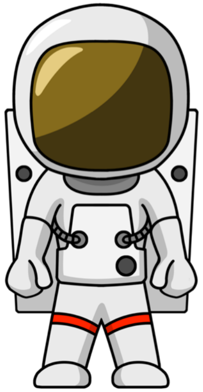
Quantified Solutions for Model Checking with Constrained Horn Clauses

Arie Gurfinkel

BeMC: The Best of Model Checking
July 13, 2019
New York, NY

UNIVERSITY OF WATERLOO

SPACER: The Final Frontier



joint work with Nikolaj Bjorner, Anvesh Komuraveli, Sharon Shoham, Yakir Vizel, Hari Govind, Yu-Ting (Jeff) Chen, …

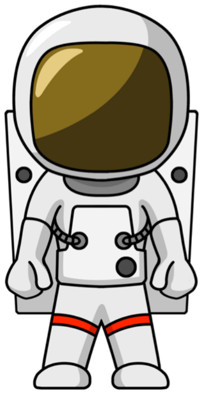
UNIVERSITY OF WATERLOO

Software Model Checking of
Programs / Transitions Systems /
Push-down Systems

=

Satisfiability of Constrained
Horn Logic (CHC) fragment of
First Order Logic

Reduce Model Checking to
FOL Satisfiability



UNIVERSITY OF WATERLOO

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, …, p_n, h are n-ary predicates
- $p_i[X]$ is an application of a predicate to first-order terms

UNIVERSITY OF WATERLOO

Horn Clauses for Program Verification

$\ell_{out}(x_0, w', e_o)$, which is an entry point into successor edges. with the edges are formulated as follows:

$$p_{init}(x_0, \boldsymbol{w}, \bot) \leftarrow x = x_0 \quad \text{where } x \text{ occurs in } \boldsymbol{w}$$
$$p_{exit}(x_0, ret, \top) \leftarrow \ell(x_0, \boldsymbol{w}, \top) \quad \text{for each label } \ell\text{, and re}$$
$$p(x, ret, \bot, \bot) \leftarrow p_{exit}(x, ret, \bot)$$
$$p(x, ret, \bot, \top) \leftarrow p_{exit}(x, ret, \top)$$
$$\ell_{out}(x_0, \boldsymbol{w}', e_o) \leftarrow \ell_{in}(x_0, \boldsymbol{w}, e_i) \wedge \neg e_i \wedge \neg wlp(S, \neg(e_i =$$

Weakest Preconditions If we apply Boogie directly we obtain a translation from programs to Horn logic using a weakest liberal pre-condition calculus [26]:

$$\mathsf{ToHorn}(program) := wlp(Main(), \top) \wedge \bigwedge_{decl \in program} \mathsf{ToHorn}(decl)$$
$$\mathsf{ToHorn}(\textbf{def } p(x)\ \{S\}) := wlp\begin{pmatrix}\textbf{havoc } x_0; \textbf{assume } x_0 = x; & \\ \textbf{assume } p_{pre}(x); S, & p(x_0, ret)\end{pmatrix}$$
$$wlp(x := E, Q) := \textbf{let } x = E \textbf{ in } Q$$
$$wlp((\textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2), Q) := wlp(((\textbf{assume } E; S_1)\Box(\textbf{assume } \neg E; S_2)), Q)$$
$$wlp((S_1 \Box S_2), Q) := wlp(S_1, Q) \wedge wlp(S_2, Q)$$
$$wlp(S_1; S_2, Q) := wlp(S_1, wlp(S_2, Q))$$
$$wlp(\textbf{havoc } x, Q) := \forall x\ .\ Q$$
$$wlp(\textbf{assert } \varphi, Q) := \varphi \wedge Q$$
$$wlp(\textbf{assume } \varphi, Q) := \varphi \rightarrow Q$$
$$wlp((\textbf{while } E \textbf{ do } S), Q) := inv(\boldsymbol{w}) \wedge \forall \boldsymbol{w}\ .\ \begin{pmatrix}((inv(\boldsymbol{w}) \wedge E) \rightarrow wlp(S, inv(\boldsymbol{w}))) \\ \wedge\ ((inv(\boldsymbol{w}) \wedge \neg E) \rightarrow Q)\end{pmatrix}$$

```
5. incorrect :- Z=W+1, W≥0, W+1<
                read(A,W,U), read(A,Z
6. p(I1,N,B) :- 1≤I, I<N, D=I−1, I1=I+1, V=U+1,
                read(A,D,U), write(A
7. p(I,N,A) :- I=1, N>1.
```

De Angelis et al. Verifying Array Programs by Transforming Verification Conditions. VMCAI'14

To translate a procedure call $\ell : y := q(E); \ell'$ within a procedure p, create he clauses:

$$p(\boldsymbol{w}_0, \boldsymbol{w}_4) \leftarrow p(\boldsymbol{w}_0, \boldsymbol{w}_1), call(\boldsymbol{w}_1, \boldsymbol{w}_2), q(\boldsymbol{w}_2, \boldsymbol{w}_3), return(\boldsymbol{w}_1, \boldsymbol{w}_3, \boldsymbol{w}_4)$$
$$q(\boldsymbol{w}_2, \boldsymbol{w}_2) \leftarrow p(\boldsymbol{w}_0, \boldsymbol{w}_1), call(\boldsymbol{w}_1, \boldsymbol{w}_2)$$
$$call(\boldsymbol{w}, \boldsymbol{w}') \leftarrow \pi = \ell, x' = E, \pi' = \ell_{q_{init}}$$
$$return(\boldsymbol{w}, \boldsymbol{w}', \boldsymbol{w}'') \leftarrow \pi' = \ell_{q_{exit}}, \boldsymbol{w}'' = \boldsymbol{w}[ret'/y, \ell'/\pi]$$

Bjørner, Gurfinkel, McMillan, and Rybalchenko:

Horn Clause Solvers for Program Verification

UNIVERSITY OF WATERLOO

Horn Clauses for Concurrent / Distributed / Parameterized Systems

For assertions $R_1, \ldots, R_N$ over V and $E_1, \ldots, E_N$ over V, V',

$$
\begin{array}{lll}
\text{CM1}: & init(V) & \rightarrow R_i(V) \\
\text{CM2}: & R_i(V) \wedge \rho_i(V,V') & \rightarrow R_i(V') \\
\text{CM3}: & (\bigvee_{i \in 1..N \setminus \{j\}} R_i(V) \wedge \rho_i(V,V')) & \rightarrow E_j(V,V') \\
\text{CM4}: & R_i(V) \wedge E_i(V,V') \wedge \rho_i^{=}(V,V') & \rightarrow R_i(V') \\
\text{CM5}: & R_1(V) \wedge \cdots \wedge R_N(V) \wedge error(V) & \rightarrow false \\
\hline
 & \text{multi-threaded program } P \text{ is safe} &
\end{array}
$$

Rybalchenko et al. Synthesizing Software Verifiers from Proof Rules. PLDI'12

$$\left\{ R(\mathsf{g}, \mathsf{p}_{\sigma(1)}, \mathsf{l}_{\sigma(1)}, \ldots, \mathsf{p}_{\sigma(k)}, \mathsf{l}_{\sigma(k)}) \leftarrow dist(\mathsf{p}_1, \ldots, \mathsf{p}_k) \wedge R(\mathsf{g}, \mathsf{p}_1, \mathsf{l}_1, \ldots, \mathsf{p}_k, \mathsf{l}_k) \right\}_{\sigma \in S_k} \quad (6)$$

$$R(\mathsf{g}, \mathsf{p}_1, \mathsf{l}_1, \ldots, \mathsf{p}_k, \mathsf{l}_k) \leftarrow dist(\mathsf{p}_1, \ldots, \mathsf{p}_k) \wedge Init(\mathsf{g}, \mathsf{l}_1) \wedge \cdots \wedge Init(\mathsf{g}, \mathsf{l}_k) \quad (7)$$

$$R(\mathsf{g}', \mathsf{p}_1, \mathsf{l}_1', \ldots, \mathsf{p}_k, \mathsf{l}_k) \leftarrow dist(\mathsf{p}_1, \ldots, \mathsf{p}_k) \wedge ((\mathsf{g}, \mathsf{l}_1) \xrightarrow{\mathsf{p}_1} (\mathsf{g}', \mathsf{l}_1')) \wedge R(\mathsf{g}, \mathsf{p}_1, \mathsf{l}_1, \ldots, \mathsf{p}_k, \mathsf{l}_k) \quad (8)$$

$$R(\mathsf{g}', \mathsf{p}_1, \mathsf{l}_1, \ldots, \mathsf{p}_k, \mathsf{l}_k) \leftarrow dist(\mathsf{p}_0, \mathsf{p}_1, \ldots, \mathsf{p}_k) \wedge ((\mathsf{g}, \mathsf{l}_0) \xrightarrow{\mathsf{p}_0} (\mathsf{g}', \mathsf{l}_0')) \wedge RConj(0, \ldots, k) \quad (9)$$

$$false \leftarrow dist(\mathsf{p}_1, \ldots, \mathsf{p}_r) \wedge \Big(\bigwedge_{j=1,\ldots,m} (\mathsf{p}_j = p_j \wedge (\mathsf{g}, \mathsf{l}_j) \in E_j) \Big) \wedge RConj(1, \ldots, r) \quad (10)$$

Figure 4: Horn constraints encoding a homogeneous infinite system with the help of a k-indexed invariant. S_k is the symmetric group on $\{1, \ldots, k\}$, i.e., the group of all permutations of k numbers; as an optimisation, any generating subset of S_k, for instance transpositions, can be used instead of S_k. In (10), we define $r = \max\{m, k\}$.

Hojjat et al. Horn Clauses for Communicating Timed Systems. HCVS'14

$$Init(i,j,\overline{v}) \wedge Init(j,i,\overline{v}) \wedge Init(i,i,\overline{v}) \wedge Init(j,j,\overline{v}) \Rightarrow I_2(i,j,\overline{v})$$

$$I_2(i,j,\overline{v}) \wedge Tr(i,\overline{v},\overline{v}') \Rightarrow I_2(i,j,\overline{v}') \quad (3)$$

$$I_2(i,j,\overline{v}) \wedge Tr(j,\overline{v},\overline{v}') \Rightarrow I_2(i,j,\overline{v}') \quad (4)$$

$$I_2(i,j,\overline{v}) \wedge I_2(i,k,\overline{v}) \wedge I_2(j,k,\overline{v}) \wedge Tr(k,\overline{v},\overline{v}') \wedge k \neq i \wedge k \neq j \Rightarrow I_2(i,j,\overline{v}') \quad (5)$$

$$I_2(i,j,\overline{v}) \Rightarrow \neg Bad(i,j,\overline{v})$$

Figure 3: $VC_2(T)$ for two-quantifier invariants.

Gurfinkel et al. SMT-Based Verification of Parameterized Systems. FSE 2016

(initial) $\mathsf{init}(g, x_1) \wedge \cdots \wedge \mathsf{init}(g, x_n) \rightarrow Inv(g, \ell_{\mathsf{init}}, x_1, \ldots, \ell_{\mathsf{init}}, x_k)$

(inductive) $Inv(g, \ell_1, x_1, \ldots, \ell_i, x_i, \ldots, \ell_k, x_k) \wedge s(g, x_i, g', x_i') \rightarrow Inv(g', \ell_1, x_1, \ldots, \ell_i', x_i', \ldots, \ell_k,$

(non-interference) $Inv(g, \ell_1, x_1, \ldots, \ell_k, x_k) \wedge$
$Inv(g, \ell^\dagger, x^\dagger, \ell_2, x_2, \ldots, \ell_k, x_k) \wedge$
$\vdots$
$Inv(g, \ell_1, x_1, \ldots, \ell_{k-1}, x_{k-1}, \ell^\dagger, x^\dagger) \wedge s(g, x^\dagger, g', \cdot) \rightarrow Inv(g', \ell_1, x_1, \ldots, \ell_k, x_k)$

(safe) $Inv(g, \ell_1, x_1, \ldots, \ell_k, x_k) \wedge \mathsf{err}(g, \ell_1, x_1, \ldots, \ell_m, x_m) \rightarrow false$

Figure 6. Horn clause encoding for thread modularity at level k (where (ℓ_i, s, ℓ_i') and $(\ell^\dagger, s, \cdot)$ refer to statement s on a from ℓ_i to ℓ_i' and, respectively, from $\ell^\dagger$ to some other location in the control flow graph)

Hoenicke et al. Thread Modularity at Many Levels. POPL'17

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

UNIVERSITY OF WATERLOO

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

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);
```

IS SAT?

```
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
```

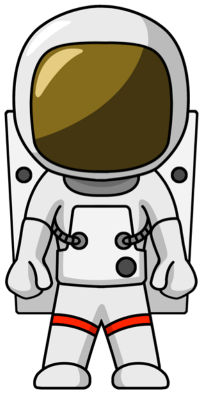
In SMT-LIB

```
(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
```

Spacer: Solving SMT-constrained CHC



Spacer: SAT procedure for SMT-constrained Horn Clauses

- now the default 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
- 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.

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
```

UNIVERSITY OF WATERLOO

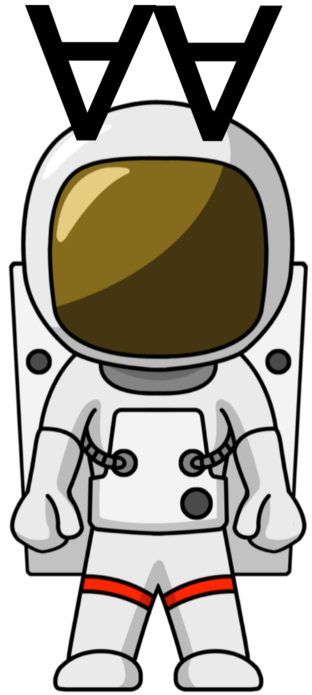
In SMT-LIB

```
(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)

$\forall\ 0 <= j < i < N \rightarrow A[j] = 0$



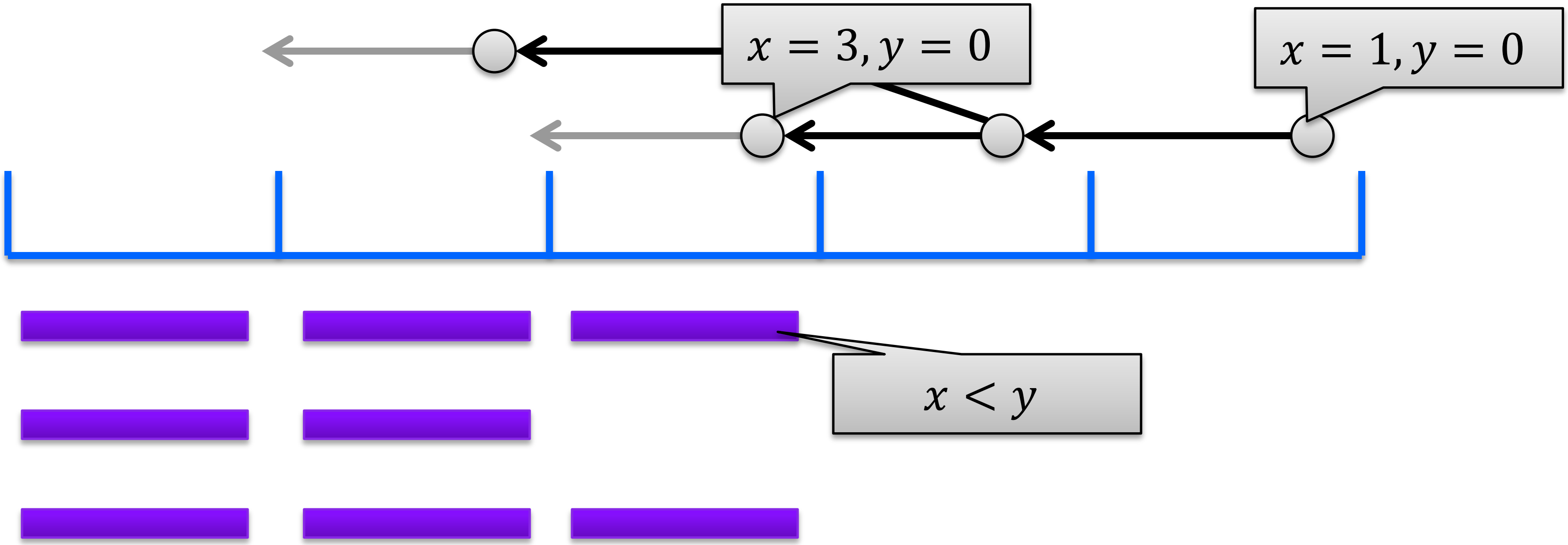
Extends Spacer with reasoning about quantified solutions

QUIC3: QUANTIFIED IC3

Arie Gurfinkel, Sharon Shoham, Yakir Vizel: Quantifiers on Demand. ATVA 2018

IC3/PDR In Pictures: MkSafe

MkSafe



Predecessor

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

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

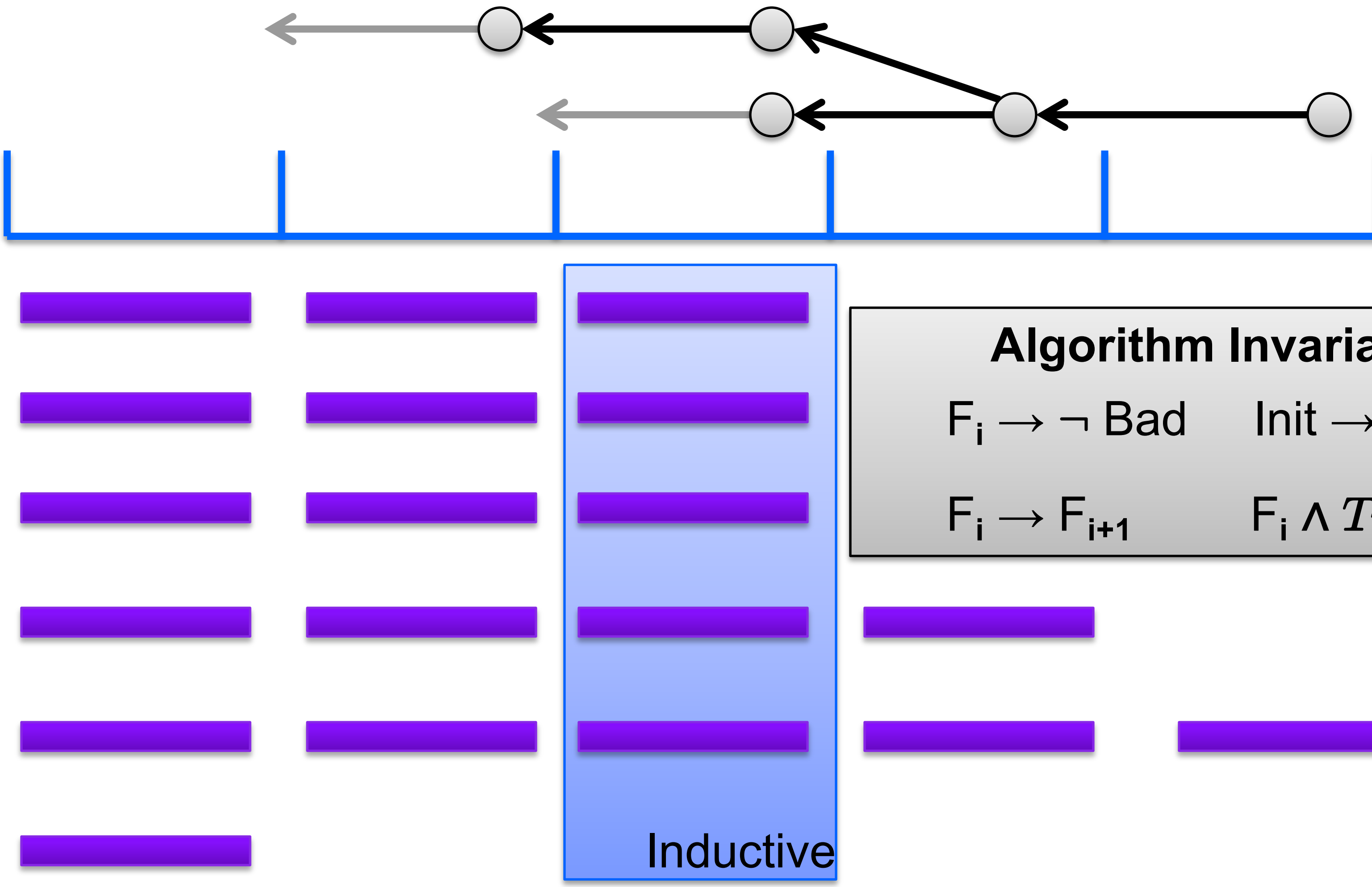
NewLemma

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

UNIVERSITY OF WATERLOO

IC3/PDR in Pictures: Push

Push



Algorithm Invariants

$F_i \rightarrow \neg Bad$ $\quad$ $Init \rightarrow F_i$

$F_i \rightarrow F_{i+1}$ $\quad$ $F_i \wedge Tr \rightarrow F_{i+1}$

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

UNIVERSITY OF WATERLOO

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

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

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

$$= \quad ???$$

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

Quantified POBs and Lemmas

Must deal with existentially quantified POBs

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

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

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

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

But universal quantifiers make even basic SMT queries undecidable!

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

UNIVERSITY OF WATERLOO

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

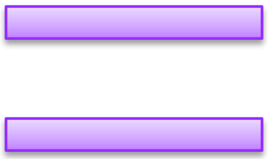
- `z3 fp.spacer.ground_pobs=false fp.spacer.q3.use_qgen=true NAME.smt2`

UNIVERSITY OF WATERLOO

QUIC3: Trace and Proof Obligations

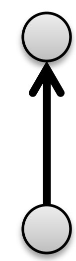
A quantified trace $Q = Q_0, \dots, Q_N$ is a sequence of frames.

- A frame Q_i is a set of (ℓ, σ), where ℓ is a lemma and σ a substitution.
- $qi(Q) = \{\ell\sigma \mid (\ell, \sigma) \in Q\}$ $\quad\quad$ $\forall Q = \{\forall \ell \mid (\ell, \sigma) \in Q\}$
- 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: $\mathcal{F}(A) = (A(X) \wedge \mathit{Tr}(X, X')) \vee \mathit{Init}(X')$; $\mathit{qi}(Q) = \{\ell\sigma \mid (\ell, \sigma) \in Q\}$; $\forall Q = \{\forall \ell \mid (\ell, \sigma) \in Q\}$.

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 $\mathit{qi}(Q_N) \to \neg \mathit{Bad}$, then set $N \leftarrow N + 1$.

Candidate If for some m, $m \to \mathit{qi}(Q_N) \wedge \mathit{Bad}$, 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. $M \models \mathit{qi}(Q_i) \wedge \mathit{Tr} \wedge (m'_{sk})$, add $\langle \psi, \sigma, i\rangle$ to $\mathcal{Q}$, where $(\psi, \sigma) = \mathit{abs}(U, \varphi)$ and $(\varphi, U) = \textsc{pMbp}(X' \cup \mathit{SK}, \mathit{Tr} \wedge m'_{sk}, M)$.

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

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

until ∞;

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. $M \models qi(Q_i) \wedge \mathit{Tr} \wedge (m'_{sk})$, 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. $qi(Q_i) \wedge \mathit{Tr} \wedge m'_{sk}$ 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 $(\varphi, \sigma) \notin Q_{i+1}$, $\mathit{Init} \to \forall \varphi$ and $(\forall \varphi) \wedge \forall Q_i \wedge qi(Q_i) \wedge \mathit{Tr} \to \forall \varphi'$, 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)))
)
```

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
```

$$i < N \wedge A[i] \neq 0 \longleftarrow \text{true}$$

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

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

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

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

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

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

Quantified Generalizer

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

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

Quantified generalizer is a heuristic to generalize POBs using existential quantifiers

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

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

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)))
)
```

UNIVERSITY OF WATERLOO

THE CURSE OF INTERPOLATION

UNIVERSITY OF WATERLOO

current work with Hari Govind and Yu-Ting (Jeff) Chen

The Curse of Interpolation

Interpolation is capable of generating many interesting terms

- (almost) any inductive invariant is an interpolant of something under the right conditions!

Interpolation often works in practice

- creates false sense of security
- predicate / term generation is a solved problem

But, interpolation is very hard to control!

- Small changes to input result in big change in interpolants
- Small changes to solver parameter result in big change in interpolants
- Works well overall (i.e., large benchmark set), but poorly for any given user problem!

GitHub, Inc. [US] | https://github.com/Z3Prover/z3/issues/2278

Apps · Getting Started · Google Bookmark · Add to Wish List · + Pocket · Google Bookmark · Application Funda... » Other Bookmarks

Search or jump to... / Pull requests Issues Marketplace Explore

Z3Prover / z3

Unwatch 169 · Star 4,193 · Fork 716

Code · Issues 136 · Pull requests 8 · Projects 0 · Wiki · Insights

Spacer's difficulty in finding linear invariants #2278

New issue

Open **shiatsumat** opened this issue 4 days ago · 1 comment

shiatsumat commented 4 days ago

It seems that Spacer is still not very good at finding linear invariants.
Are there any configurations on Spacer to enhance invariant finding?

Example:

```
(declare-rel loop (Int Int Int Int))
(rule (forall ((i Int) (x Int) (n Int) (r Int)) (=> (< i n) (loop (+ i 1) (+ x 1) n r) (loop i x n r
(rule (forall ((i Int) (x Int) (n Int) (r Int)) (=> (>= i n) (= r x) (loop i x n r))))
(declare-rel q ())
(rule (forall ((x Int) (n Int) (r Int)) (=> (loop 0 x n r) (>= n 0) (distinct r (+ x n)) q)))
; this is ok (rule (forall ((i Int) (x Int) (n Int) (r Int)) (=> (loop i x n r) (>= n i) (distinct r
(query q) ; timeout
```

Assignees
No one assigned

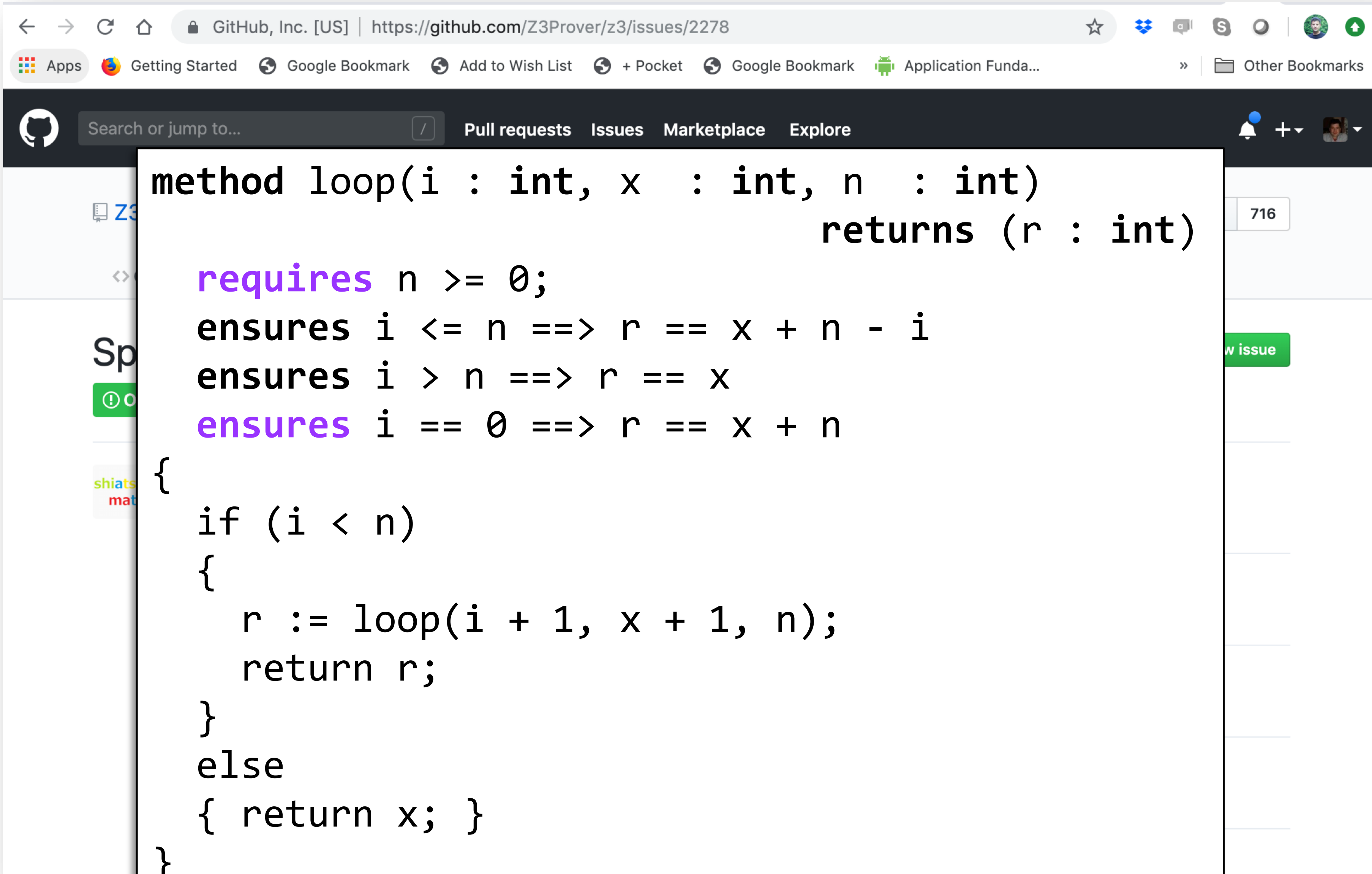
Labels
None yet

Projects
None yet

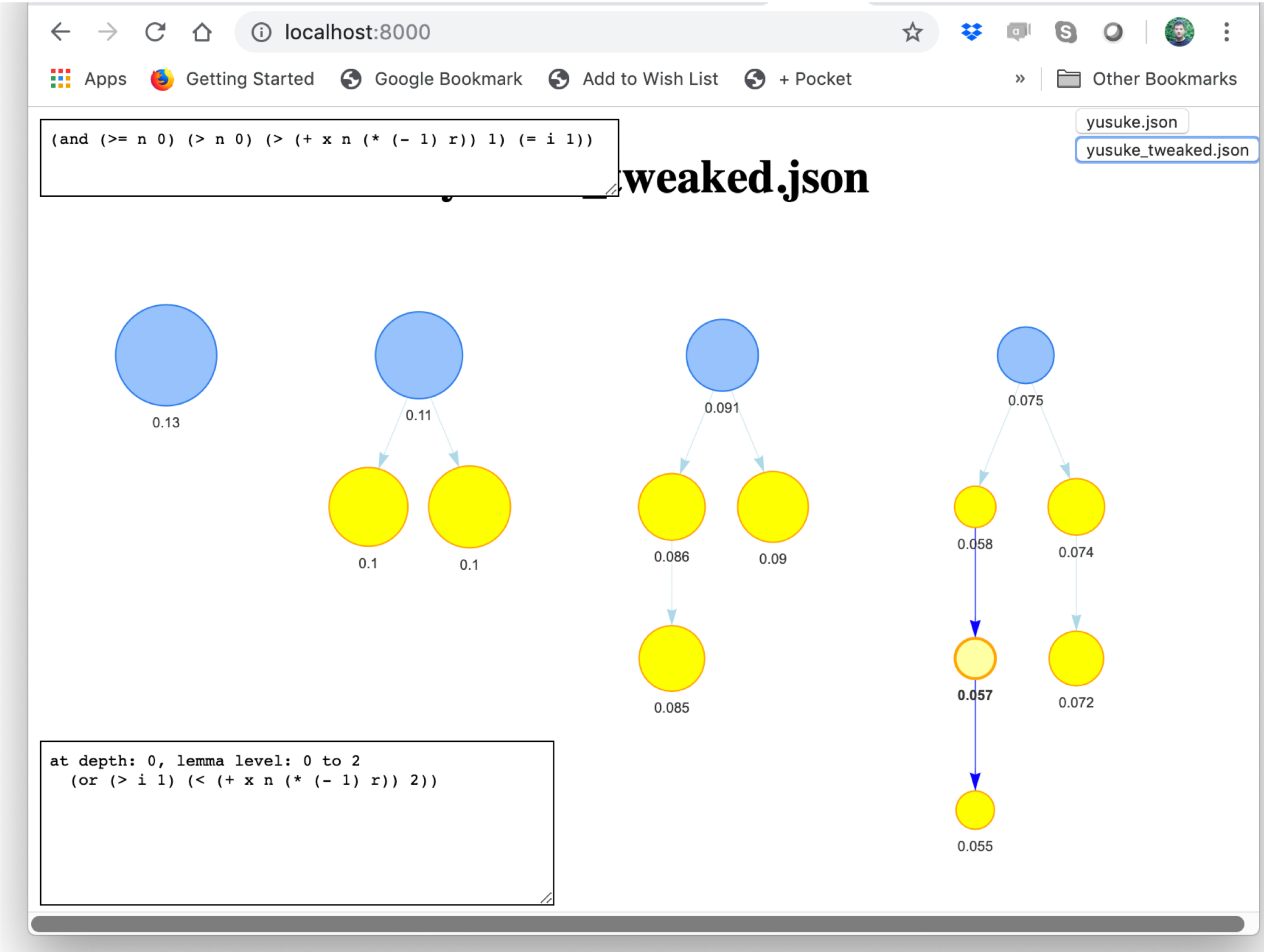
Milestone
No milestone

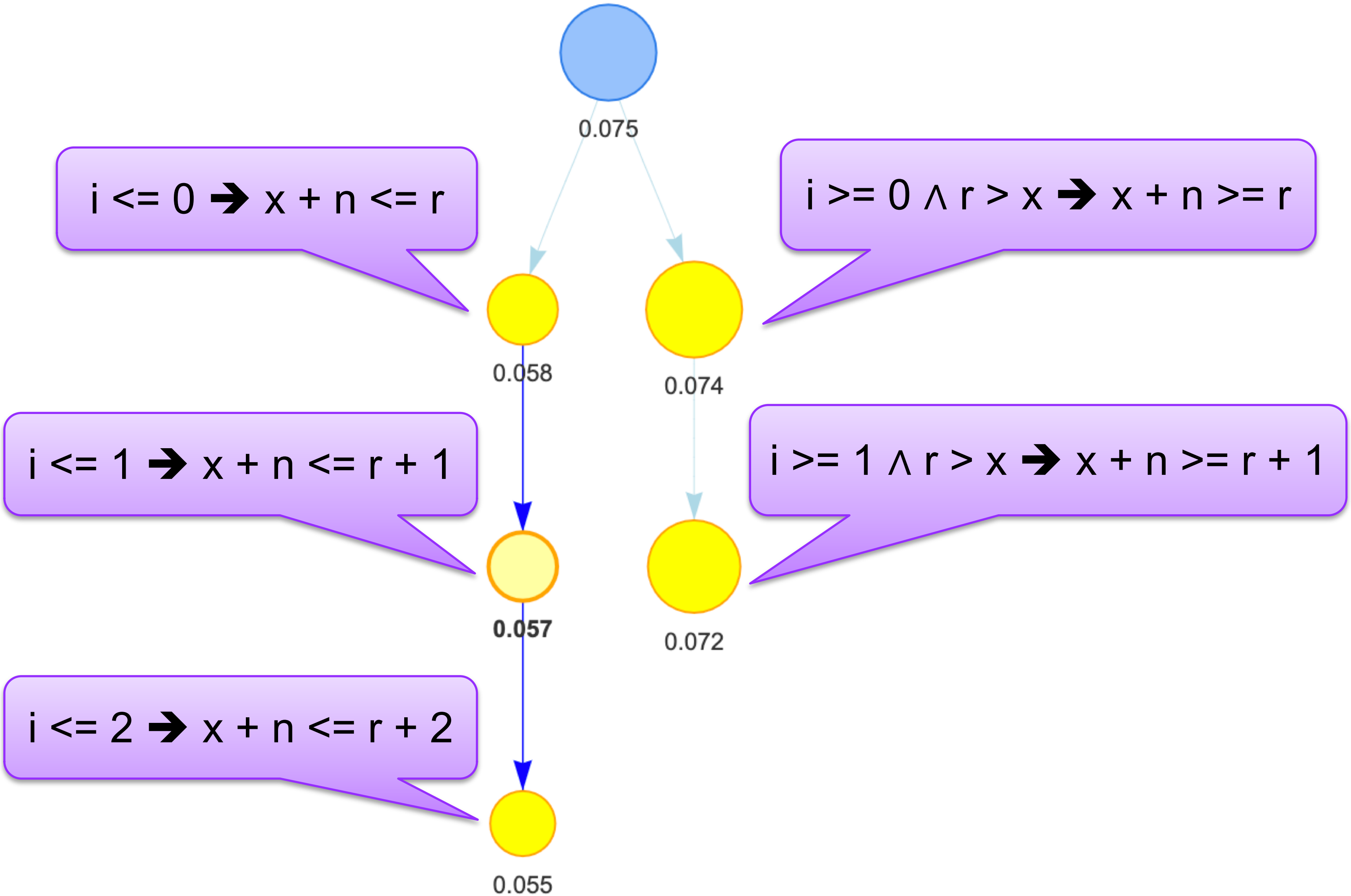
Notifications

UNIVERSITY OF WATERLOO




```
method loop(i : int, x  : int, n  : int)
                              returns (r : int)
  requires n >= 0;
  ensures i <= n ==> r == x + n - i
  ensures i > n ==> r == x
  ensures i == 0 ==> r == x + n
{
  if (i < n)
  {
    r := loop(i + 1, x + 1, n);
    return r;
  }
  else
  { return x; }
}
```


localhost:8000
Apps
Getting Started
Google Bookmark
Add to Wish List
+ Pocket
Other Bookmarks
yusuke.json
yusuke_tweaked.json
(and (>= n 0) (> n 0) (> (+ x n (* (- 1) r)) 1) (= i 1))
weaked.json
0.13
0.11
0.1
0.1
0.091
0.086
0.09
0.085
0.075
0.058
0.074
0.057
0.072
0.055
at depth: 0, lemma level: 0 to 2
(or (> i 1) (< (+ x n (* (- 1) r)) 2))



UNIVERSITY OF WATERLOO

Data Driven Generalization & Lemma Discovery

Global view of the current solver state

- **group** lemmas (and pobs) based on syntactic/semantic similarity
  - we currently use anti-unification on interpreted constants
- detect whenever global proof is diverging and mitigate

One lemma to rule them all

- **merge** lemmas in group to form a single *universal* lemma
- interpolation and inductive generalization can be applied to generalize further
- new lemma reduces the global proof by blocking all POBs in its group

Reduce, reuse, recycle

- under-approximate groups that cannot be merged in current theory
- learn multiple (simple) lemmas to block a (complex) pob

$i < 0 \rightarrow x + n <= r + 0$

Lemma 1

$i < 1 \rightarrow x + n <= r + 1$

Lemma 2

$0 <= \boldsymbol{v} <= 1 \rightarrow$

$(i < \boldsymbol{v} \rightarrow x + n <= r + \boldsymbol{v})$

Group 1

$x + n <= r + i$

Generalized Lemma

UNIVERSITY OF WATERLOO

$i < 0 \rightarrow x + n <= r + 0$

$r > x \wedge i >= 0 \rightarrow r + 0 <= x + n$

$i < 1 \rightarrow x + n <= r + 1$

$r > x \wedge i >= 1 \rightarrow r + 1 <= x + n$

$0 <= v <= 1 \rightarrow$

$(i < v \rightarrow x + n <= r +$

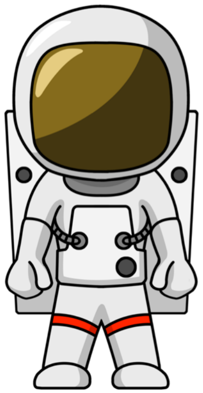
$0 <= v <= 1 \rightarrow$

$r > x \wedge i >= v \rightarrow r + v <= x + n$

$x + n <= r + i$

$r > x \rightarrow r + i <= x + n$

Conclusion



Verification of Safety Properties is FOL satisfiability

- Logic: Constrained Horn Clauses (CHC)
- “Decision” procedure: Spacer
- Now with (universal) quantifiers!

The Curse of Interpolation

- Interpolation can be amazing at guessing required terms
- but, is hard to control and masks the underlying problem!

Data driven generalization

- supplement interpolation with data-driven learning
- global view of the overall proof process
- identify diverging patterns / groups
- generalize lemmas based on groups

UNIVERSITY OF WATERLOO

?

?

?

?

?

?

THE END

Quic3: 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

Quic3: 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