Inductive Reasoning for Precise and Scalable Verification of Array Programs

Divyesh Unadkat[1,2]

Advisors: Supratik Chakraborty[1], Ashutosh Gupta[1]

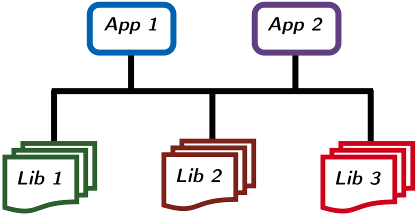
Indian Institute of Technology Bombay[1]

TCS Research[2]

9th Jan'23

Motivation

Photo by Sahin Sezer Dincer from Pexels



- Software in safety critical applications often use arrays
- Implementation uses arrays of parametric size
  - Model-specific parameter instantiation at deployment
- Libraries with parametric sized arrays used across applications
- Important to verify parametric properties of such software

Prove Correctness of Parametric Array Programs

```
void foo(int A[], int N)
{
  for (int i=0; i<N; i++)
  {
    if(!(i==0 || i==N-1))
    {
      if (A[i] < i*i*N)
      {
        A[i+1] = A[i] - N;
        A[i] = A[i-1]*(A[i-1]+1)*N;
      }
    } else {
      A[i] = (i+1)*(i+1)*N;
    }
  }
}
```

Prove Correctness of Parametric Array Programs

```
void foo(int A[], int N)
{
  for (int i=0; i<N; i++)
  {
    if(!(i==0 || i==N-1))
    {
      if (A[i] < i*i*N)
      {
        A[i+1] = A[i] - N;
        A[i] = A[i-1]*(A[i-1]+1)*N;
      }
    } else {
      A[i] = (i+1)*(i+1)*N;
    }
  }
}
```

Arrays of parametric size N

Prove Correctness of Parametric Array Programs

```
void foo(int A[], int N)
{
  for (int i=0; i<N; i++)
  {
    if(!(i==0 || i==N-1))
    {
      if (A[i] < i*i*N)
      {
        A[i+1] = A[i] - N;
        A[i] = A[i-1]*(A[i-1]+1)*N;
      }
    } else {
      A[i] = (i+1)*(i+1)*N;
    }
  }
}
```

Arrays of parametric size N

Branch conditions dependent on N
Possibly with nested loops

Prove Correctness of Parametric Array Programs

```
void foo(int A[], int N)
{
  for (int i=0; i<N; i++)
  {
    if(!(i==0 || i==N-1))
    {
      if (A[i] < i*i*N)
      {
        A[i+1] = A[i] - N;
        A[i] = A[i-1]*(A[i-1]+1)*N;
      }
    } else {
      A[i] = (i+1)*(i+1)*N;
    }
  }
}
```

P_N

Arrays of parametric size N

Branch conditions dependent on N

Possibly with nested loops

Prove Correctness of Parametric Array Programs

```
assume(∀x∈[0,N) A[x] = *)
```

$\varphi(N)$

```
void foo(int A[], int N)
{
  for (int i=0; i<N; i++)
  {
    if(!(i==0 || i==N-1))
    {
      if (A[i] < i*i*N)
      {
        A[i+1] = A[i] - N;
        A[i] = A[i-1]*(A[i-1]+1)*N;
      }
    } else {
      A[i] = (i+1)*(i+1)*N;
    }
  }
}
```

$\Big\}\, P_N$

```
assert(∀x∈[0,N) A[x] ≥ x²×N)
```

$\psi(N)$

Arrays of parametric size N

Branch conditions dependent on N

Possibly with nested loops

Pre- and post-condition formulas

- Quantified or quantifier-free
- Non-linear terms

Prove Correctness of Parametric Array Programs

```
assume(∀x∈[0,N) A[x] = *)
```

$\varphi(N)$

```
void foo(int A[], int N)
{
  for (int i=0; i<N; i++)
  {
    if(!(i==0 || i==N-1))
    {
      if (A[i] < i*i*N)
      {
        A[i+1] = A[i] - N;
        A[i] = A[i-1]*(A[i-1]+1)*N;
      }
    } else {
      A[i] = (i+1)*(i+1)*N;
    }
  }
}
```

P_N

```
assert(∀x∈[0,N) A[x] ≥ x²×N)
```

$\psi(N)$

Arrays of parametric size N

Branch conditions dependent on N

Possibly with nested loops

Pre- and post-condition formulas

- Quantified or quantifier-free
- Non-linear terms

Prove the parametric Hoare triple

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

Earlier Work on Reasoning with Array Programs

⋆ Invariant Generation Techniques

Abstract Interpretation, Static/Dynamic Analysis, Theorem Proving, SyGus, Templates

- FreqHorn - Fedyukovich et al.'19, QUIC3 - Gurfinkel et al.'18, UPDR - Karbyshev et al.'17, InvGen - Gupta & Rybalchenko'09, ...

Earlier Work on Reasoning with Array Programs

- ⋆ Invariant Generation Techniques

  Abstract Interpretation, Static/Dynamic Analysis, Theorem Proving, SyGus, Templates

  - FreqHorn - Fedyukovich et al.'19, QUIC3 - Gurfinkel et al.'18, UPDR - Karbyshev et al.'17, InvGen - Gupta & Rybalchenko'09, ...

- ⋆ Abstraction Refinement

  - Prophic3 - Mann et al.'21, VeriAbs - Afzal et al.'20, VapHor - Monniaux & Gonnord'16, Booster - Alberti et al.'14, ...

Earlier Work on Reasoning with Array Programs

⋆ Invariant Generation Techniques

Abstract Interpretation, Static/Dynamic Analysis, Theorem Proving, SyGus, Templates

- FreqHorn - Fedyukovich et al.'19, QUIC3 - Gurfinkel et al.'18, UPDR - Karbyshev et al.'17, InvGen - Gupta & Rybalchenko'09, ...

⋆ Abstraction Refinement

- Prophic3 - Mann et al.'21, VeriAbs - Afzal et al.'20, VapHor - Monniaux & Gonnord'16, Booster - Alberti et al.'14, ...

⋆ Logic-based Reasoning: RAPID - Kovacs et al.'20, VIAP - Rajkhowa & Lin'19, ...

⋆ Program Transformations: Ish-Shalom et al.'20, Seghir & Brain'12, ...

Earlier Work on Reasoning with Array Programs

- ⋆ Invariant Generation Techniques

  Abstract Interpretation, Static/Dynamic Analysis, Theorem Proving, SyGus, Templates
  - FreqHorn - Fedyukovich et al.'19, QUIC3 - Gurfinkel et al.'18, UPDR - Karbyshev et al.'17, InvGen - Gupta & Rybalchenko'09, . . .
- ⋆ Abstraction Refinement
  - Prophic3 - Mann et al.'21, VeriAbs - Afzal et al.'20, VapHor - Monniaux & Gonnord'16, Booster - Alberti et al.'14, . . .
- ⋆ Logic-based Reasoning: RAPID - Kovacs et al.'20, VIAP - Rajkhowa & Lin'19, . . .
- ⋆ Program Transformations: Ish-Shalom et al.'20, Seghir & Brain'12, . . .
- ⋆ Polyhedral Analysis
  - R-stream - Meister et al.'22, PolyMage - Mullapudi et al.'15, LLVM/Polly - Feautrier & Lengauer'11, GCC/Graphite - Trifunovic et al.'10, . . .
  - Bondhugula et al.'08, Pouchet et al.'07, Bastoul et al.'03, Feautrier'96, . . .

Earlier Work on Reasoning with Array Programs

⋆ Invariant Generation Techniques

Abstract Interpretation, Static/Dynamic Analysis, Theorem Proving, SyGus, Templates

- FreqHorn - Fedyukovich et al.'19, QUIC3 - Gurfinkel et al.'18, UPDR - Karbyshev et al.'17, InvGen - Gupta & Rybalchenko'09, ...

⋆ Abstraction Refinement

- Prophic3 - Mann et al.'21, VeriAbs - Afzal et al.'20, VapHor - Monniaux & Gonnord'16, Booster - Alberti et al.'14, ...

⋆ Logic-based Reasoning: RAPID - Kovacs et al.'20, VIAP - Rajkhowa & Lin'19, ...

⋆ Program Transformations: Ish-Shalom et al.'20, Seghir & Brain'12, ...

⋆ Polyhedral Analysis

- R-stream - Meister et al.'22, PolyMage - Mullapudi et al.'15, LLVM/Polly - Feautrier & Lengauer'11, GCC/Graphite - Trifunovic et al.'10, ...
- Bondhugula et al.'08, Pouchet et al.'07, Bastoul et al.'03, Feautrier'96, ...

⋆ Scalar Evolution Analysis: Engelen'01, Bachmann'94 ...

Thesis Contributions: New Perspectives on Inductive Reasoning

Thesis Contributions: New Perspectives on Inductive Reasoning

- ⋆ Compositional Inductive *Verification by Tiling* - [SAS'17] A glimpse
  - Prototyped in Tiler

Thesis Contributions: New Perspectives on Inductive Reasoning

- ⋆ Compositional Inductive *Verification by Tiling* - [SAS'17] A glimpse
  - Prototyped in Tiler
- ⋆ Verification by *Full-Program Induction* - [TACAS'20, SV-COMP'20, STTT'22] In detail
  - Notions of "difference" program and "difference" pre-condition
  - Prototyped in Vajra

Thesis Contributions: New Perspectives on Inductive Reasoning

- ⋆ Compositional Inductive *Verification by Tiling* - [SAS'17] A glimpse
  - Prototyped in Tiler
- ⋆ Verification by *Full-Program Induction* - [TACAS'20, SV-COMP'20, STTT'22] In detail
  - Notions of "difference" program and "difference" pre-condition
  - Prototyped in Vajra
- ⋆ *Relational Full-Program Induction* - [CAV'21, SV-COMP'22] A glimpse
  - Prototyped in Diffy

Syntax of Input Programs P_N

```
assume(∀x∈[0,N) A[x]=*)

void foo(int A[], int N)
{
  for (int i = 0; i < N; i++)
  {
    if(!(i==0 || i==N-1))
    {
      if (A[i] < i*i*N)
      {
        A[i+1] = A[i] - N;
        A[i] = A[i-1]*(A[i-1]+1)*N;
      }
    } else {
      A[i] = (i+1)*(i+1)*N;
    }
  }
}

assert(∀k∈[0,N) A[k]≥k²×N)
```

$$
\begin{array}{rcl}
\mathsf{PB} & ::= & \mathsf{St} \\
\mathsf{St} & ::= & \mathsf{AssignSt} \mid \mathsf{St;St} \mid \\
 & & \mathbf{if}(\mathsf{BoolE})\ \mathbf{then}\ \mathsf{St}\ \mathbf{else}\ \mathsf{St} \mid \\
 & & \mathbf{for}\ (\ell := 0;\ \ell < \mathsf{UB};\ \ell := \ell{+}1)\ \ \{\mathsf{St}\} \\
\mathsf{AssignSt} & ::= & v := \mathsf{E} \mid A[\mathsf{IndE}] := \mathsf{E} \\
\mathsf{E} & ::= & \mathsf{E}\ \ \mathsf{op}\ \ \mathsf{E} \mid A[\mathsf{IndE}] \mid v \mid \ell \mid \mathsf{c} \mid N \\
\mathsf{IndE} & ::= & \mathsf{IndE}\ \ \mathsf{op}\ \ \mathsf{IndE} \mid v \mid \ell \mid \mathsf{c} \mid N \\
\mathsf{UB} & ::= & \mathsf{UB}\ \mathsf{op}\ \mathsf{UB} \mid \ell \mid \mathsf{c} \mid N \\
\mathsf{BoolE} & ::= & \mathsf{E}\ \mathsf{relop}\ \mathsf{E} \mid \mathsf{BoolE\ AND\ BoolE} \mid \\
 & & \mathsf{NOT\ BoolE} \mid \mathsf{BoolE\ OR\ BoolE} \\
\mathsf{op} & ::= & + \mid - \mid \times \mid \div \\
\mathsf{relop} & ::= & == \mid < \mid \leq \mid > \mid \geq
\end{array}
$$

Specifying $\varphi(N)$ and $\psi(N)$

```
assume(∀x 0≤x<N ⇒ A[x]=*)

void foo(int A[], int N)
{
  for (int i = 0; i < N; i++)
  {
    if(!(i==0 || i==N-1))
    {
      if (A[i] < i*i*N)
      {
        A[i+1] = A[i] - N;
        A[i] = A[i-1]*(A[i-1]+1)*N;
      }
    } else {
      A[i] = (i+1)*(i+1)*N;
    }
  }
}

assert(∀k 0≤k<N ⇒ A[k]≥k²×N)
```

Quantified formulas

- $\forall I \ \ (\alpha(I, N) \Rightarrow \beta(\mathcal{A}, \mathcal{V}, I, N))$
- $\exists I \ \ (\alpha(I, N) \wedge \beta(\mathcal{A}, \mathcal{V}, I, N))$

Quantifier-free formulas: $\gamma(\mathcal{A}, \mathcal{V}, N)$

β, γ in theory of arrays/integer arithmetic

α in theory of linear integer arithmetic

No quantifier alternations

Verification by Tiling

Motivation & Intuition

```
assume(∀x∈[0,N) A[x]=*)

void foo(int A[], int N)
{
  for (int i = 0; i < N; i++)
  {
    if(i==0 || i==N-1)
    {
      A[i] = 5;
    } else {
      if (A[i] < 5)
        A[i] = A[i-1] + 1;
    }
  }
}

assert(∀x∈[0,N) A[x]≥5)
```

Motivation & Intuition

```
assume(∀x∈[0,N) A[x]=*)

void foo(int A[], int N)
{
  for (int i = 0; i < N; i++)
  {
    if(i==0 || i==N-1)
    {
      A[i] = 5;
    } else {
      if (A[i] < 5)
        A[i] = A[i-1] + 1;
    }
  }
}

assert(∀x∈[0,N) A[x]≥5)
```

A single iteration typically updates a small part of the array

Motivation & Intuition

```
assume(∀x∈[0,N) A[x]=*)

void foo(int A[], int N)
{
  for (int i = 0; i < N; i++)
  {
    if(i==0 || i==N-1)
    {
      A[i] = 5;
    } else {
      if (A[i] < 5)
        A[i] = A[i-1] + 1;
    }
  }
}

assert(∀x∈[0,N) A[x]≥5)
```

A single iteration typically updates a small part of the array

Each iteration ensures the desired property over that part

Motivation & Intuition

```
assume(∀x∈[0,N) A[x]=*)

void foo(int A[], int N)
{
  for (int i = 0; i < N; i++)
  {
    if(i==0 || i==N-1)
    {
      A[i] = 5;
    } else {
      if (A[i] < 5)
        A[i] = A[i-1] + 1;
    }
  }
}

assert(∀x∈[0,N) A[x]≥5)
```

A single iteration typically updates a small part of the array

Each iteration ensures the desired property over that part

Proofs from successive iterations compose to establish the property

Tiling an Array

Identify the region of an array that is modified in a generic loop iteration is localized

A Tile is a predicate that captures such a region

$\mathsf{Tile} : \mathsf{LoopCounter} \times \mathsf{Indices} \rightarrow \{\mathbf{tt}, \mathbf{ff}\}$ for loop L

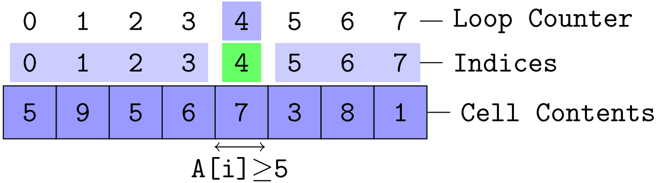
Tiling an Array

Identify the region of an array that is modified in a generic loop iteration is localized

A Tile is a predicate that captures such a region

Tile : LoopCounter $\times$ Indices $\rightarrow \{\mathbf{tt}, \mathbf{ff}\}$ for loop L

```
for (int i = 0; i < N; i++)
{
  if(i==0 || i==N-1)
  {
    A[i] = 5;
  } else {
    if (A[i] < 5)
      A[i] = A[i-1] + 1;
  }
}
```



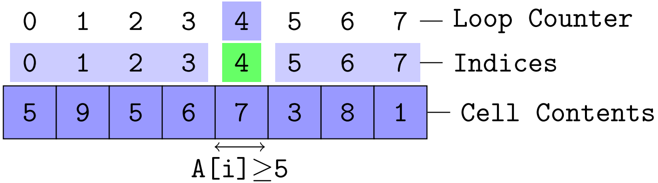
Tiling an Array

Identify the region of an array that is modified in a generic loop iteration is localized

A Tile is a predicate that captures such a region

Tile : LoopCounter $\times$ Indices $\rightarrow \{\mathbf{tt}, \mathbf{ff}\}$ for loop L

```
for (int i = 0; i < N; i++)
{
  if(i==0 || i==N-1)
  {
    A[i] = 5;
  } else {
    if (A[i] < 5)
      A[i] = A[i-1] + 1;
  }
}
```



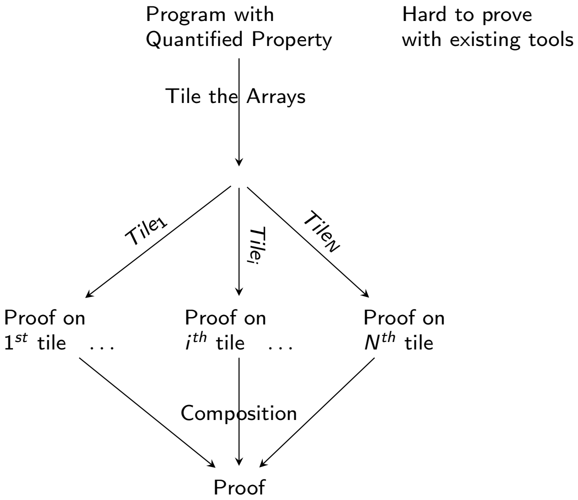
More complex tiles are possible

Challenging Benchmarks Solved by Tiling

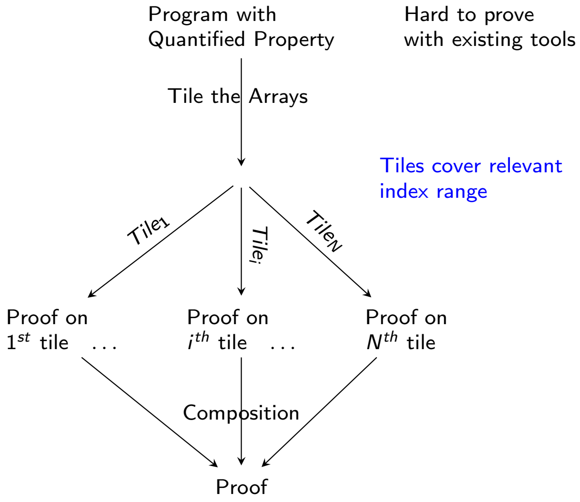
```
assume(∀x∈[0,N) A[x]=*)

void skipped(int A[], int N)
{
  for(int i = 1; i <= N/4; i++)
  {
    if( A[4*i-1] > 4*i-1 )
      A[4*i-1] = 4*i-1;

    if( A[4*i-3] > 4*i-3 )
      A[4*i-3] = 4*i-3;

    if( A[4*i-2] > 4*i-2 )
      A[4*i-2] = 4*i-2;

    if( A[4*i-4] > 4*i-4 )
      A[4*i-4] = 4*i-4;
  }
}

assert(∀x∈[0,N) A[x]≤x)
```

```
assume(∀x∈[0,N) A[x]=*)

void nonlin(int A[], int N)
{
  for (int i = 0; i < N; i++)
  {
    if(!(i==0 || i==N-1))
    {
      if (A[i] < i*i*N)
      {
        A[i+1] = A[i] - N;
        A[i] = A[i-1]*(A[i-1]+1)*N;
      }
    } else {
      A[i] = (i+1)*(i+1)*N;
    }
  }
}

assert(∀k∈[0,N) A[k]≥k²×N)
```

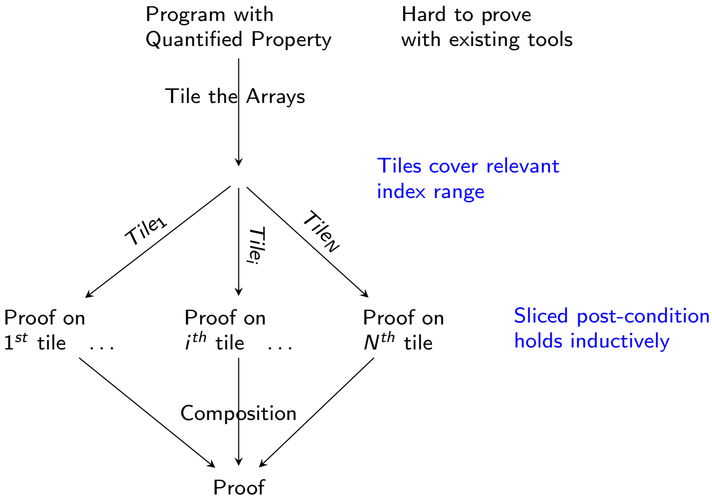
Using Tiles to Prove the Post-condition



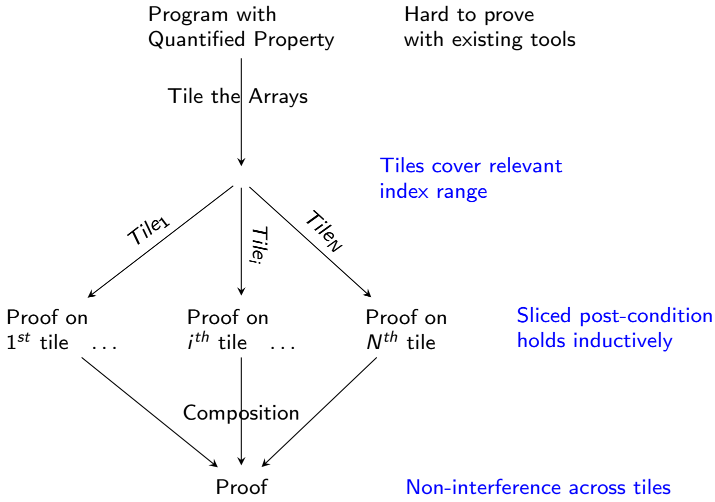
Using Tiles to Prove the Post-condition



Using Tiles to Prove the Post-condition



Using Tiles to Prove the Post-condition



Tiler - Tile-wise Reasoning Tool

Benchmark	#L	**Tiler**	Booster	Vaphor	SMACK+Corral
cpynrev.c	2	✓3.8	✓3.1	✓5.4	†
cpynswp.c	2	✓4.2	✓12.4	✓1.38	†
cpynswp2.c	3	✓10.2	✓198	✓7.2*	†
maxinarr.c	1	✓0.51	✓0.01	✓0.11	†
mininarr.c	1	✓0.53	✓0.02	✓0.13	†
poly1.c	1	TO	✓15.7	TO	†
poly2.c	2	? 6.44	? 19.5	TO	†
tcpy.c	1	? 0.65	TO	✓25.1	†
rew.c	1	✓0.48	✓0.01	TO	†
skipped.c	1	✓1.24	TO	TO	†
rewrev.c	1	✓0.39	TO	TO	†
pr4.c	1	✓0.68	TO	TO	†
pr5.c	1	✓1.32	TO	TO	†
pnr4.c	1	✓0.86	TO	TO	†
pnr5.c	1	✓1.98	TO	TO	†
mbpr4.c	4	✓12.75	TO	TO	†
mbpr5.c	5	✓18.08	TO	TO	†
copy9u.c	9	✗0.16	✗0.44	✗30.8	✗4.48
skippedu.c	1	✗0.81	✗0.02	TO	✗2.94
init9u.c	9	✗0.15	✗0.32	✗0.14	✗3.77

Tiler - Tile-wise Reasoning Tool

Benchmark	#L	**Tiler**	Booster	Vaphor	SMACK+Corral
cpynrev.c	2	✓3.8	✓3.1	✓5.4	†
cpynswp.c	2	✓4.2	✓12.4	✓1.38	†
cpynswp2.c	3	✓10.2	✓198	✓7.2*	†
maxinarr.c	1	✓0.51	✓0.01	✓0.11	†
mininarr.c	1	✓0.53	✓0.02	✓0.13	†
poly1.c	1	TO	✓15.7	TO	†
poly2.c	2	? 6.44	? 19.5	TO	†
tcpy.c	1	? 0.65	TO	✓25.1	†
rew.c	1	✓0.48	✓0.01	TO	†
skipped.c	1	✓1.24	TO	TO	†
rewrev.c	1	✓0.39	TO	TO	†
pr4.c	1	✓0.68	TO	TO	†
pr5.c	1	✓1.32	TO	TO	†
pnr4.c	1	✓0.86	TO	TO	†
pnr5.c	1	✓1.98	TO	TO	†
mbpr4.c	4	✓12.75	TO	TO	†
mbpr5.c	5	✓18.08	TO	TO	†
copy9u.c	9	✗0.16	✗0.44	✗30.8	✗4.48
skippedu.c	1	✗0.81	✗0.02	TO	✗2.94
init9u.c	9	✗0.15	✗0.32	✗0.14	✗3.77

Tiler - Tile-wise Reasoning Tool

Benchmark	#L	**Tiler**	Booster	Vaphor	SMACK+Corral
cpynrev.c	2	✓3.8	✓3.1	✓5.4	†
cpynswp.c	2	✓4.2	✓12.4	✓1.38	†
cpynswp2.c	3	✓10.2	✓198	✓7.2*	†
maxinarr.c	1	✓0.51	✓0.01	✓0.11	†
mininarr.c	1	✓0.53	✓0.02	✓0.13	†
poly1.c	1	TO	✓15.7	TO	†
poly2.c	2	? 6.44	? 19.5	TO	†
tcpy.c	1	? 0.65	TO	✓25.1	†
rew.c	1	✓0.48	✓0.01	TO	†
skipped.c	1	✓1.24	TO	TO	†
rewrev.c	1	✓0.39	TO	TO	†
pr4.c	1	✓0.68	TO	TO	†
pr5.c	1	✓1.32	TO	TO	†
pnr4.c	1	✓0.86	TO	TO	†
pnr5.c	1	✓1.98	TO	TO	†
mbpr4.c	4	✓12.75	TO	TO	†
mbpr5.c	5	✓18.08	TO	TO	†
copy9u.c	9	✗0.16	✗0.44	✗30.8	✗4.48
skippedu.c	1	✗0.81	✗0.02	TO	✗2.94
init9u.c	9	✗0.15	✗0.32	✗0.14	✗3.77

Pitfalls of Tile-wise Reasoning

```
assume(∀x∈[0,N) A[x]=1)

void foo(int A[], int N)
{
  int S=0;

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

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

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

assert(S=(2N^4-3N^3+4N^2+3N)/6)
```

Pitfalls of Tile-wise Reasoning

- Induction must be repeated for each loop in the program

```
assume(∀x∈[0,N) A[x]=1)

void foo(int A[], int N)
{
  int S=0;

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

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

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

assert(S=(2N^4-3N^3+4N^2+3N)/6)
```

Pitfalls of Tile-wise Reasoning

- Induction must be repeated for each loop in the program
- Requires post-conditions after each sequencially composed loop

```
assume(∀x∈[0,N) A[x]=1)

void foo(int A[], int N)
{
  int S=0;

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

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

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

assert(S=(2N^4-3N^3+4N^2+3N)/6)
```

Pitfalls of Tile-wise Reasoning

- Induction must be repeated for each loop in the program
- Requires post-conditions after each sequencially composed loop
- Untile-able properties

```
assume(∀x∈[0,N) A[x]=1)

void foo(int A[], int N)
{
  int S=0;

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

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

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

assert(S=(2N^4-3N^3+4N^2+3N)/6)
```

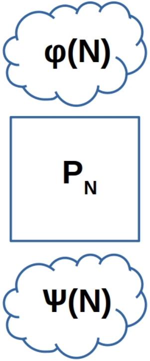
Pitfalls of Tile-wise Reasoning

- Induction must be repeated for each loop in the program
- Requires post-conditions after each sequencially composed loop
- Untile-able properties
  - For example, aggregation of array content in a scalar

```
assume(∀x∈[0,N) A[x]=1)

void foo(int A[], int N)
{
  int S=0;

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

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

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

assert(S=(2N^4-3N^3+4N^2+3N)/6)
```

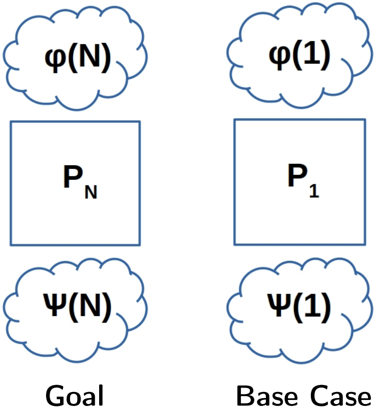
Full-Program Induction (FPI)

Full-Program Induction (FPI)

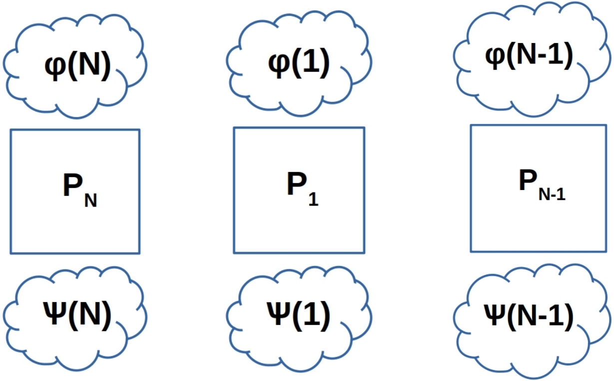


Goal

Full-Program Induction (FPI)



Full-Program Induction (FPI)

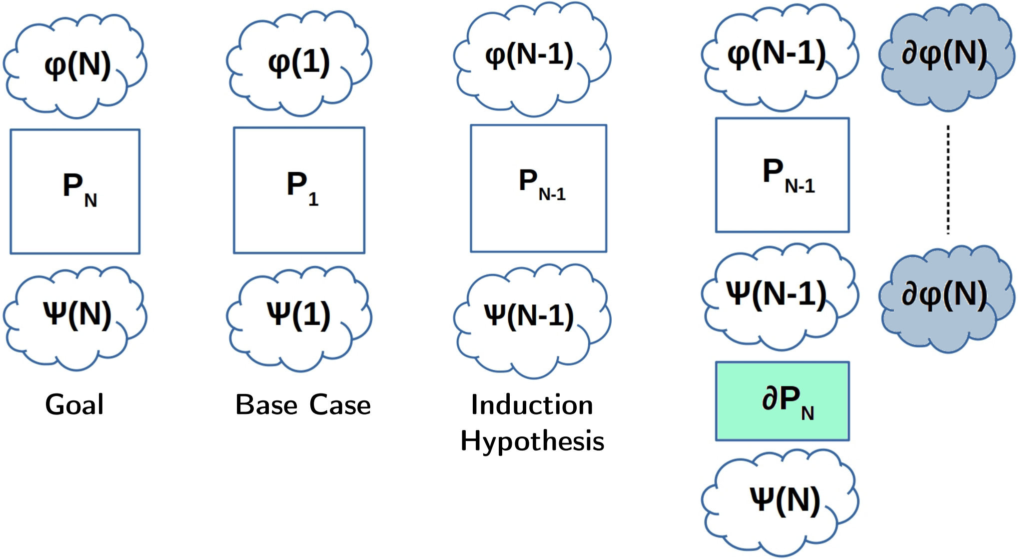


Goal

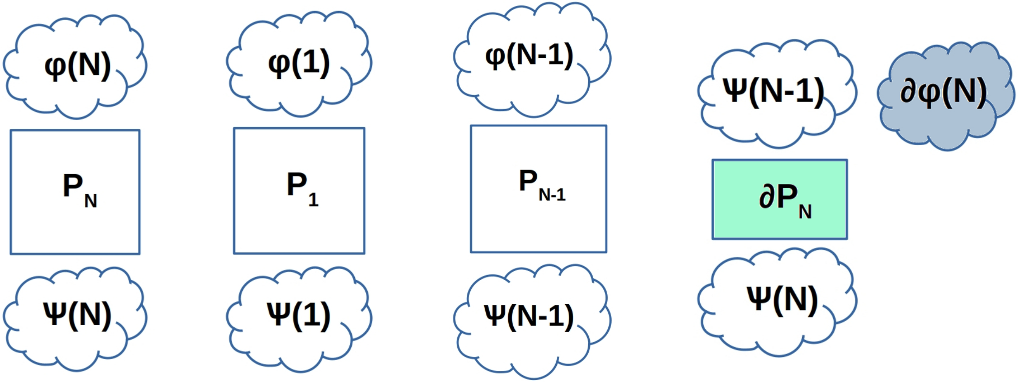
Base Case

Induction Hypothesis

Full-Program Induction (FPI)



Full-Program Induction (FPI)



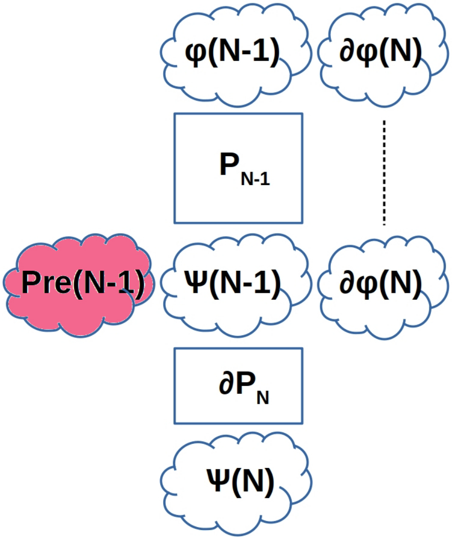
Goal

Base Case

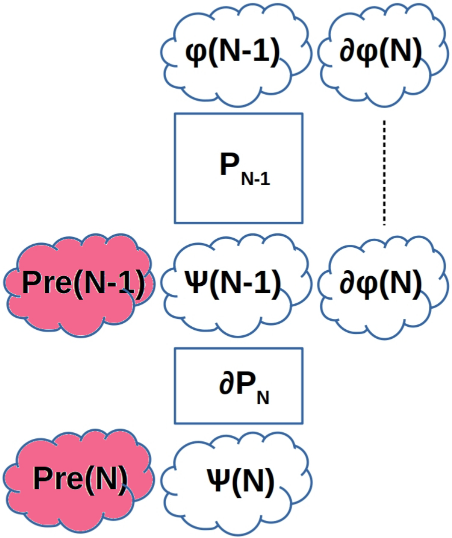
Induction Hypothesis

Induction Step

Strengthening Pre- and Post-conditions



Strengthening Pre- and Post-conditions



A Simple Example

A Simple Example

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

```
assume(∀x∈[0,N) A[x]=*)

void simple(int A[], int N) {
 int S=0;

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

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

assert(S=N)
```

A Simple Example

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

```
assume(∀x∈[0,N) A[x]=*)

1. void simple(int A[], int N) {
2.  int S=0;

3.  for (int i=0; i<N; i++)
4.    A[i] = 1;

5.  for (int j=0; j<N; j++)
6.    S = S + A[j];
7. }

assert(S=N)
```

Base-case: Substitute N=1

```
assume(∀x∈[0,1) A[x]=*)

1. void simple(int A[], int N) {
2.  int S=0;

3.  for (int i=0; i<1; i++)
4.    A[i] = 1;

5.  for (int j=0; j<1; j++)
6.    S = S + A[j];
7. }

assert(S=1)
```

A Simple Example

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

```
assume(∀x∈[0,N) A[x]=*)

void simple(int A[], int N) {
 int S=0;

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

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

assert(S=N)
```

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

```
assume(∀x∈[0,N-1) A[x]=*)

void simple(int A[], int N) {
 int S=0;

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

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

assert(S=N-1)
```

A Simple Example

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

```
assume(∀x∈[0,N) A[x]=*)

1. void simple(int A[], int N) {
2.  int S=0;

3.  for (int i=0; i<N; i++)
4.    A[i] = 1;

5.  for (int j=0; j<N; j++)
6.    S = S + A[j];
7. }

assert(S=N)
```

To prove

```
assume(∀x∈[0,N) A[x]=*)    // φ(N)

1. void simple(int A[], int N) {
2.  int S=0;

3.  for (int i=0; i<N-1; i++)
4.    A[i] = 1;
4a. A[N-1] = 1;         // Peeled iteration

5.  for (int j=0; j<N-1; j++)
6.    S = S + A[j];
7. }
7a. S = S + A[N-1];    // Peeled iteration

assert(S=N)      // ψ(N)
```

A Simple Example

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

```
assume(∀x∈[0,N) A[x]=*)

void simple(int A[], int N) {
 int S=0;

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

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

assert(S=N)
```

To prove

```
assume(∀x∈[0,N) A[x]=*)    // φ(N)

void simple(int A[], int N) {
 int S=0;

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

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

A[N-1] = 1;          // Peeled iteration
S = S + A[N-1];      // Peeled iteration

assert(S=N)      // ψ(N)
```

A Simple Example

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

```
assume(∀x∈[0,N) A[x]=*)

void simple(int A[], int N) {
 int S=0;

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

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

assert(S=N)
```

Inductive Step

```
assume(∀x∈[0,N-1) A[x]=*)     // φ(N-1)

void simple(int A[], int N) {
 int S=0;

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

 for (int j=0; j<N-1; j++)
   S = S + A[j];
}
```

P_{N-1}

```
assume(S=N-1)      // ψ(N-1)
assume(A[N-1]=*)     // ∂φ(N)

 A[N-1] = 1;
 S = S + A[N-1];
```

∂P_N

```
assert(S=N)       // ψ(N)
```

Complication in Computing Difference Program

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

```
assume(∀x∈[0,N) A[x]=*)

void affected(int A[], int N) {
 int S = 0;

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

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

assert(S=N²)
```

Complication in Computing Difference Program

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

```
assume(∀x∈[0,N) A[x]=*)

1. void affected(int A[], int N) {
2.  int S = 0;

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

5.  for (int j=0; j<N; j++)
6.    S = S + A[j];
7. }

assert(S=N^2)
```

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

```
assume(∀x∈[0,N-1) A[x]=*)

1. void affected(int A[], int N) {
2.  int S=0;

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

5.  for (int j=0; j<N-1; j++)
6.    S = S + A[j];
7. }

assert(S=(N-1)^2)
```

Complication in Computing Difference Program

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

```
assume(∀x∈[0,N) A[x]=*)

1. void affected(int A[], int N) {
2.  int S = 0;

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

5.  for (int j=0; j<N; j++)
6.    S = S + A[j];
7. }

assert(S=N^2)
```

Does peeling work here?

```
assume(∀x∈[0,N-1) A[x]=*)    // φ(N-1)

1. void affected(int A[], int N) {
2.  int S=0;
3.  for (int i=0; i<N-1; i++)
4.    A[i] = N-1;
5.  for (int j=0; j<N-1; j++)
6.    S = S + A[j];
7. }

// A[i] incorrect for i∈[0,N-1)

8. A[N-1] = N;

// Value of S incorrect

9. S = S + A[N-1];

assert(S=N^2)      // ψ(N)
```

Complication in Computing Difference Program

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

```
assume(∀x∈[0,N) A[x]=*)

void affected(int A[], int N) {
 int S = 0;

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

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

assert(S=N^2)
```

Does peeling work here?

```
assume(∀x∈[0,N-1) A[x]=*)    // φ(N − 1)

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

// A[i] incorrect for i∈[0,N-1)
// A[i] data-dependent on N, hence ‘affected’
A[N-1] = N;

// Value of S incorrect
// S data-dependent on ‘affected’ A[i], hence ‘affected’
S = S + A[N-1];

assert(S=N^2)      // ψ(N)
```

Complication in Computing Difference Program

Inductive Step

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

```
assume(∀x∈[0,N) A[x]=*)

1. void affected(int A[], int N) {
2.  int S = 0;

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

5.  for (int j=0; j<N; j++)
6.    S = S + A[j];
7. }

assert(S=N²)
```

```
assume(∀x∈[0,N-1) A[x]=*)     // φ(N − 1)

1. void affected(int A[], int N) {
2.  int S=0;
3.  for (int i=0; i<N-1; i++)
4.    A[i] = N-1;
5.  for (int j=0; j<N-1; j++)
6.    S = S + A[j];
7. }
```

P_{N-1}

```
assume(S=(N-1)²)     // ψ(N − 1)
assume(A[N-1]=*)     // ∂φ(N)

8.  for (int i=0; i<N-1; i++)
9.    A[i] = A[i] + 1;
10. A[N-1] = N;
11. for (int j=0; j<N-1; j++)
12.   S = S + 1;
13. S = S + A[N-1];
```

$\partial\mathsf{P}'_N$

```
assert(S=N²)       // ψ(N)
```

Complication in Computing Difference Program

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

```
assume(∀x∈[0,N) A[x]=*)

void affected(int A[], int N) {
 int S = 0;

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

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

assert(S=N²)
```

Inductive Step

```
assume(∀x∈[0,N-1) A[x]=*)     // φ(N − 1)

void affected(int A[], int N) {
 int S=0;

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

 for (int j=0; j<N-1; j++)
   S = S + A[j];
}
```

P_{N-1}

```
assume(S=(N-1)²)     // ψ(N − 1)
assume(A[N-1]=*)     // ∂φ(N)

 A[N-1] = N;
 S = S + N-1;
 S = S + A[N-1];
```

$\partial\mathsf{P}_N$

```
assert(S=N²)       // ψ(N)
```

Correctness of the Computed Differences

We have devised an algorithm to compute $\partial \mathsf{P}_N$

Theorem

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

Correctness of the Computed Differences

We have devised an algorithm to compute $\partial \mathsf{P}_N$

Theorem

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

We have devised an algorithm to compute $\partial\varphi(N)$

Theorem

Computed formula $\partial\varphi(N)$ satisfies the following conditions:

(a) $\varphi(N) \to \varphi(N-1) \wedge \partial\varphi(N)$

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

Soundness Guarantee

Theorem

Consider a formula $\mathrm{Pre}(M)$ *for* $M \geq 1$ *such that*

Soundness Guarantee

Theorem

Consider a formula $\mathsf{Pre}(M)$ *for* $M \geq 1$ *such that*

(a) $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ *for* $0 < N \leq M$

Soundness Guarantee

Theorem

Consider a formula $\mathrm{Pre}(M)$ *for* $M \geq 1$ *such that*

(a) $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ *for* $0 < N \leq M$

(b) $\{\varphi(M)\}\ \mathsf{P}_M\ \{\psi(M) \wedge \mathrm{Pre}(M)\}$

Soundness Guarantee

Theorem

Consider a formula $\mathrm{Pre}(M)$ *for* $M \geq 1$ *such that*

(a) $\{\varphi(N)\}\ \mathrm{P}_N\ \{\psi(N)\}$ *for* $0 < N \leq M$

(b) $\{\varphi(M)\}\ \mathrm{P}_M\ \{\psi(M) \land \mathrm{Pre}(M)\}$

(c) $\{\partial\varphi(N) \land \psi(N-1) \land \mathrm{Pre}(N-1)\}\ \partial\mathrm{P}_N\ \{\psi(N) \land \mathrm{Pre}(N)\}$ *for* $N > M$

Soundness Guarantee

Theorem

Consider a formula $\mathsf{Pre}(M)$ *for* $M \geq 1$ *such that*

(a) $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ *for* $0 < N \leq M$

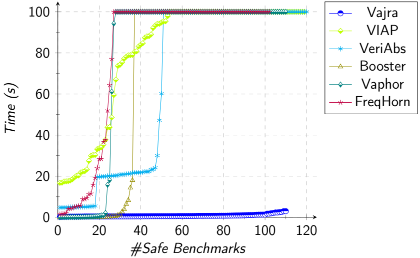
(b) $\{\varphi(M)\}\ \mathsf{P}_M\ \{\psi(M) \wedge \mathsf{Pre}(M)\}$

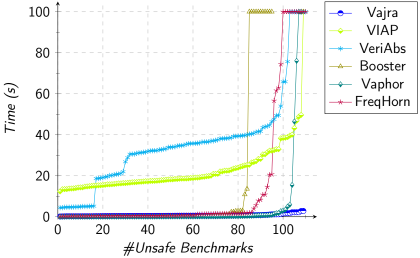
(c) $\{\partial\varphi(N) \wedge \psi(N-1) \wedge \mathsf{Pre}(N-1)\}\ \partial\mathsf{P}_N\ \{\psi(N) \wedge \mathsf{Pre}(N)\}$ *for* $N > M$

Then $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ *holds for all* $N \geq 1$.

Vajra - Prototyping FPI

Tool	Success	CE	Inconclusive	TO
		SAFE		
Vajra	110	0	11	0
VIAP	58	0	2	61
VeriAbs	50	1	0	70
Booster	36	27	17	41
VapHor	27	9	2	83
FreqHorn	26	0	19	76
		UNSAFE		
Vajra	0	109	1	0
VIAP	1	108	0	1
VeriAbs	0	102	0	8
Booster	0	84	15	11
VapHor	1	106	1	2
FreqHorn	0	99	0	11





Vajra Participated in SV-COMP 2020 and later editions

- **Vajra** integrated into TCS verification tool VeriAbs
  - Bundled with VeriAbs **v1.4** as a part of its SV-COMP 2020 archive

Vajra Participated in SV-COMP 2020 and later editions

- **Vajra** integrated into TCS verification tool VeriAbs
  - Bundled with VeriAbs **v1.4** as a part of its SV-COMP 2020 archive

- "**Gold** ● **Medal**" in the Reach-safety category
  - Vajra improved the score in *Arrays sub-category*
  - **1$^{\text{st}}$** place in 2020, with 694/759 points, solved 410/436 programs
    - Map2Check - 2nd place in 2020, with 379/759 points
  - **2$^{\text{nd}}$** place in 2019, with 365/418 points, solved 196/231 programs
  - Vajra solved **158** additional programs in *Arrays sub-category*

Challenging Benchmarks Solved by Full-Program Induction

```
assume(true)

1. int A[N], B[N], C[N];
2. A[0]=6;  B[0]=1;  C[0]=0;

3. for (int i=1; i<N; i++)
4.   A[i] = A[i-1] + 6;

5. for (int j=1; j<N; j++)
6.   B[j] = B[j-1] + A[j-1];

7. for (int k=1; k<N; k++)
8.   C[k] = C[k-1] + B[k-1];

assert(∀x∈[0,N) C[x]=x^3)
```

- Non-linear loop invariants needed
- Inter-loop data dependence
- Difference program: peeled iterations

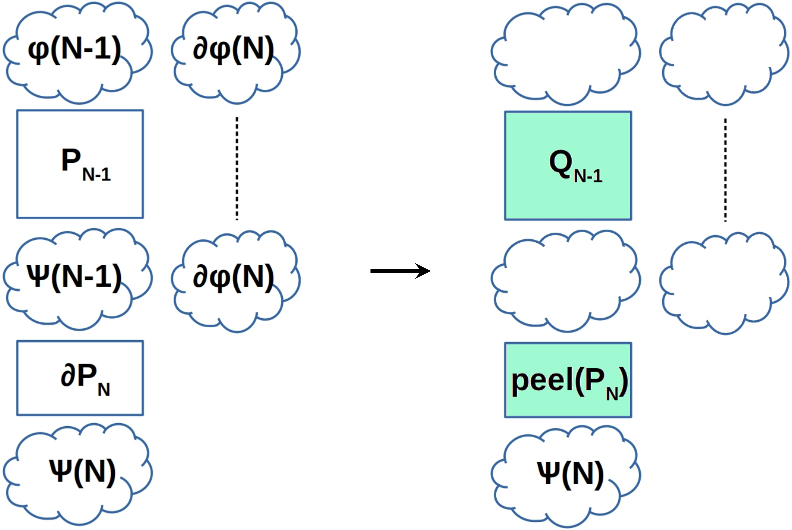
Challenging Benchmarks Solved by Full-Program Induction

```
assume(∀x∈[0,N) A[x] = 1)

S = 0;
for(i=0; i<N; i++) {
  S = S + A[i];
}

for(j=0; j<N; j++) {
  A1[j] = A[j] + S*j;
}

S1 = S;
for(k=0; k<N; k++) {
  S1 = S1 + A1[k]*k;
}

assert(S1 = (2N^4-3N^3+4N^2+3N)/6)
```

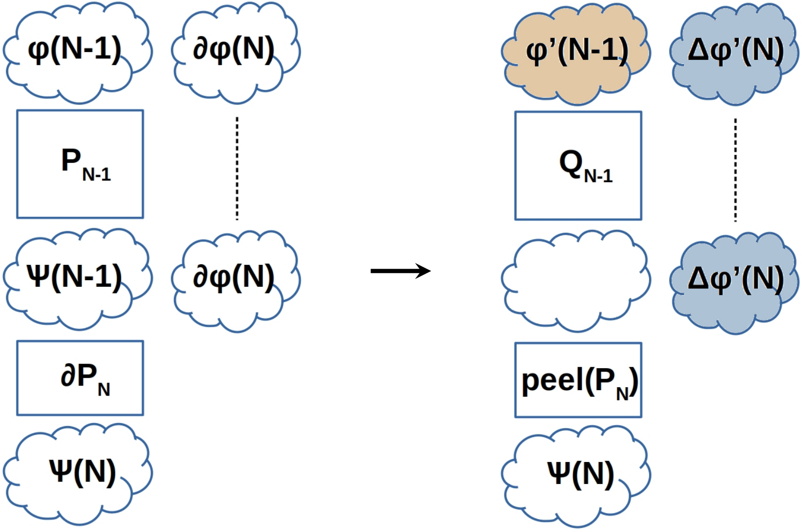
- Non-trivial non-linear loop invariants needed
- Inter and intra-loop data dependence
- Aggregation of array content
- Difference program: beyond peeled iterations

Relational Full-Program Induction

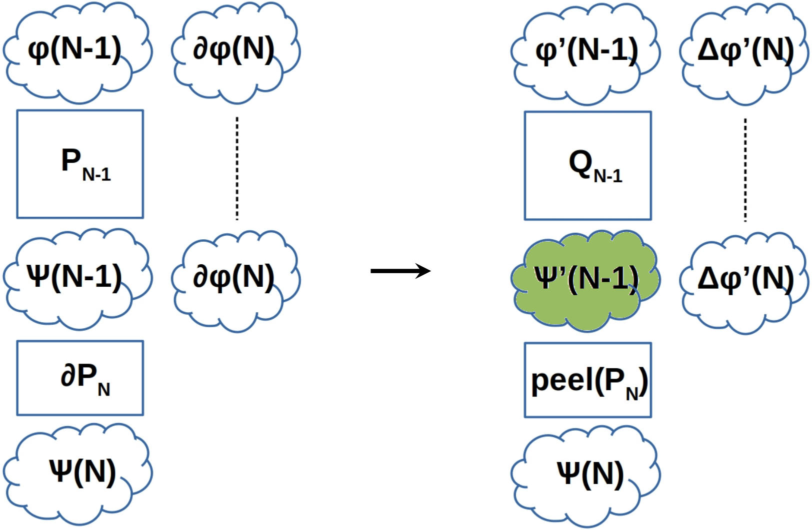
Relational Full-Program Induction (RFPI)



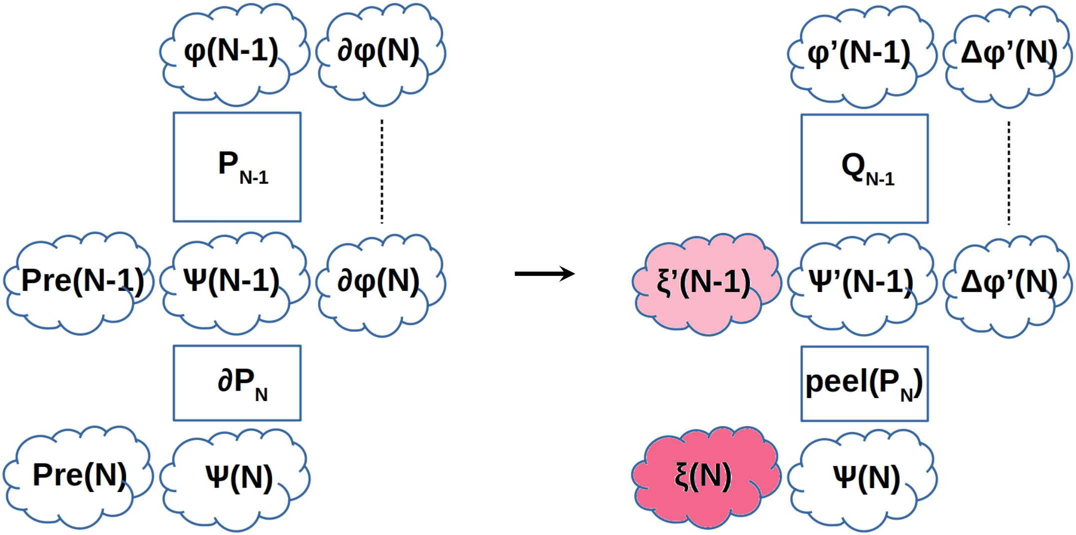
Relational Full-Program Induction (RFPI)



Relational Full-Program Induction (RFPI)



Relational Full-Program Induction (RFPI)



Relational Full-Program Induction

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

```
assume(∀x∈[0,N) A[x]=*)

void affected(int A[], int N) {
 int S = 0;

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

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

assert(S=N²)
```

Relational Full-Program Induction

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

```
assume(∀x∈[0,N-1) A[x]=*)

void affected(int A[], int N) {
 int S=0;

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

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

assert(S=(N-1)^2)
```

Relational Full-Program Induction

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

```
assume(∀x∈[0,N-1) A[x]=*)

void affected(int A[], int N) {
 int S=0;

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

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

assert(S=(N-1)^2)
```

RFPI Inductive Step

```
assume(∀x∈[0,N-1) A[x]=*)     // φ(N − 1)

void affected(int A[], int N) {
 int S=0;

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

 for (int j=0; j<N-1; j++)
   S = S + A[j];
}
```

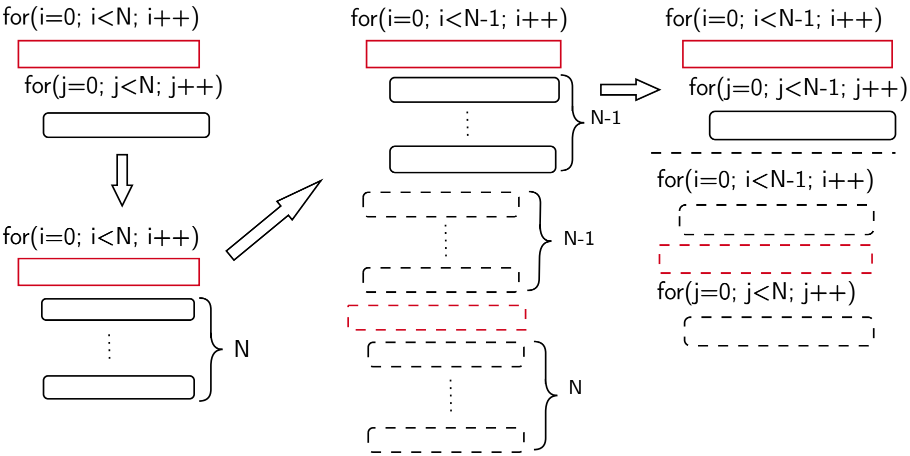
Q_{N-1}

```
assume(S=N^2-N)     // ψ'(N − 1)
assume(A[N-1]=*)     // Δφ(N)

 A[N-1] = N;
 S = S + A[N-1];
```

$\text{peel}(\mathsf{P}_N)$

```
assert(S=N^2)      // ψ(N)
```

Handling Nested Loops



Correctness of Program Transformations

We have devised an algorithm to compute Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$

Theorem

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

Reduction in Verification Complexity

Theorem

For a class of programs where upper bounds of loops are linear in N & loop counters

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

Soundness of Relational Full-Program Induction

Theorem

Suppose

1) $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N) \wedge \xi(N)\}$ *holds for* $1 \leq N \leq M$*, for some* $M > 0$

Soundness of Relational Full-Program Induction

Theorem

Suppose

1) $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N) \wedge \xi(N)\}$ *holds for* $1 \leq N \leq M$, *for some* $M > 0$

2) $\xi(N) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}) \Rightarrow \xi'(N)$ *holds for all* $N > 0$

Soundness of Relational Full-Program Induction

Theorem

Suppose

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

Soundness of Relational Full-Program Induction

Theorem

Suppose

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

Then $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ **holds** *for all* $N > 0$

Relative Completeness

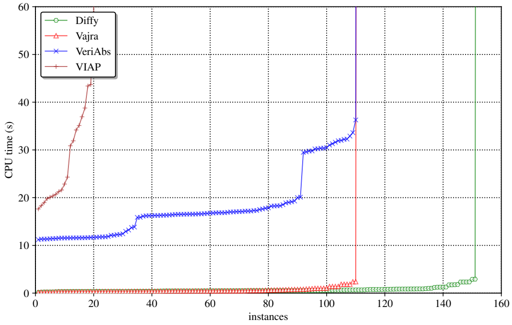
Theorem

RFPI is `sound and relatively complete` *for post-conditions when*

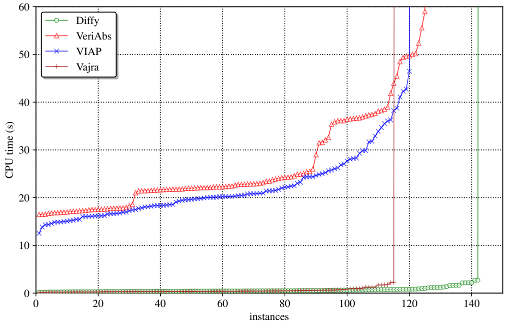
- *programs have only non-nested loops*
- *variables and arrays computed in the program are not affected*

Completeness is relative to the capabilites of the underlying SMT solver

Diffy - Implementing RFPI



(a)



(b)

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

- ⋆ Diffy is 10× faster than VIAP, VeriAbs; proves more benchmarks than Vajra
- ⋆ Violations are reported by simple bmc when base case fails
- ⋆ Diffy incorporated in VeriAbs since SV-COMP 2022

Limitations of (R)FPI

Verifying programs with side-effects

- Updates to shared resources
- Updates to global variables/heap

Precision of affected variables analysis

Computation of difference artifacts for more general classes of programs

...

Conclusion

- ★ Novel perspectives to inductive reasoning
- ★ Adapted induction in ways different from classical methods
- ★ Contributed 3 new verification techniques - Tiling, FPI, RFPI
- ★ Prototyped in the verification tools Tiler, Vajra, Diffy
- ★ Outperforms state-of-the-art tools and techniques on large benchmark suites
- ★ All tools incorporated in VeriAbs - a portfolio verifier from TCS Research

Future Prospects

★ Applications to compiler optimizations such as incrementalization, loop fusion

Future Prospects

★ Applications to compiler optimizations such as incrementalization, loop fusion

★ Automated synthesis techniques for generation of $\partial \mathsf{P}_N$ and relational invariants

Future Prospects

- ★ Applications to compiler optimizations such as incrementalization, loop fusion
- ★ Automated synthesis techniques for generation of $\partial \mathsf{P}_N$ and relational invariants
- ★ Integration of 'simple' program fragments to compute a database of $\partial \mathsf{P}_N$

Future Prospects

- ★ Applications to compiler optimizations such as incrementalization, loop fusion
- ★ Automated synthesis techniques for generation of $\partial \mathrm{P}_N$ and relational invariants
- ★ Integration of 'simple' program fragments to compute a database of $\partial \mathrm{P}_N$
- ★ Expand scope of inductive reasoning to data-structures beyond arrays

Future Prospects

- ★ Applications to compiler optimizations such as incrementalization, loop fusion
- ★ Automated synthesis techniques for generation of $\partial \mathsf{P}_N$ and relational invariants
- ★ Integration of 'simple' program fragments to compute a database of $\partial \mathsf{P}_N$
- ★ Expand scope of inductive reasoning to data-structures beyond arrays
- ★ Support for larger classes of programs and properties