SMT-query: } \vdash \ell \wedge F_i \wedge Tr \implies \ell'$$

Predecessor in array-zero example

```
Inv(A, N, i) & i >= N & 0 <= j < N & A[j] != 0 → false
```

Tr: i < N & 0 <= j < N & A[j] != 0

POB: true

$$\exists j \cdot i \geq N \land 0 \leq j < N \land A[j] \neq 0$$

$$= \quad i \geq N \land \exists j \cdot (0 \leq j < N \land A[j] \neq 0)$$

$$= \quad ???$$

No way to eliminate the existential quantifier!

- can use the value of j in the current model
- but this only works when A[j] is not important

UNIVERSITY OF WATERLOO

Quantified POBs and Lemmas

Must deal with existentially quantified POBs

$$\text{find } M \text{ s.t. } M \models F_i \land Tr \land m'$$

$$\text{find } m \text{ s.t. } (M \models m) \land (m \implies \exists V' \cdot Tr \land m')$$

Learning universally quantified lemmas is easy!

- if POB m is existentially quantified, then it's negation is universally quantified
- checking that Tr implies a universally quantified lemma is easy

$$\text{find } \ell \text{ s.t. } (F_i \land Tr \implies \ell') \land (\ell \implies \neg m)$$

But universal quantifiers make even basic SMT queries undecidable!

- cannot assume that SMT-solver will magically handle this for us

QUIC3: Quantified IC3 [kwik-ee]

Spacer extends IC3/PDR from Propositional logic to LIA + Arrays

Quic3 extends Spacer to discovering Universally Quantified solutions

- Extend proof obligations with free (implicitly existentially quantified) variables
- Allow universal quantifiers in lemmas
- Explicitly manage quantifier instantiations to guarantee progress
  - **without** syntactic restriction of formulas (e.g., MBQI, Local Theory, APF)
  - **without** user-specified patterns
- Quantified generalization to heuristically infer new quantifiers

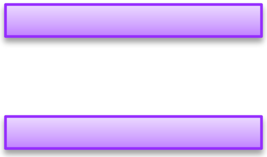
Implemented in spacer in Z3 master branch

- `z3 fp.spacer.ground_pobs=false fp.spacer.q3.use_qgen=true NAME.smt2`

QUIC3: Trace and Proof Obligations

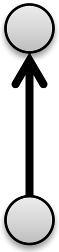
A quantified trace Q = $Q_0, \ldots, Q_N$ is a sequence of frames.

- A frame Q_i is a set of (ℓ, σ), where ℓ is a lemma and σ a substitution.
- $qi(Q) = \{\ell\sigma \mid (\ell, \sigma) \in Q\}$ $\qquad$ $\forall Q = \{\forall \ell \mid (\ell, \sigma) \in Q\}$
- Invariants:
  - Bounded Safety: $\forall\ i < N\ .\ \forall Q_i \rightarrow \neg Bad$
  - Monotonicity: $\forall\ i < N\ .\ \forall Q_{i+1} \subseteq \forall Q_i$
  - Inductiveness: $\forall\ i < N\ .\ \forall Q_i \wedge Tr \rightarrow \forall Q'_{i+1}$



A priority queue $\mathcal{Q}$ of quantified proof obligations (POBs)

- $(m, \xi, i) \in \mathcal{Q}$ where m is a cube, ξ is a ground substitution for all free variables of m, and i is a numeric level
- if $(m, \xi, i) \in \mathcal{Q}$ then there exists a path of length (N-i) from a state in $m\xi$ to a state in Bad



UNIVERSITY OF WATERLOO

QUIC3: Rules

Input: A safety problem $\langle \mathit{Init}(X), \mathit{Tr}(X,X'), \mathit{Bad}(X)\rangle$.
Assumptions: *Init*, *Tr* and *Bad* are quantifier free.
Data: A POB queue $\mathcal{Q}$, where a POB $c \in \mathcal{Q}$ is a triple $\langle m, \sigma, i\rangle$, m is a conjunction of literals over X and free variables, σ is a substitution s.t. $m\sigma$ is ground, and $i \in \mathbb{N}$. A level N. A quantified trace $\mathcal{T} = Q_0, Q_1, \ldots$, where for every pair $(\ell, \sigma) \in Q_i$, ℓ is a quantifier-free formula over X and free variables and σ a substitution s.t. $\ell\sigma$ is ground.
Notation: $\mathcal{F}(A) = (A(X) \wedge \mathit{Tr}(X,X')) \vee \mathit{Init}(X')$; $qi(Q) = \{\ell\sigma \mid (\ell,\sigma) \in Q\}$; $\forall Q = \{\forall \ell \mid (\ell,\sigma) \in Q\}$.

Output: *Safe* or *Cex*
Initially: $\mathcal{Q} = \emptyset$, $N = 0$, $Q_0 = \{(\mathit{Init}, \emptyset)\}$, $\forall i > 0 \cdot Q_i = \emptyset$.
repeat

- **Safe** If there is an $i < N$ s.t. $\forall Q_i \subseteq \forall Q_{i+1}$ **return** *Safe*.
- **Cex** If there is an m, σ s.t. $\langle m, \sigma, 0\rangle \in \mathcal{Q}$ **return** *Cex*.
- **Unfold** If $qi(Q_N) \rightarrow \neg \mathit{Bad}$, then set $N \leftarrow N + 1$.
- **Candidate** If for some m, $m \rightarrow qi(Q_N) \wedge \mathit{Bad}$, then add $\langle m, \emptyset, N\rangle$ to $\mathcal{Q}$.
- **Predecessor** If $\langle m, \xi, i+1\rangle \in \mathcal{Q}$ and there is a model M s.t. $M \models qi(Q_i) \wedge \mathit{Tr} \wedge (m'_{sk})$, add $\langle \psi, \sigma, i\rangle$ to $\mathcal{Q}$, where $(\psi, \sigma) = abs(U, \varphi)$ and $(\varphi, U) = \textsc{pMbp}(X' \cup SK, \mathit{Tr} \wedge m'_{sk}, M)$.
- **NewLemma** For $0 \leq i < N$, given a POB $\langle m, \sigma, i+1\rangle \in \mathcal{Q}$ s.t. $\mathcal{F}(qi(Q_i)) \wedge m'_{sk}$ is unsatisfiable, and $L' = \textsc{Itp}(\mathcal{F}(qi(Q_i)), m'_{sk})$, add (ℓ, σ) to Q_j for $j \leq i + 1$, where $(\ell, \_) = abs(SK, L)$.
- **Push** For $0 \leq i < N$ and $((\varphi \vee \psi), \sigma) \in Q_i$, if $(\varphi, \sigma) \notin Q_{i+1}$, $\mathit{Init} \rightarrow \forall \varphi$ and $(\forall \varphi) \wedge \forall Q_i \wedge qi(Q_i) \wedge \mathit{Tr} \rightarrow \forall \varphi'$, then add (φ, σ) to Q_j, for all $j \leq i + 1$.

until ∞;

UNIVERSITY OF WATERLOO

QUIC3: Predecessor, NewLemma, and Push

repeat

⋮

Predecessor If $\langle m, \xi, i+1 \rangle \in \mathcal{Q}$ and there is a model M s.t. $M \models qi(Q_i) \wedge Tr \wedge (m'_{sk})$, add $\langle \psi, \sigma, i \rangle$ to $\mathcal{Q}$, where $(\psi, \sigma) = abs(U, \varphi)$ and $(\varphi, U) = \textsc{pMbp}(X' \cup SK, Tr \wedge m'_{sk}, M)$.

NewLemma For $0 \leq i < N$, given a POB $\langle m, \sigma, i+1 \rangle \in \mathcal{Q}$ s.t. $qi(Q_i) \wedge Tr \wedge m'_{sk}$ is unsatisfiable, and $L' = \textsc{Itp}(\mathcal{F}(qi(Q_i)), m'_{sk})$, add (ℓ, σ) to Q_j for $j \leq i+1$, where $(\ell, \_) = abs(SK, L)$.

Push For $0 \leq i < N$ and $((\varphi \vee \psi), \sigma) \in Q_i$, if $(\varphi, \sigma) \notin Q_{i+1}$, $Init \to \forall \varphi$ and $(\forall \varphi) \wedge \forall Q_i \wedge qi(Q_i) \wedge Tr \to \forall \varphi'$, then add (φ, σ) to Q_j, for all $j \leq i+1$.

until ∞;

In **Predecessor** and **NewLemma** only use current instantiations of quantified lemmas. All SMT queries are quantifier free

In **Push**, quantified lemmas are required for relative completeness

- in practice, we use incomplete pattern-based instantiation and hope that it is sufficient together with $qi(Q_i)$

Progress and Counterexamples

The **Predecessor** rule is only finitely applicable to any POB

- follows from how quantified terms are abstracted by free variables and how quantified lemmas are instantiated
- assumes that Skolemization is deterministic
- uses finiteness of Model Based Projection

MkSafe in Quic3 is terminating for any given bound N

- w.l.o.g, assume Bad is a single POB
- Follows by induction on the bound N

MkSafe in Quic3 computes a quantified interpolation sequence

If there is a counterexample, Quic3 will terminate with the shortest counterexample

In SMT-LIB

```
(set-logic HORN)

;; Inv(A, N, i)
(declare-fun Inv ( (Array Int Int) Int Int ) Bool)

(assert
 (forall ( (A (Array Int Int)) (N Int) (C Int)) (Inv A N 0)))

(assert
 (forall ( (A (Array Int Int)) (N Int) (i Int) )
        (=>
         (and (Inv A N i) (< i N) )
         (Inv (store A i 0) N (+ i 1))
         )
        )
 )
(assert
 (forall ( (A (Array Int Int)) (N Int) (i Int) (j Int))
        (=> (and (Inv A N i )
                (>= i N) (<= 0 j) (< j N) (not (= (select A
j) 0)))
           false
           )
        )
 )

(check-sat)
(get-model)
```

```
$ z3 array-zero.smt2

sat
(model
  (define-fun Inv ((x!0 (Array Int Int)) (x!1 Int) (x!2 Int)) Bool
    (let ((a!1 (forall ((sk!0 Int))
                 (! (or (not (>= sk!0 0))
                        (>= (select x!0 sk!0) 0)
                        (<= (+ x!2 (* (- 1) sk!0)) 0))
                    :weight 15)))
          (a!2 (forall ((sk!0 Int))
                 (! (or (not (>= sk!0 0))
                        (<= (select x!0 sk!0) 0)
                        (<= (+ x!2 (* (- 1) sk!0)) 0))
                    :weight 15))))
      (and a!1 a!2)))
)
```

UNIVERSITY OF WATERLOO

almost …

THE END

HORN(ALIA): Arrays + LIA

```
int A[N];
for (int i = 0; i < N; ++i)
   A[i] = 0;
for (i = 0; i < N; ++i)
  assert(A[i] == 0);
```

IS SAT?

```
Inv1(A, N, 0)
Inv1(A, N, i) & i < N → Inv1(A[i := 0], N, i+1)
Inv1(A, N, i) & i >= N → Inv2(A, N, 0)
Inv2(A, N, i) & i < N & A[i] = 0 → Inv2(A, N, i+1)
Inv2(A, N, i) & i < N & A[i] != 0 → false
```

In SMT-LIB

```
(set-logic HORN)

;; Inv(A, N, i)
(declare-fun Inv1 ( (Array Int Int) Int Int ) Bool)
(declare-fun Inv2 ( (Array Int Int) Int Int ) Bool)

(assert
 (forall ( (A (Array Int Int)) (N Int) (C Int)) (Inv1 A N 0)))

(assert
 (forall ( (A (Array Int Int)) (N Int) (i Int) )
        (=>
         (and (Inv1 A N i) (< i N) )
         (Inv1 (store A i 0) N (+ i 1))
         )
        )
 )
(assert
 (forall ( (A (Array Int Int)) (N Int) (i Int) )
        (=>
         (and (Inv1 A N i) (>= i N) ) (Inv2 A N 0)
        )
 ))

(assert
 (forall ( (A (Array Int Int)) (N Int) (i Int) )
        (=>
         (and (Inv2 A N i) (< i N) (= (select A i) 0) ) (Inv2 A N (+ i 1))
        )
 ))

(assert
 (forall ( (A (Array Int Int)) (N Int) (i Int) )
        (=>
         (and (Inv2 A N i) (< i N) (not (= (select A i) 0)) ) false
        )
 ))



(check-sat)
(get-model)
```

```
$ z3 -t:100 array-zero2.smt2
canceled
unknown
```

Why this example diverges?

```
Inv2(A, N, i) & i < N & A[i] != 0 → false
```

$i < N \wedge A[i] \neq 0$ ← true

```
Inv1(A, N, i) & i >= N → Inv2(A, N, 0)
```

$0 < N \leq i \wedge A[0] \neq 0$ ← $i < N \wedge A[i] \neq 0$

```
Inv2(A, N, i) & i < N & A[i] = 0 → Inv2(A, N, i+1)
```

$i + 1 < B \wedge A[i] = 0 \wedge A[i+1] \neq 0$ ← $i < N \wedge A[i] \neq 0$

```
Inv1(A, N, i) & i >= N → Inv2(A, N, 0)
```

$1 < N \leq i \wedge A[0] = 0 \wedge A[1] \neq 0$ ← $i + 1 < B \wedge A[i] = 0 \wedge A[i+1] \neq 0$

Quantified Generalizer

"… to boldly go where no one has gone before" (but many have been)

$$1 < N \leq i \wedge A[0] = 0 \wedge A[1] \neq 0$$

Quantified generalizer is a heuristic to generalize POBs using existential quantifiers

- e.g., in our example, we want to generalize the pob into

$$\exists j \cdot 1 < N \leq i \wedge 0 \leq j < N \wedge A[j] \neq 0$$

We look for a pattern in the formula (anti-unification)

Use convex closure (i.e., abstract join) to capture the pattern by a conjunction

Apply **after** pob is blocked and generalized

- As any generalization, it is a *dark* magic

In SMT-LIB

```
(set-logic HORN)

;; Inv(A, N, i)
(declare-fun Inv1 ( (Array Int Int) Int Int ) Bool)
(declare-fun Inv2 ( (Array Int Int) Int Int ) Bool)

(assert
 (forall ( (A (Array Int Int)) (N Int) (C Int)) (Inv1 A N 0)))

(assert
 (forall ( (A (Array Int Int)) (N Int) (i Int) )
        (=>
         (and (Inv1 A N i) (< i N) )
         (Inv1 (store A i 0) N (+ i 1))
         )
        )
 )
(assert
 (forall ( (A (Array Int Int)) (N Int) (i Int) )
        (=>
         (and (Inv1 A N i) (>= i N) ) (Inv2 A N 0)
        )
 ))

(assert
 (forall ( (A (Array Int Int)) (N Int) (i Int) )
        (=>
         (and (Inv2 A N i) (< i N) (= (select A i) 0) ) (Inv2 A N (+ i 1))
        )
 ))

(assert
 (forall ( (A (Array Int Int)) (N Int) (i Int) )
        (=>
         (and (Inv2 A N i) (< i N) (not (= (select A i) 0)) ) false
        )
 ))



(check-sat)
(get-model)
```

```
$ z3 array-zero2.smt2

sat
(model
  (define-fun Inv2 ((x!0 (Array Int Int)) (x!1 Int) (x!2 Int)) Bool
    (let ((a!1 (forall ((sk!0 Int))
                 (! (or (<= (+ x!1 (* (- 1) sk!0)) 0)
                        (<= (select x!0 sk!0) 0)
                        (<= (+ sk!0 (* (- 1) x!2)) 0))
                    :weight 15)))
          (a!2 (or (<= (+ x!1 (* (- 1) x!2)) 0) (<= (select x!0 x!2) 0)))
          (a!3 (or (>= (select x!0 x!2) 0) (<= (+ x!1 (* (- 1) x!2)) 0)))
          (a!4 (forall ((sk!0 Int))
                 (! (or (<= (+ x!1 (* (- 1) sk!0)) 0)
                        (>= (select x!0 sk!0) 0)
                        (<= (+ sk!0 (* (- 1) x!2)) 0))
                    :weight 15))))
      (and a!1 a!2 a!3 a!4)))
  (define-fun Inv1 ((x!0 (Array Int Int)) (x!1 Int) (x!2 Int)) Bool
    (let ((a!1 (forall ((sk!0 Int))
                 (! (or (<= (select x!0 sk!0) 0)
                        (<= (+ x!2 (* (- 1) sk!0)) 0)
                        (<= sk!0 0))
                    :weight 15)))
          (a!2 (forall ((sk!0 Int))
                 (! (let ((a!1 (>= (+ sk!0 (* (- 1) (select x!0 sk!0))) 0)))
                      (or (not (>= sk!0 0)) (<= (+ x!2 (* (- 1) sk!0)) 0) a!1))
                    :weight 15)))
          (a!3 (forall ((sk!0 Int))
                 (! (or (<= (+ x!2 (* (- 1) sk!0)) 0)
                        (>= (select x!0 sk!0) 0)
                        (<= sk!0 0))
                    :weight 15))))
      (and a!1 a!2 (or (>= (select x!0 0) 0) (<= x!2 0)) a!3)))
)
```

DEMO

Related Work

Predicate Abstraction

- extend predicates with fresh universally quantified variables
- relies on a decision procedure for quantified logic

Model-Checking Modulo Theories (MCMT)

- model checking of array manipulating programs
- supported by multiple tools: cubicle, mcmt, safari, …
- quantifier elimination to compute predecessors
- requires checking satisfiability of quantified formulas for sub-sumption and convergence

Discovery of Universal Invariants with Abstract Interpretation

- compute universally quantified inductive invariants of a certain shape
- often specialized for reasoning about arrays in programming languages
- not property directed, no guarantees, but often very quick
- can be combined with Quic3 as pre-processing

Most Closely Related Work

Safari and Booster

- extends Lazy Abstraction with Interpolants (LAWI) to array manipulating programs
- solves mkSafe() using quantifier free theory of arrays and computes ***quantifier free*** sequence interpolant
- heuristically guesses quantified lemmas by abstracting terms
- see Avy for in-depth comparison between interpolation and IC3

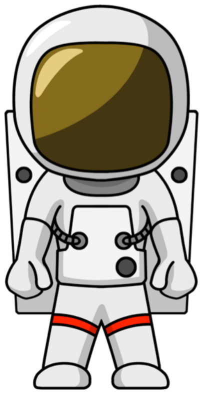
Transformation into non-linear CHC

- guess number of quantifiers and instances statically
- use quantifier-free **non-linear** CHC solver to find template invariant
- generalizes most Abstract Interpretation / Template-based approaches
- cannot discover counterexamples
- can be simulated in Quic3 by restricting instantiations used

UPDR

- existential pobs and universal lemmas over decidable theories

Conclusion



Quic3 brings reasoning about quantified invariants to CHC

- Implemented in spacer
- can discover non-trivial quantified invariants of complex code

Guarantee progress and counterexamples

- don't get stuck with a quantified SMT query
- find shortest counterexample

Many open questions remain

- strides – memory is traversed in a stride (e.g., x=x+4)
- additional quantified generalizers (speed vs precision)
- Enumerating invariants in a decidable fragment (EssenUF, APF, etc.)

CHC-COMP: CHC Solving Competition

First edition on July 13, 2018 at HVCS@FLOC

Constrained Horn Clauses (CHC) is a fragment of First Order Logic (FOL) that is sufficiently expressive to describe many verification, inference, and synthesis problems including inductive invariant inference, model checking of safety properties, inference of procedure summaries, regression verification, and sequential equivalence. The CHC competition (CHC-COMP) will compare state-of-the-art tools for CHC solving with respect to performance and effectiveness on a set of publicly available benchmarks. The winners among participating solvers are recognized by measuring the number of correctly solved benchmarks as well as the runtime.

Web: https://chc-comp.github.io/

Gitter: https://gitter.im/chc-comp/Lobby

GitHub: https://github.com/chc-comp

Format: https://chc-comp.github.io/2018/format.html

CHC COMP