

Diffy: Inductive Reasoning of Array Programs using Difference Invariants



Supratik Chakraborty¹, Ashutosh Gupta¹, Divyesh Unadkat^{1,2}

Indian Institute of Technology Bombay¹
TCS Research²

CAV 2021



Divyesh Unadkat

CAV 2021

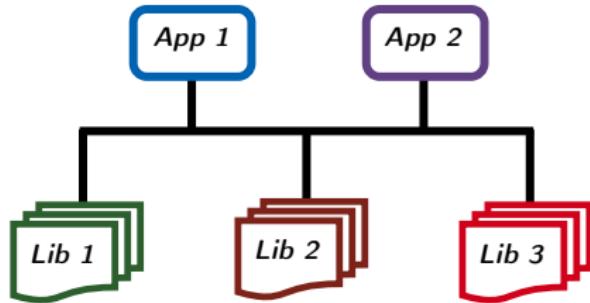
Diffy: Inductive Reasoning of Array Programs using Difference Invariants

1 / 35

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

Divyesh Unadkat

Program with Parametric Array Size

assume($\forall x \in [0, N], A[x] = N$)

$\varphi(N)$

```
1. void foo(int A[], int N)
2. {
3.     int S = 0;
4.     for(int i=0; i<N; i++)
5.         for(int j=0; j<N; j++)
6.             if(i+j < N)
7.                 S = S + A[i+j];
8.     for(int k=0; k<N; k++)
9.         for(int l=0; l<N; l++)
10.            A[l] = A[l] + 1;
11.            A[k] = A[k] + S;
12. }
```

Arrays of parametric size N

Nested loops and branch conditions dependent on N

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

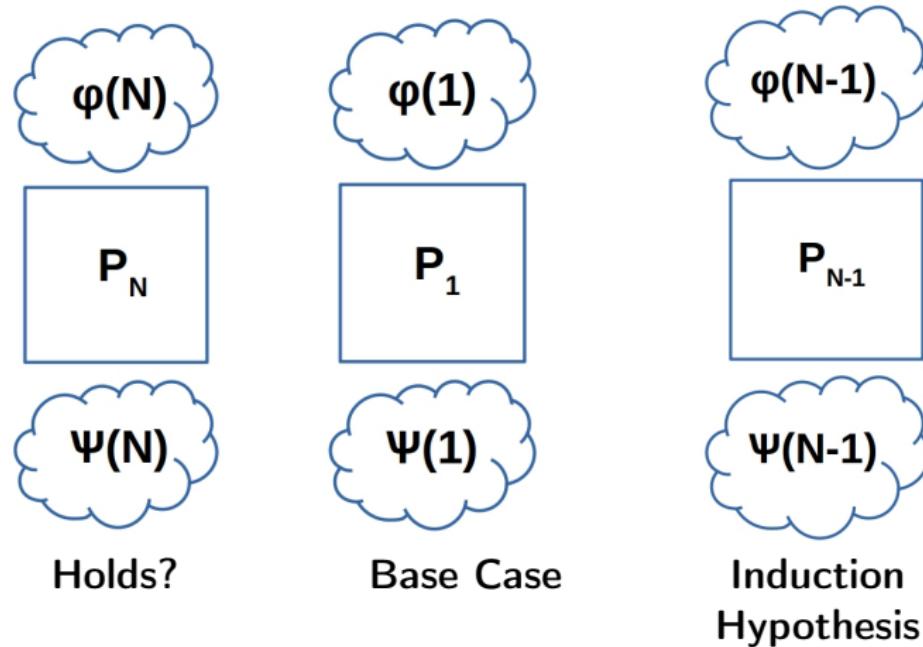
Prove the parametric Hoare triple
 $\{\varphi(N)\} P_N \{\psi(N)\}$ for all $N > 0$

assert($\forall x \in [0, N], A[x] = N * (N+5)/2$) $\psi(N)$

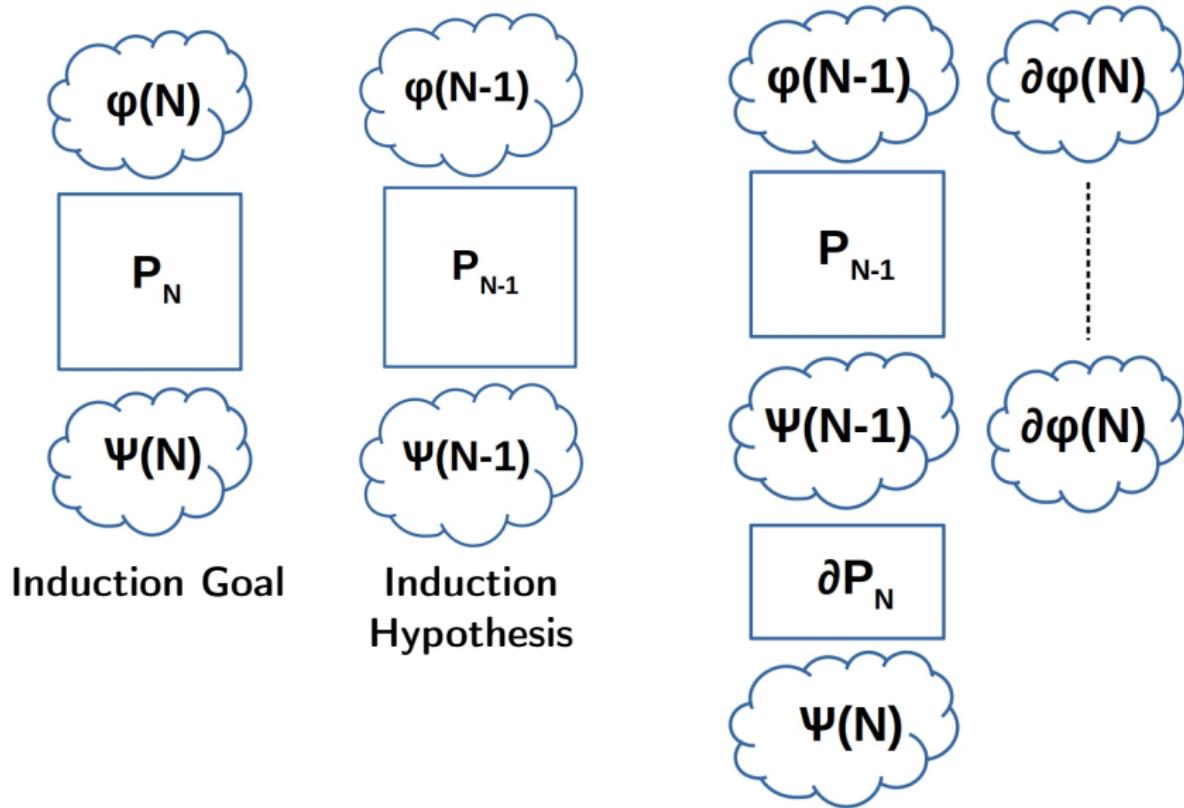
Earlier Work on Proving Programs with Arrays

- ★ 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)

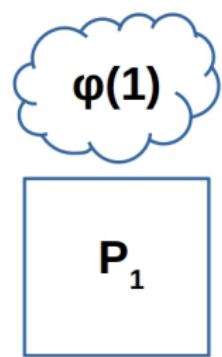
Full-Program Induction (FPI)



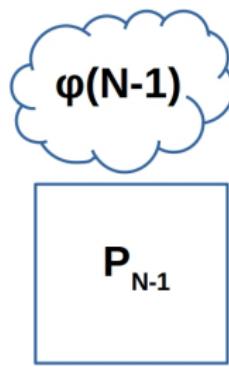
Full-Program Induction (FPI)



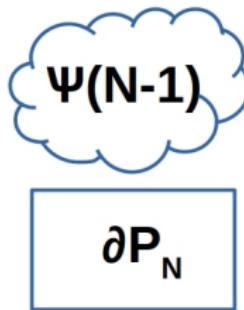
Full-Program Induction (FPI)



Base Case

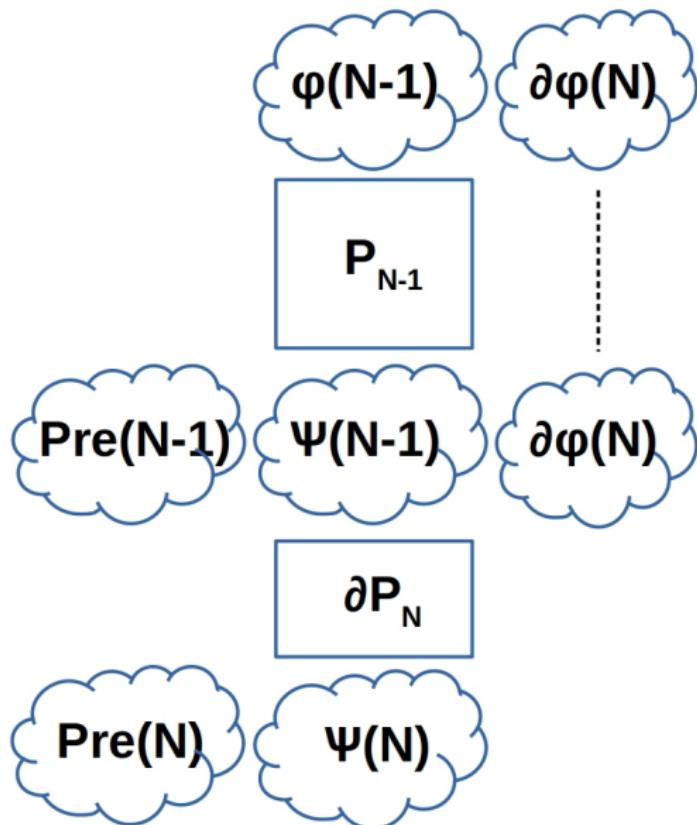


Induction Hypothesis



Inductive Step

Full-Program Induction (FPI)



Pros of Full-Program Induction

- Simplifies verification when ∂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 ∂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 ∂P_N

∂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

$x = 0;$
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;

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

$\{ \varphi(N) \} P_N \{ \psi(N) \}$

assume(true)

```
1. void foo(int A[], int N) {  
2.     for (int i=0; i<N; i++)  
3.         A[i] = 0;  
  
4.     for (int j=0; j<N; j++) {  
5.         if(N%2 == 0) {  
6.             A[j] = A[j] + 2;  
7.         } else {  
8.             A[j] = A[j] + 1;  
9.         }  
10.    }  
11.}
```

assert($\forall x \in [0, N], A[x] = N \% 2$)

$\{ \varphi(N - 1) \} P_{N-1} \{ \psi(N - 1) \}$

assume(true)

```
1. void foo(int A[], int N) {  
2.     for (int i=0; i<N-1; i++)  
3.         A[i] = 0;  
  
4.     for (int j=0; j<N-1; j++) {  
5.         if((N-1)%2 == 0) {  
6.             A[j] = A[j] + 2;  
7.         } else {  
8.             A[j] = A[j] + 1;  
9.         }  
10.    }  
11.}
```

assert($\forall x \in [0, N-1], A[x] = (N-1) \% 2$)

Additional Difficulties with FPI - Nested Loops

$$\{ \varphi(N) \} P_N \{ \psi(N) \}$$

```
assume(true)
```

```
1. void foo(int A[], int N) {  
2.     int S=0;  
3.     for(int i=0; i<N; i++) A[i] = 0;  
4.     for(int j=0; j<N; j++) S = S + 1;  
5.     for(int k=0; k<N; k++) {  
6.         for(int l=0; l<N; l++) {  
7.             A[l] = A[l] + 1;  
8.         }  
9.         A[k] = A[k] + S;  
10.    }  
11.}
```

```
assert(∀x∈[0,N), A[x] = 2*N)
```

$$\{ \varphi(N-1) \} P_{N-1} \{ \psi(N-1) \}$$

```
assume(true)
```

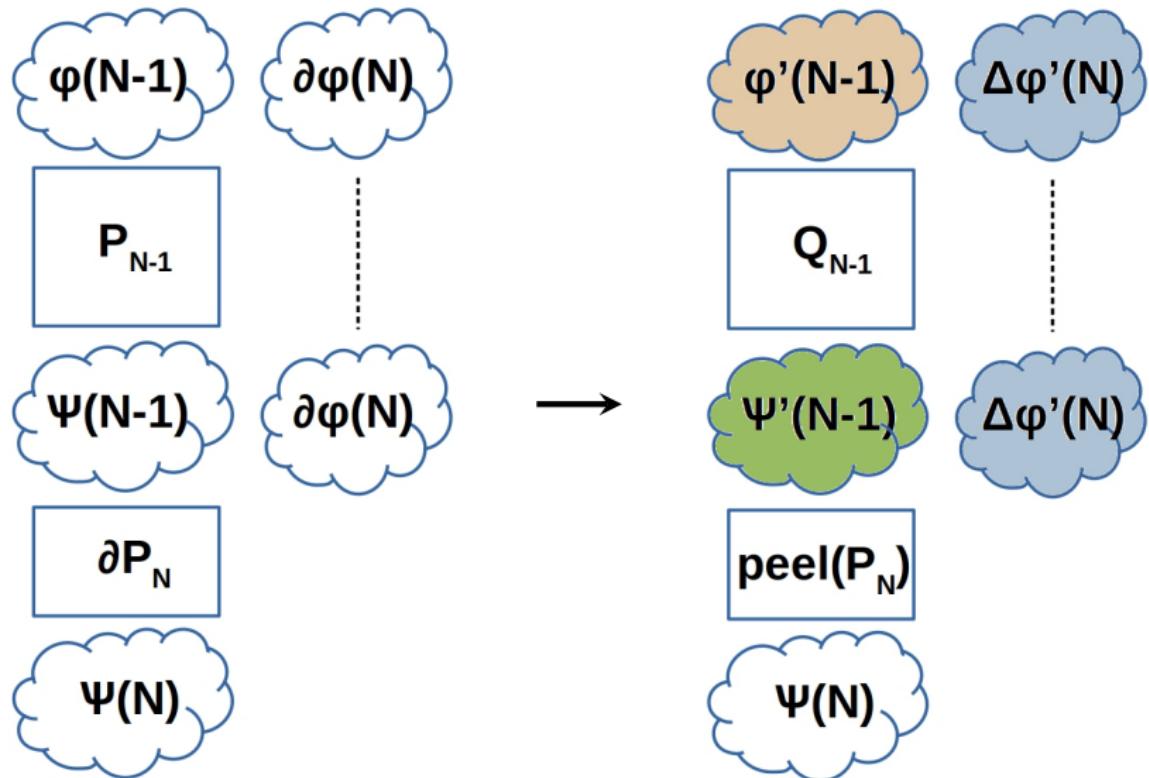
```
1. void foo(int A[], int N) {  
2.     int S=0;  
3.     for(int i=0; i<N-1; i++) A[i] = 0;  
4.     for(int j=0; j<N-1; j++) S = S + 1;  
5.     for(int k=0; k<N-1; k++) {  
6.         for(int l=0; l<N-1; l++) {  
7.             A[l] = A[l] + 1;  
8.         }  
9.         A[k] = A[k] + S;  
10.    }  
11.}
```

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

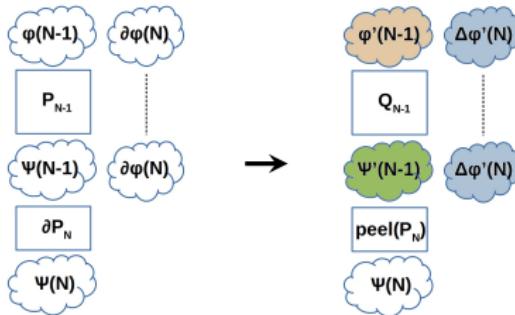
A Relook at Full-Program Induction

- 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



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

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

Divyesh Unadkat

Overcoming Difficulties with FPI

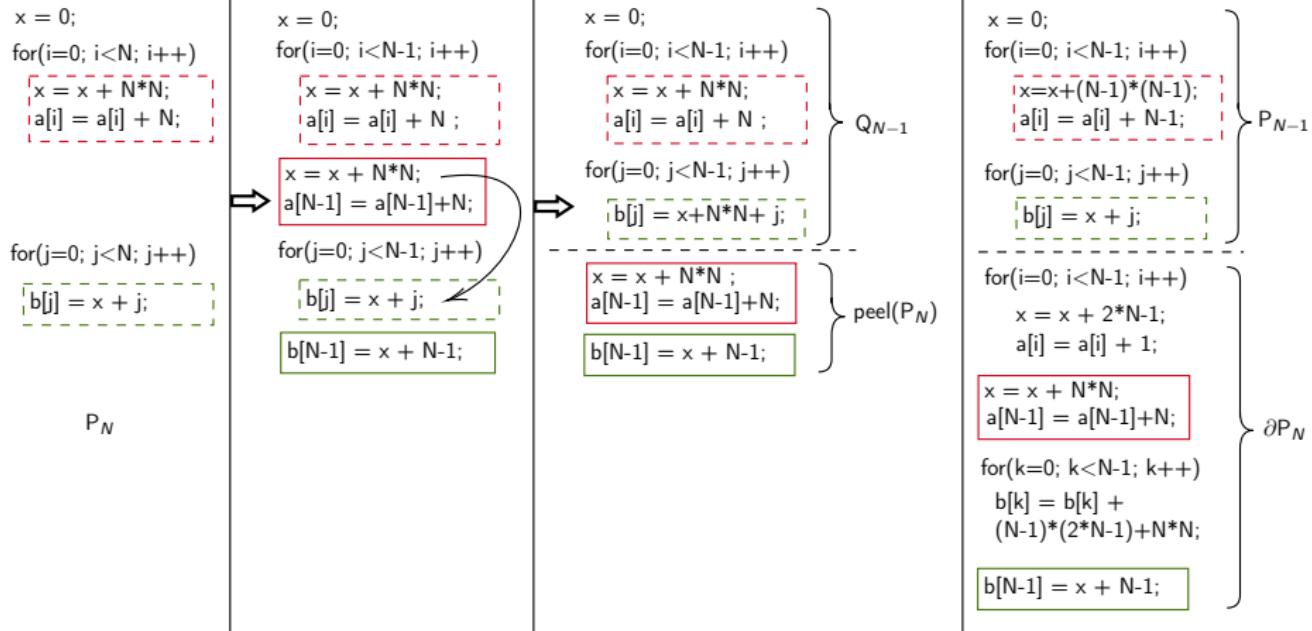
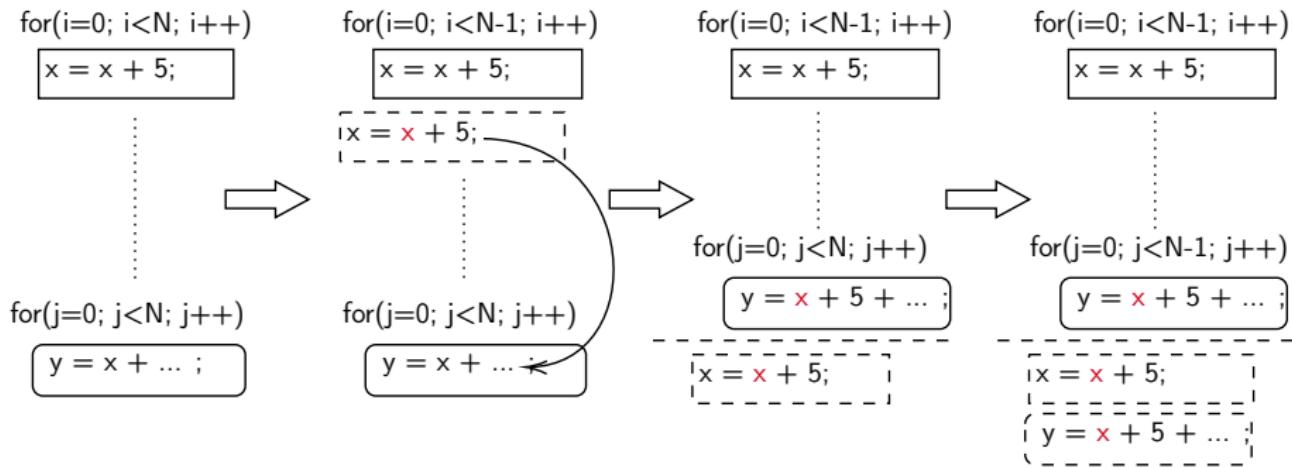


Figure: Algorithmic Construction of Programs Q_{N-1} and $\text{peel}(P_N)$

Divyesh Unadkat

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



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

```
for(i=0; i<N; i++)
```



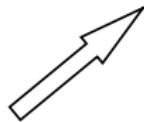
```
    for(j=0; j<i; j++)
```



```
for(i=0; i<N; i++)
```



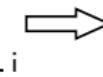
⋮



```
for(i=0; i<N-1; i++)
```



⋮



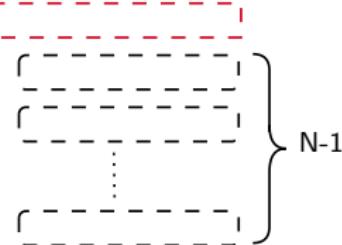
```
for(i=0; i<N-1; i++)
```



```
    for(j=0; j<i; j++)
```



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

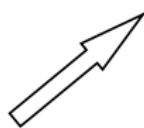


Computing Q_{N-1} and $\text{peel}(P_N)$

```
for(i=0; i<N; i++)
```



```
    for(j=0; j<N; j++)
```



```
for(i=0; i<N; i++)
```



} N

```
for(i=0; i<N-1; i++)
```



} N-1



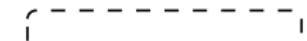
```
for(i=0; i<N-1; i++)
```



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



```
for(i=0; i<N-1; i++)
```



```
    for(j=0; j<N; j++)
```



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

```
for(i=0; i<N; i++)
```



```
    for(j=0; j<N; j++)
```

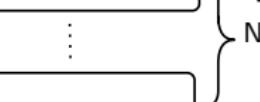


```
for(i=0; i<N; i++)
```



} N

```
for(i=0; i<N-1; i++)
```



N-1

$\{(i=0, j=N-1)\}$

$\{(i=N-1, j=N-1)\}$

$\{(i=N-1, j=0)\}$

$\{(i=N-1, j=N-1)\}$

```
for(i=0; i<N-1; i++)
```



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



for(i=0; i<N-1; i++)

$\{(i, j=N-1)\}$

$\{(i=N-1)\}$

for(j=0; j<N; j++)

$\{(i=N-1, j)\}$

Syntactic Restrictions on Programs

PB ::= St

St ::= $v := E$ | $A[E] := E$ | $St; St$ | **if**(BoolE) **then** St **else** St |
for ($\ell := 0$; $\ell < UB$; $\ell := \ell + 1$) {St}

UB ::= UB op UB | ℓ | c | N

E ::= E op E | A[E] | v | ℓ | c | N

op ::= + | - | * | /

BoolE ::= E relop E | BoolE AND BoolE | NOT BoolE |
BoolE OR BoolE

- 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

Divyesh Unadkat

Correctness and Progress Guarantees

Lemma

Suppose

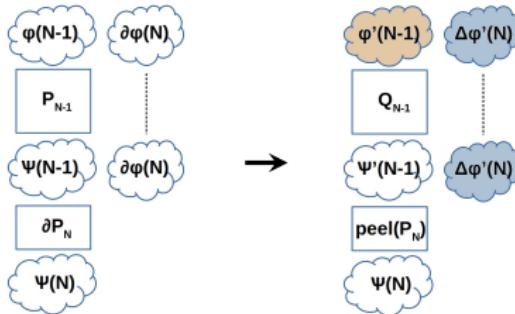
- 1) Program P_N satisfies our syntactic restrictions
- 2) Upper bound expressions of all loops are linear expressions in N , the loop counters of outer loops

Max loop nesting depth in $\text{peel}(P_N) < \text{Max loop nesting depth in } P_N$

Theorem

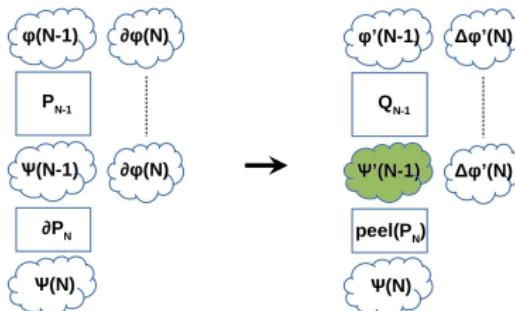
$$Q_{N-1}; \text{peel}(P_N) \equiv P_N$$

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



- Suppose $\varphi(N) \equiv \forall i \in [0, N], a[i] = N$
- $\nexists \partial\varphi(N)$ s.t. $\varphi(N) \Rightarrow \varphi(N-1) \wedge \partial\varphi(N)$
- $\exists \Delta\varphi'(N), \varphi'(N-1)$ s.t. $\varphi(N) \Rightarrow \varphi'(N-1) \wedge \Delta\varphi'(N)$
 - ▶ $\varphi'(N-1) \equiv \forall i \in [0, N-1], a[i] = N$
 - ▶ $\Delta\varphi'(N) \equiv a[N-1] = N$
- 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
 - ▶ $\bullet = \bullet + f_1(N)$ and $\forall i. \bullet [g(i)] = \bullet [g(i)] + f_2(i, N)$
 - ▶ f, g linear/quadratic

Divyesh Unadkat

Computing $\psi'(N - 1)$

Loop invariants:

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

```
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
```

$$x = i \times \mathbf{N}$$

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

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

Computing $\psi'(N - 1)$

Difference invariants $D(V_Q, V_P)$:

$$\phi(\text{N-1}) \equiv \forall i \in [0, \text{N-1}), a[i] = \text{N-1}$$

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

$$\phi'(\text{N-1}) \equiv \forall i \in [0, \text{N-1}), a'[i] = \text{N}$$

```
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;
    }
}
```

$P_{\text{N-1}}$

$$x' - x = \text{N-1}$$

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

```
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;
    }
}
```

$Q_{\text{N-1}}$

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

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

$$\psi'(\text{N-1}) \equiv ??$$

Computing $\psi'(N - 1)$

Difference invariants $D(V_Q, V_P)$:

$$\phi(\text{N-1}) \equiv \forall i \in [0, \text{N-1}), a[i] = \text{N-1}$$

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

$$\phi'(\text{N-1}) \equiv \forall i \in [0, \text{N-1}), a'[i] = \text{N}$$

```
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;
    }
}
```

$P_{\text{N-1}}$

$$x' - x = \text{N-1}$$

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

```
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;
    }
}
```

$Q_{\text{N-1}}$

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

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

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

Proving \exists Quantified Post-conditions

```
assume(true) // $\varphi(N)$ 
```

```
1. void imax(int A[], int N) { //PN
2.     int max = A[0];
4.     for (int i=0; i<N; i++) {
5.         if(max < A[i]) {
6.             max = A[i];
7.         }
8.     }
9. }
```

```
assume(true) // $\varphi'(N-1)$ 
```

```
1. void imax'(int A'[], int N) { //QN-1
2.     int max' = A'[0];
3.     for (int i=0; i<N-1; i++) {
4.         if(max' < A'[i]) {
5.             max' = A'[i];
6.         }
7.     }
8. }
```

$\text{max}' = \text{max} \wedge \forall x \in [0, N-1], A'[x] = A[x] \ // D(V', V)$

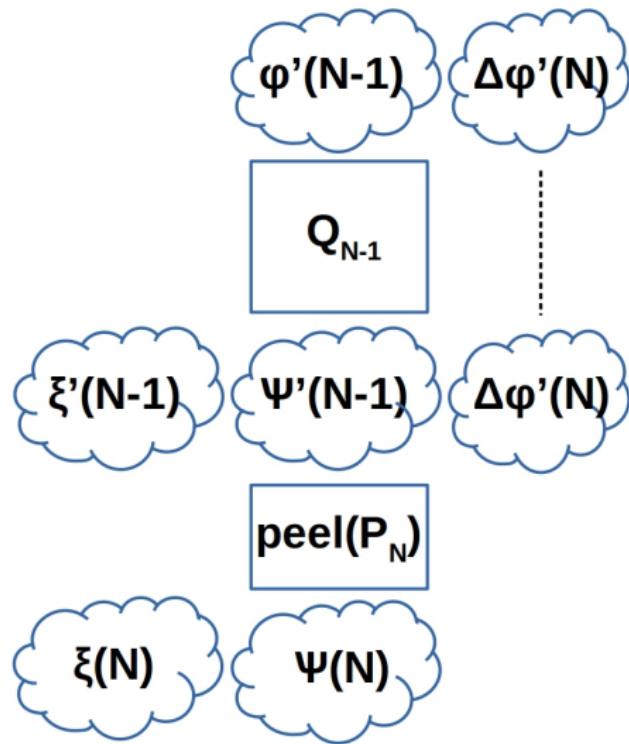
```
assert( $\exists x \in [0, N], A[x] = \text{max}$ ) // $\psi(N)$ 
```

```
assume( $\exists x \in [0, N-1], A'[x] = \text{max}'$ ) // $\psi'(N-1)$ 
```

```
9. if(max' < A'[N-1]) { // peel(PN)
10.    max' = A'[N-1];
11.}
```

```
assert( $\exists x \in [0, N], A'[x] = \text{max}'$ ) // $\psi(N)$ 
```

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



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

Divyesh Unadkat

Soundness Guarantee

Theorem

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

- 1) $\{\varphi(N)\} \text{ P}_N \{\psi(N) \wedge \xi(N)\}$ holds for $1 \leq N \leq M$, for some $M > 0$
- 2) $\xi(N) \wedge D(V_Q, V_P) \Rightarrow \xi'(N)$ holds for all $N > 0$
- 3) $\{\xi'(N-1) \wedge \Delta\varphi'(N) \wedge \psi'(N-1)\} \text{ peel}(\text{P}_N) \{\xi(N) \wedge \psi(N)\}$ holds for all $N \geq M$

$\{\varphi(N)\} \text{ P}_N \{\psi(N)\}$ 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.

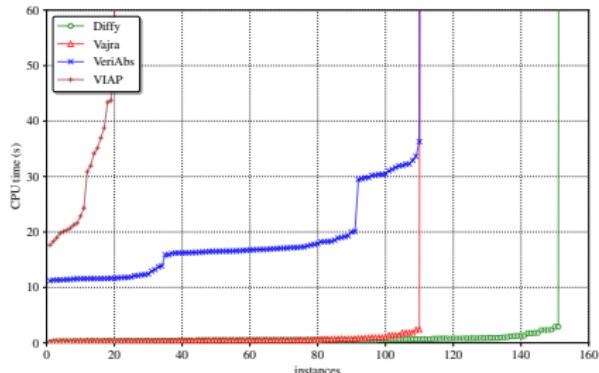
C1 - programs with standard array operations such as min, max, copy

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

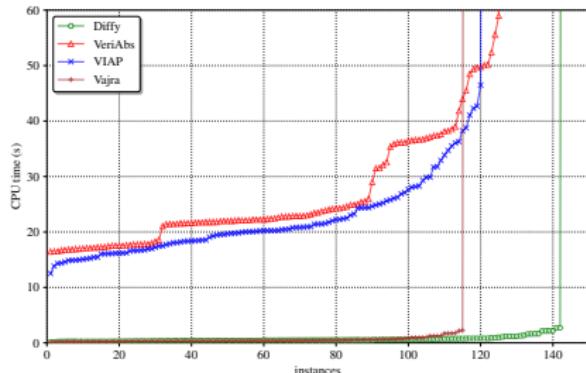
C3 - nested loop programs

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

Performance Evaluation - Overall



(a)

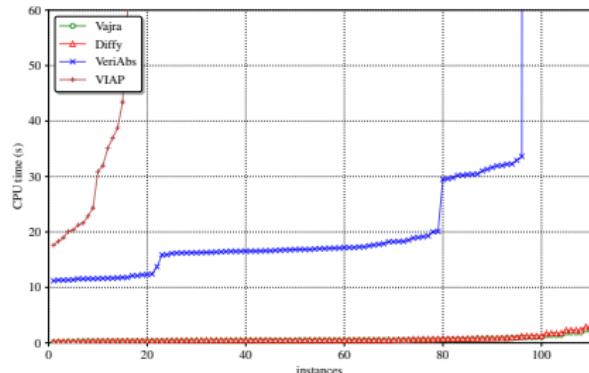


(b)

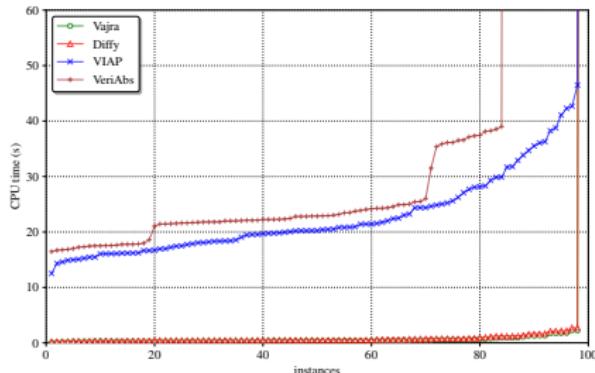
Figure: Cactus Plots (a) All Safe Benchmarks (b) All Unsafe Benchmarks

- ★ Diffy is 10× faster on most benchmarks compared to VIAP, VeriAbs
- ★ Violations are reported by simple bmc when base case fails

Performance Evaluation - C1



(a)

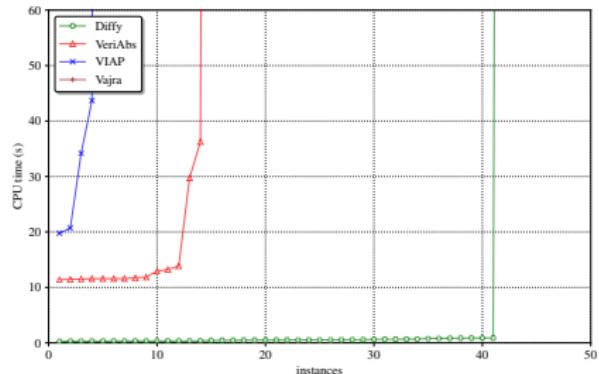


(b)

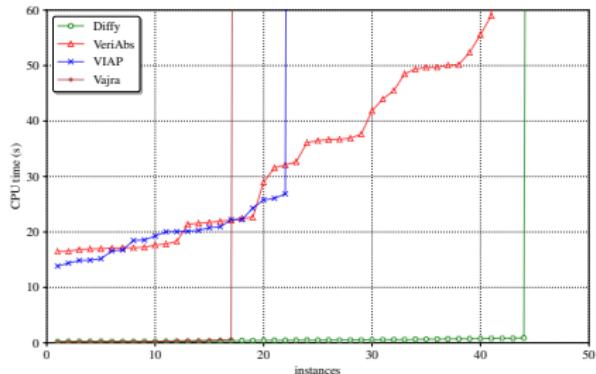
Figure: Cactus Plots (a) Safe C1 Benchmarks (b) Unsafe C1 Benchmarks

- ★ 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



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

Thank you