Exploiting Induction and Difference Computation to Verify Array Programs

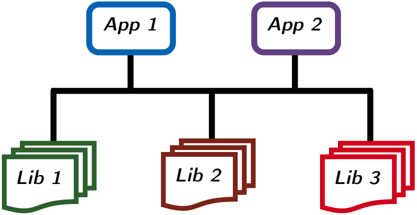
Supratik Chakraborty[1], Ashutosh Gupta[1], Divyesh Unadkat[1,2]

Indian Institute of Technology Bombay[1]
TCS Research[2]

FMI 2021

Motivation

Photo by Sahin Sezer Dincer from Pexels



- Software in safety critical applications often use arrays
- Software developed with parametric array sizes
  - Deployment-specific parameter instantiation
- Array libraries with parametric array sizes used across applications
- Verifying parametric properties of such software important

Program with Parametric Array Size

```
assume(∀x∈[0,N), A[x] = N)
```
$\varphi(N)$

```
void foo(int A[], int N)
{

  int S = 0;
  for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
      if(i+j < N)
        S = S + A[i+j];

  for(int k=0; k<N; k++)
    for(int l=0; l<N; l++)
      A[l] = A[l] + 1;
    A[k] = A[k] + S;

}
```

P_N

Arrays of parametric size N

Nested loops and branch conditions dependent on N

Quantified (possibly non-linear) pre- and post-conditions

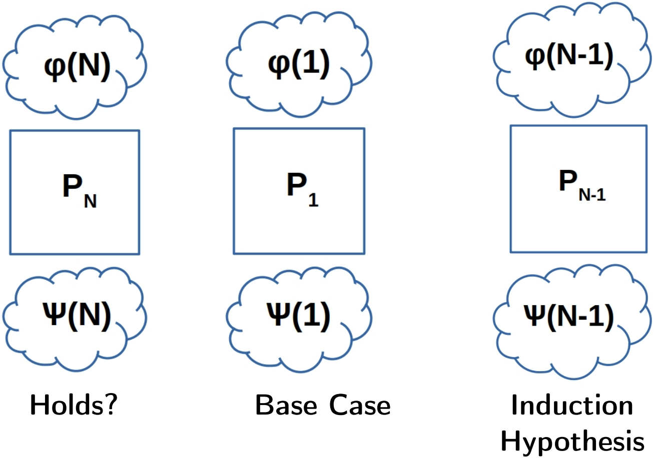
Prove the parametric Hoare triple $\{\varphi(N)\}\ \mathrm{P}_N\ \{\psi(N)\}$ for all $N > 0$

```
assert(∀x∈[0,N), A[x] = N*(N+5)/2)
```
$\psi(N)$

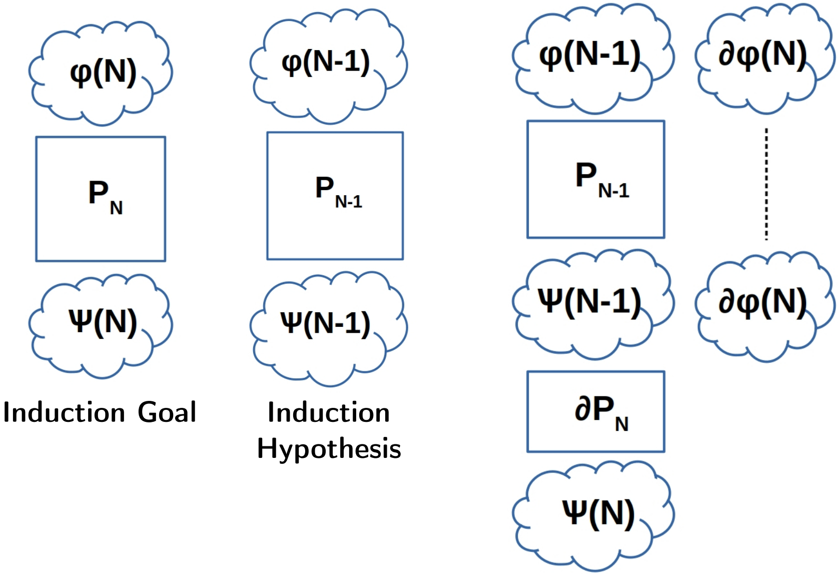
Earlier Work on Proving Programs with Arrays

- ⋆ Alberti et al 2014 - Lazy abstraction based interpolation and acceleration
- ⋆ Monniaux & Gonnord 2016 - Abstraction to array-free Horn formulas
- ⋆ Chakraborty et al 2017 - Relate array indices and loop counters for inductive proofs
- ⋆ Fedyukovich et al 2019 - Infer quantified invariants in CHCs
- ⋆ Rajkhowa & Lin 2019 - Inductive encoding of arrays as uninterpreted functions
- ⋆ Afzal et al 2020 - Output abstraction, Loop shrinking/pruning
- ⋆ Chakraborty et al 2020 - Full-Program Induction (FPI)

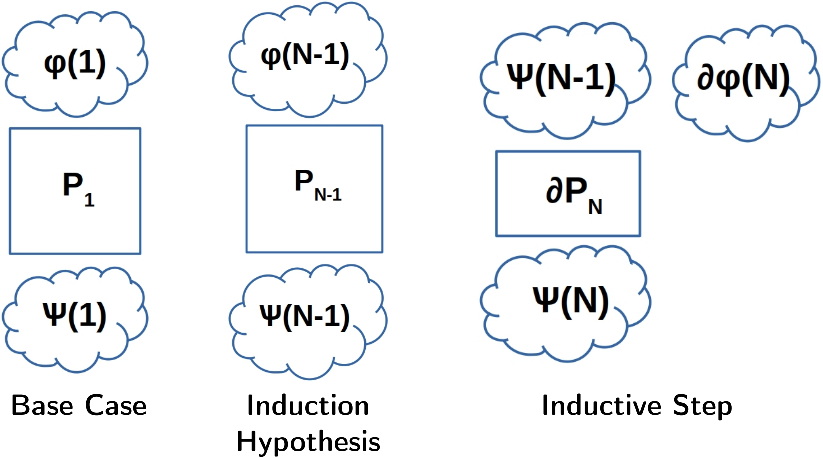
Full-Program Induction (FPI)



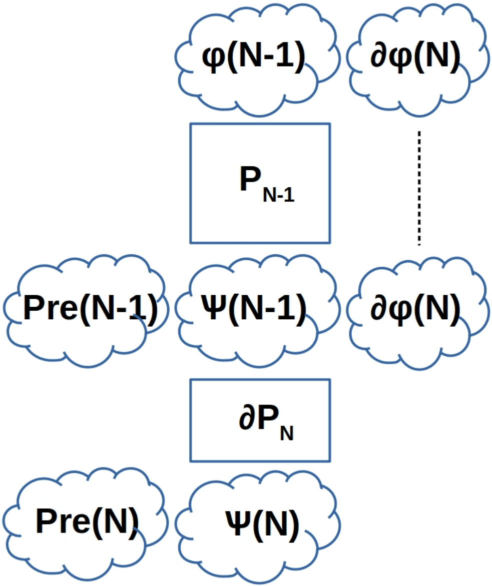
Full-Program Induction (FPI)



Full-Program Induction (FPI)



Full-Program Induction (FPI)



Divyesh Unadkat

Pros of Full-Program Induction

- Simplifies verification when $\partial \mathsf{P}_N$ is loop-free
- Computing P_{N-1}, $\psi(N-1)$ requires a simple substitution
- $\psi(N-1)$ directly useful during the inductive step
- Computing $\partial \mathsf{P}_N$ is simple for certain classes of programs
  - Programs with no data dependence across loops and loop iterations
  - No variable or array element's value depends on N
  - "Peeled" iterations of loops suffices as $\partial \mathsf{P}_N$

$\partial \mathsf{P}_N$ can be computed in other cases too, but ...

Difficulties with Full-Program Induction

```
x = 0;
for(i=0; i<N; i++)
  x = x + N*N;
  a[i] = a[i] + N;

for(j=0; j<N; j++)
  b[j] = x + j;
```

P_N

Constructing ∂P_N can get non-trivial in some cases

Computed ∂P_N may have loops

- Two loops in this example

Analyzing ∂P_N as hard as P_N

Recursive application of FPI on ∂P_N may be required even for non-nested loops

P_{N-1}:

```
x = 0;
for(i=0; i<N-1; i++)
  x = x+(N-1)*(N-1);
  a[i] = a[i] + N-1;

for(j=0; j<N-1; j++)
  b[j] = x + j;
```

∂P_N:

```
for(i=0; i<N-1; i++)
  x = x + 2*N-1;
  a[i] = a[i] + 1;

x = x + N*N;
a[N-1] = a[N-1]+N;

for(k=0; k<N-1; k++)
  b[k] = b[k] +
  (N-1)*(2*N-1)+N*N;

b[N-1] = x + N-1;
```

Additional Difficulties with FPI - Branch Conditions

$\{\ \varphi(N)\ \}\ \mathsf{P}_N\ \{\ \psi(N)\ \}$

```
assume(true)

void foo(int A[], int N) {

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

  for (int j=0; j<N; j++) {
    if(N%2 == 0) {
      A[j] = A[j] + 2;
    } else {
      A[j] = A[j] + 1;
    }
  }
}

assert(∀x∈[0,N), A[x] = N%2)
```

$\{\ \varphi(N-1)\ \}\ \mathsf{P}_{N-1}\ \{\ \psi(N-1)\ \}$

```
assume(true)

void foo(int A[], int N) {

  for (int i=0; i<N-1; i++)
    A[i] = 0;

  for (int j=0; j<N-1; j++) {
    if((N-1)%2 == 0) {
      A[j] = A[j] + 2;
    } else {
      A[j] = A[j] + 1;
    }
  }
}

assert(∀x∈[0,N-1), A[x] = (N-1)%2)
```

Additional Difficulties with FPI - Nested Loops

$\{ \varphi(N) \} \; \mathsf{P}_N \; \{ \psi(N) \}$

```
assume(true)

void foo(int A[], int N) {
  int S=0;
  for(int i=0; i<N; i++) A[i] = 0;

  for(int j=0; j<N; j++) S = S + 1;

  for(int k=0; k<N; k++) {
    for(int l=0; l<N; l++) {
      A[l] = A[l] + 1;
    }
    A[k] = A[k] + S;
  }
}

assert(∀x∈[0,N), A[x] = 2*N)
```

$\{ \varphi(N-1) \} \; \mathsf{P}_{N-1} \; \{ \psi(N-1) \}$

```
assume(true)

void foo(int A[], int N) {
  int S=0;
  for(int i=0; i<N-1; i++) A[i] = 0;

  for(int j=0; j<N-1; j++) S = S + 1;

  for(int k=0; k<N-1; k++) {
    for(int l=0; l<N-1; l++) {
      A[l] = A[l] + 1;
    }
    A[k] = A[k] + S;
  }
}

assert(∀x∈[0,N-1), A[x] = 2*(N-1))
```

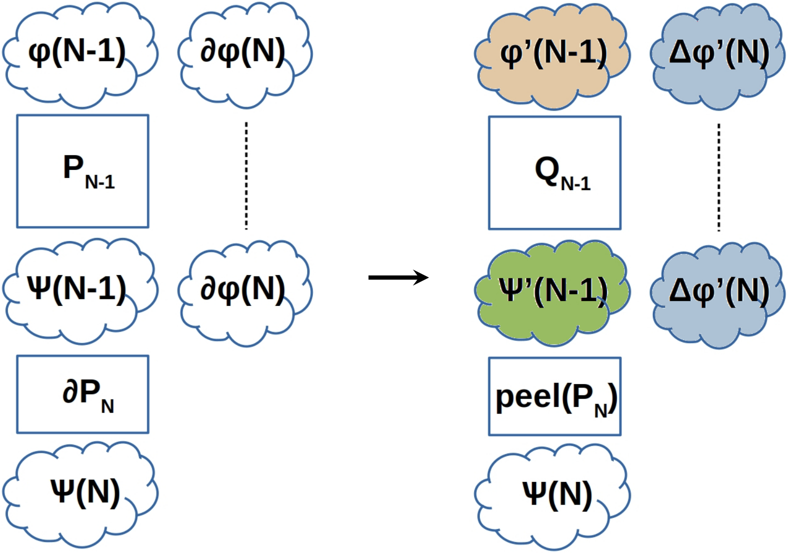
A Relook at Full-Program Induction

- Idea of inducting over full-program interesting
- Allows easy access and use of the induction hypothesis
- Can we somehow reduce the difficulty of generating of ∂P_N?
  - Caveat: No free lunches
- There is a trade-off
  - ⋆ Ease of generating difference code

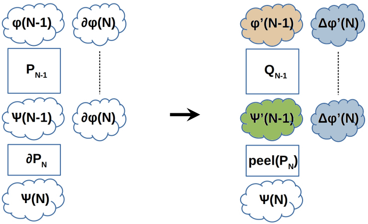
    vs

  - ⋆ Ease of using induction hypothesis

Utilizing the Available Trade-off



Utilizing the Available Trade-off



- φ'(N-1), $\Delta\varphi'$(N) similar to (not always same as) φ(N-1), $\partial\varphi$(N)
- Q_{N-1} differs from P_N only in loop bounds
- ψ'(N-1) derived from ψ(N-1) and "differences" b/w P_{N-1}, Q_{N-1}
- peel(P_N) is just the peeled iterations of loops

Ensure $\{\varphi(\text{N-1})\}\ P_{N-1}\ \{\psi(\text{N-1})\} \Rightarrow \{\varphi'(\text{N-1})\}\ Q_{N-1}\ \{\psi'(\text{N-1})\}$

Overcoming Difficulties with FPI

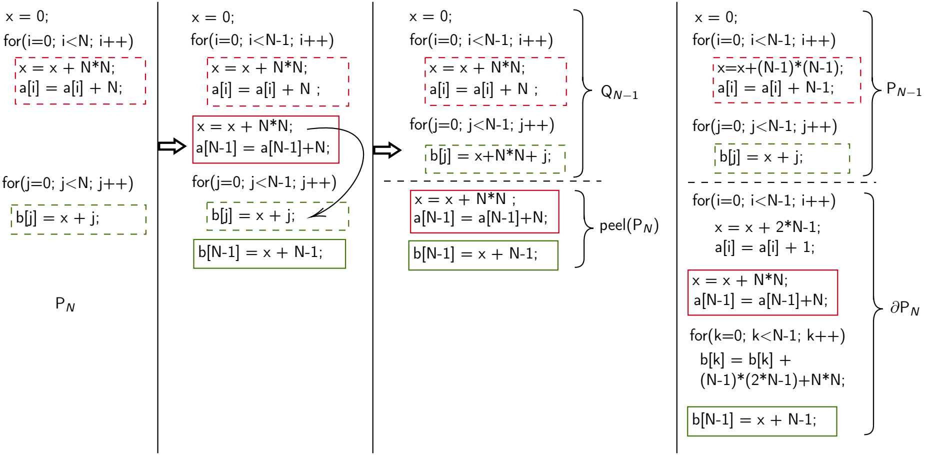
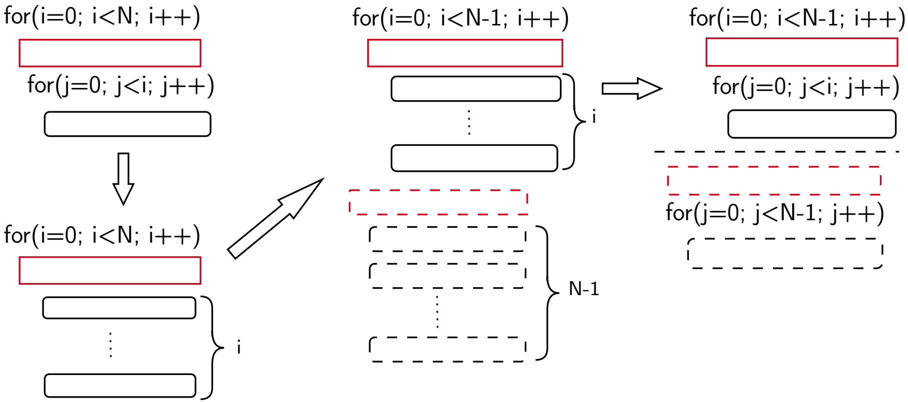


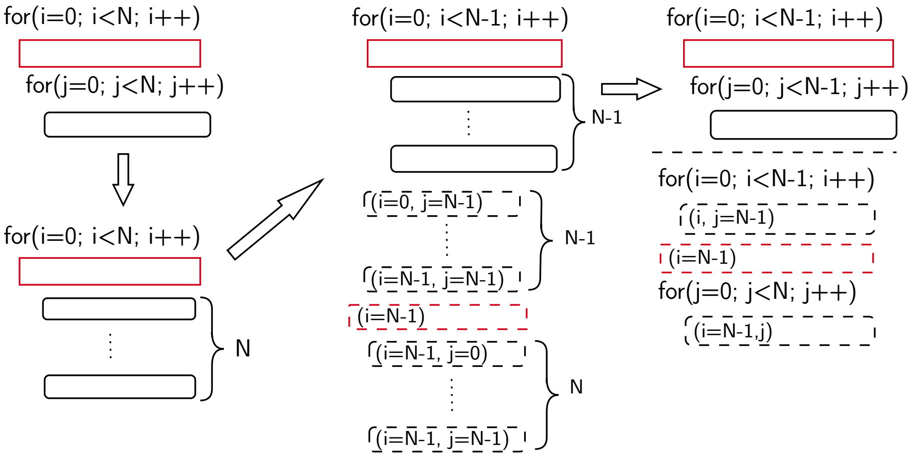
Figure: Algorithmic Construction of Programs Q_{N-1} and peel(P_N)

Divyesh Unadkat

Computing Q_{N-1} and peel(P_N)



Computing Q_{N-1} and peel(P_N)



Syntactic Restrictions on Programs

$$
\begin{array}{rcl}
\text{PB} & ::= & \text{St} \\
\text{St} & ::= & v := \text{E} \mid A[\text{E}] := \text{E} \mid \text{St;St} \mid \textbf{if}(\text{BoolE})\ \textbf{then}\ \text{St}\ \textbf{else}\ \text{St} \mid \\
 & & \textbf{for}\ (\ell := 0;\ \ell < \text{UB};\ \ell := \ell{+}1)\ \ \{\text{St}\} \\
\text{UB} & ::= & \text{UB op UB} \mid \ell \mid \text{c} \mid N \\
\text{E} & ::= & \text{E op E} \mid A[\text{E}] \mid v \mid \ell \mid \text{c} \mid N \\
\text{op} & ::= & + \mid - \mid * \mid / \\
\text{BoolE} & ::= & \text{E relop E} \mid \text{BoolE AND BoolE} \mid \text{NOT BoolE} \mid \\
 & & \text{BoolE OR BoolE}
\end{array}
$$

- No unstructured jumps
- Assignment statements in body do not update loop counter
- UB expressions is in terms of N and outer loop counters
- Nested loop is the last loop in the sequence

Correctness and Progress Guarantees

Lemma

Suppose

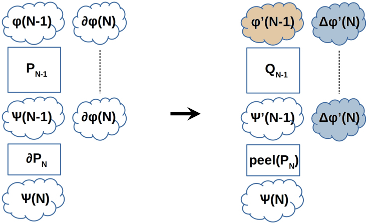
1) *Program* P_N *satisfies our syntactic restrictions*
2) *Upper bound expressions of all loops are linear expressions in* N, *the loop counters of outer loops*

Max loop nesting depth in $\mathsf{peel}(\mathsf{P}_N) <$ **Max loop nesting depth in** P_N

Theorem

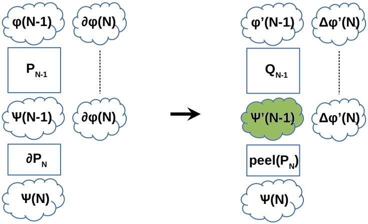
$$\mathsf{Q}_{\mathbf{N-1}};\mathbf{peel}(\mathsf{P}_{\mathbf{N}}) \equiv \mathsf{P}_{\mathbf{N}}$$

Computing $\varphi'(N-1)$ and $\Delta\varphi'(N)$



- Suppose $\varphi(N) \equiv \forall \mathtt{i} \in [0, \mathtt{N}),\ \mathtt{a[i]} = \mathtt{N}$
- $\nexists\ \partial\varphi(N)$ s.t. $\varphi(N) \Rightarrow \varphi(N-1) \wedge \partial\varphi(N)$
- $\exists\ \Delta\varphi'(N), \varphi'(N-1)$ s.t. $\varphi(N) \Rightarrow \varphi'(N-1) \wedge \Delta\varphi'(N)$
  - $\varphi'(N-1) \equiv \forall \mathtt{i} \in [0, \mathtt{N}-1),\ \mathtt{a[i]} = \mathtt{N}$
  - $\Delta\varphi'(N) \equiv \mathtt{a[N-1]} = \mathtt{N}$
- Q_{N-1} must not modify arrays/variables in $\Delta\varphi'(N)$

Computing $\psi'(N-1)$



- Infer relations between variables in Q_{N-1}, P_{N-1}
  - Difference invariant: $D(V_Q, V_P)$
- $\psi'(N-1) \equiv \exists V_P. \ (\psi(N-1) \wedge D(V_Q, V_P))$
- Finding sufficient $D(V_Q, V_P)$ is often "simpler" than generating invariants to prove the post-condition directly
- Guess-and-check $D(V_Q, V_P)$ using simple templates
  - $\bullet = \bullet + \mathtt{f}_1(\mathtt{N})$ and $\forall \mathtt{i}. \ \bullet[\mathtt{g}(\mathtt{i})] = \bullet[\mathtt{g}(\mathtt{i})] + \mathtt{f}_2(\mathtt{i}, \mathtt{N})$
  - $\mathtt{f}, \mathtt{g}$ linear/quadratic

Computing $\psi'(N-1)$

Loop invariants:

$\phi(N) \equiv \forall i \in [0,N),\ a[i] = N$

```
void foo(int a[], int N)
{
  int b[N], x = 0;

  for(int i=0; i<N; i++) {
    x = x + a[i];
  }

  for(int j=0; j<N; j++) {
    b[j] = x + j;
  }

}
```

P_N

$x = i \times N$

$\forall k_1 \in [0, j),\ b[k_1] = k_1 + N^2$

$\psi(N) \equiv \forall i \in [0,N),\ b[i] = i + N^2$

Computing $\psi'(N-1)$

Difference invariants $D(V_Q, V_P)$:

ϕ(N-1) $\equiv$ ∀i∈[0,N-1), a[i] = N-1

$\forall i \in [0, N-1),\ a'[i] - a[i] = 1$

ϕ'(N-1) $\equiv$ ∀i∈[0,N-1), a'[i] = N

```
void foo(int a[], int N)
{
  int b[N-1], x = 0;

  for(int i=0; i<N-1; i++) {
    x = x + a[i];
  }

  for(int j=0; j<N-1; j++) {
    b[j] = x + j;
  }

}
```

P_{N-1}

$x' - x = N-1$

$\forall j \in [0, N-1),\ b'[j] - b[j] = 2 \times N - 1$

```
void bar(int a'[], int N)
{
  int b'[N-1], x' = 0;

  for(int i=0; i<N-1; i++) {
    x' = x' + a'[i];
  }

  for(int j=0; j<N-1; j++) {
    b'[j] = x' + N + j;
  }

}
```

Q_{N-1}

$\psi'(N-1) \equiv \exists V_P.\ (\psi(N-1) \wedge D(V_Q, V_P))$

ψ(N-1) $\equiv$ ∀i∈[0,N-1), b[i]=i+(N-1)^2

ψ'(N-1) $\equiv$??

Computing $\psi'(N-1)$

Difference invariants $D(V_Q, V_P)$:

ϕ(N-1) $\equiv$ $\forall$i$\in$[0,N-1), a[i] = N-1

$\forall$i $\in$ [0, N-1), a′[i] - a[i] = 1

ϕ'(N-1) $\equiv$ $\forall$i$\in$[0,N-1), a'[i] = N

```
void foo(int a[], int N)
{
  int b[N-1], x = 0;

  for(int i=0; i<N-1; i++) {
    x = x + a[i];
  }

  for(int j=0; j<N-1; j++) {
    b[j] = x + j;
  }

}
```

P_{N-1}

x′ - x = N-1

$\forall$j $\in$ [0, N-1), b′[j] - b[j] = 2 $\times$ N - 1

```
void bar(int a'[], int N)
{
  int b'[N-1], x' = 0;

  for(int i=0; i<N-1; i++) {
    x' = x' + a'[i];
  }

  for(int j=0; j<N-1; j++) {
    b'[j] = x' + N + j;
  }

}
```

Q_{N-1}

$\psi'(N\text{-}1) \equiv \exists V_P.\ (\psi(N\text{-}1) \wedge D(V_Q, V_P))$

ψ(N-1) $\equiv$ $\forall$i$\in$[0,N-1), b[i]=i+$(N-1)^2$

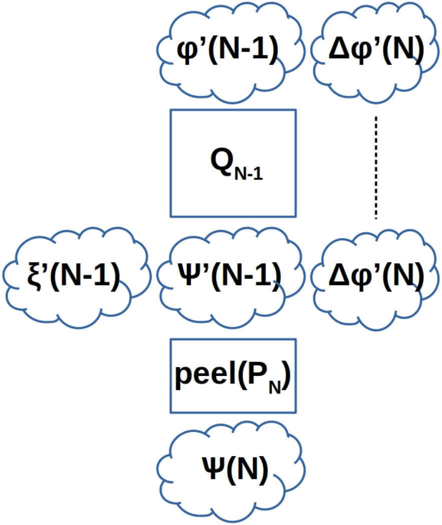
ψ'(N-1) $\equiv$ $\forall$i$\in$[0,N-1), b'[i]=i+N^2

Proving ∃ Quantified Post-conditions

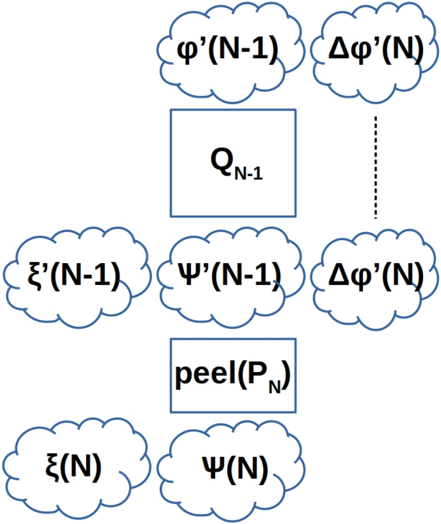
```
assume(true) //φ(N)

void imax(int A[], int N) { //P_N
  int max = A[0];

  for (int i=0; i<N; i++) {
    if(max < A[i]) {
      max = A[i];
    }
  }
}

assert(∃x∈[0,N), A[x]=max) //ψ(N)
```

```
assume(true) //φ'(N-1)

void imax'(int A'[], int N) {    //Q_N-1
  int max' = A[0];

  for (int i=0; i<N-1; i++) {
    if(max' < A'[i]) {
      max' = A'[i];
    }
  }
}

max'=max ∧ ∀x∈[0,N-1), A'[x]=A[x] //D(V',V)

assume(∃x∈[0,N-1), A'[x]=max') //ψ'(N-1)

if(max' < A'[N-1]) {    // peel(P_N)
  max' = A'[N-1];
}

assert(∃x∈[0,N), A'[x]=max') //ψ(N)
```

Strengthening Pre, Post using $\xi'(N-1)$, $\xi(N)$



Strengthening Pre, Post using $\xi'(N-1)$, $\xi(N)$



Find ξ s.t. $\exists V_P. \; (\xi(N-1) \wedge D(V_Q, V_P)) \Rightarrow \xi'(N-1)$

Soundness Guarantee

Theorem

Suppose $\exists\ M > 0$ *s.t.*

1) $\{\varphi(N)\}\ \mathrm{P}_N\ \{\psi(N) \land \xi(N)\}$ *holds for* $1 \leq N \leq M$, *for some* $M > 0$
2) $\xi(N) \land D(V_{\mathrm{Q}}, V_{\mathrm{P}}) \Rightarrow \xi'(N)$ *holds for all* $N > 0$
3) $\{\xi'(N-1) \land \Delta\varphi'(N)\ \land\ \psi'(N-1)\}\ \mathrm{peel}(\mathrm{P}_N)\ \{\xi(N) \land \psi(N)\}$ *holds for all* $N \geq M$

$$\{\varphi(N)\}\ \mathrm{P}_N\ \{\psi(N)\}\ \textbf{holds for all}\ N > 0$$

Benchmarking

- Developed a prototype tool Diffy
- Experiments on 303 benchmarks adapted from SV-COMP Arrays++
- Performance compared with the following tools:
  - ⋆ Vajra v1.0 - Chakraborty et al 2017
  - ⋆ VeriAbs v1.4.1-12 - Afzal et al 2020
  - ⋆ VIAP v1.1 - Rajkhowa & Lin 2019
- Intel i7-6500U CPU, 2.5 GHz, 16GB RAM, Ubuntu 18.04.5 LTS
- Time limit - 60s

Experimental Evaluation - Diffy

Program Category		Diffy			Vajra		VeriAbs		VIAP		
		S	U	TO	S	U	S	TO	S	U	TO
Safe C1	110	110	0	0	110	0	96	14	16	1	93
Safe C2	24	21	0	3	0	24	5	19	4	0	20
Safe C3	23	20	3	0	0	23	9	14	0	23	0
Total	157	151	3	3	110	47	110	47	20	24	113
Unsafe C1	99	98	1	0	98	1	84	15	98	0	1
Unsafe C2	24	24	0	0	17	7	19	5	22	0	2
Unsafe C3	23	20	3	0	0	23	22	1	0	23	0
Total	146	142	4	0	115	31	125	21	120	23	3

Table: Summary of the experimental results. S is successful result. U is inconclusive result. TO is timeout of 60s.

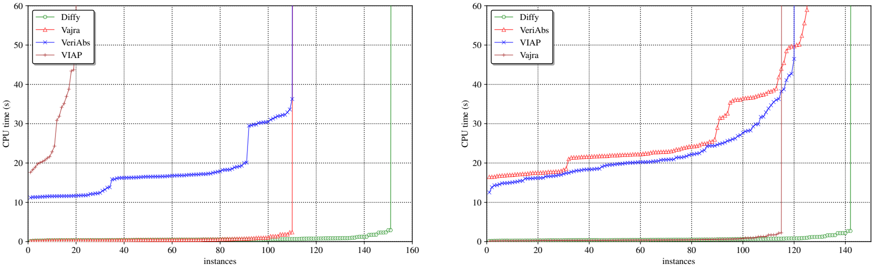
C1 - programs with standard array operations such as min, max, copy

C2 - branch conditions affected by N, modulo operators in programs

C3 - nested loop programs

Diffy out-performs *winners* of SV-COMP from Arrays sub-category

Performance Evaluation - Overall

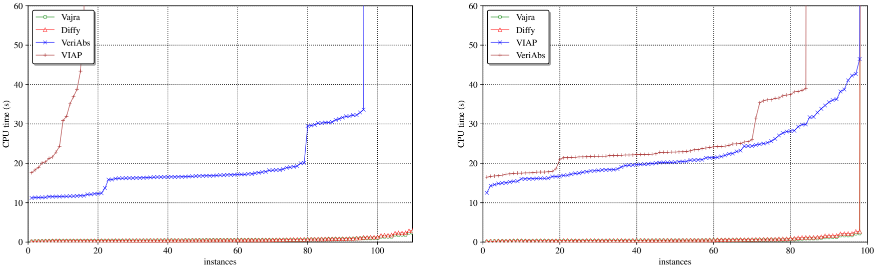


(a) (b)

Figure: Cactus Plots (a) All Safe Benchmarks (b) All Unsafe Benchmarks

- ⋆ Diffy is $10\times$ faster on most benchmarks compared to VIAP, VeriAbs
- ⋆ Violations are reported by simple bmc when base case fails

Performance Evaluation - C1

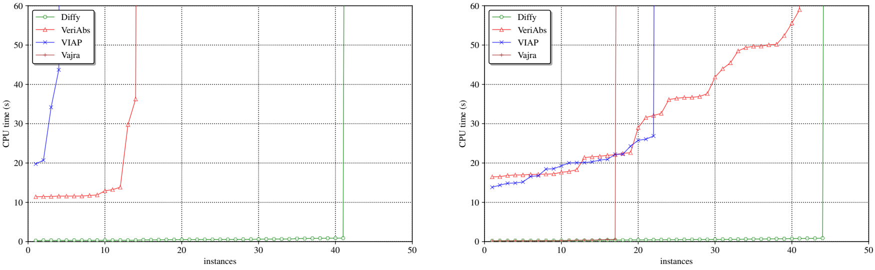


(a) (b)

Figure: Cactus Plots (a) Safe C1 Benchmarks (b) Unsafe C1 Benchmarks

- ⋆ Diffy has comparable performance with Vajra
- ⋆ Diffy out-performs VeriAbs and VIAP

Performance Evaluation - C2 & C3



(a) (b)

Figure: Cactus Plots (a) Safe C2 & C3 (b) Unsafe C2 & C3 Benchmarks

- ⋆ Nested loops are out-of-scope for Vajra and VIAP
- ⋆ VeriAbs verifies 3 programs from C2 and C3 that Diffy is unable to

Take Aways

functional
Artifact
CAV
AEC
Evaluated
available
reusable

Diffy Artifact

- Presented a novel, property driven and efficient technique that
  - performs simple transformations to enable full-program induction
  - computes *difference invariants* to aid the induction step
    - often simpler than loop invariants
- Future work
  - Leverage translation validation for computing difference invariants
  - Expand scope beyond the currently supported programs

Diffy Artifact - https://doi.org/10.6084/m9.figshare.14509467

Thank you