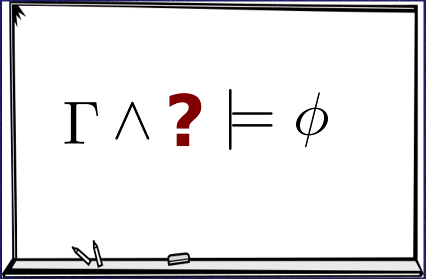
Abductive Inference and its Applications in Program Analysis, Verification, and Synthesis



Isil Dillig
University of Texas, Austin

** Joint work with Ken McMillan, Tom Dillig, Boyang Li, Alex Aiken, Mooly Sagiv*

What is Abduction?

- Abduction: Inference of missing hypotheses

What is Abduction?

- Abduction: Inference of missing hypotheses
- Given known facts Γ and desired outcome ϕ, abductive inference finds "simple" explanatory hypothesis ψ such that

$$\Gamma \land \psi \models \phi \quad \text{and} \quad \mathrm{SAT}(\Gamma \land \psi)$$

What is Abduction?

- Abduction: Inference of missing hypotheses
- Given known facts Γ and desired outcome ϕ, abductive inference finds "simple" explanatory hypothesis ψ such that

$$\Gamma \wedge \psi \models \phi \quad \text{and} \quad \mathrm{SAT}(\Gamma \wedge \psi)$$

- i.e., given invalid formula $\Gamma \Rightarrow \phi$, find a "simple" formula ψ such that $\Gamma \wedge \psi \Rightarrow \phi$ is valid and ψ does not contradict Γ.

Simple Example

- Premises: “If it rains, then it is wet and cloudy”, “If it is wet, then it is slippery”: $(R \Rightarrow W \wedge C) \wedge (W \Rightarrow S)$

Simple Example

- Premises: "If it rains, then it is wet and cloudy", "If it is wet, then it is slippery": $(R \Rightarrow W \wedge C) \ \wedge (W \Rightarrow S)$
- Conclusion: "It is cloudy and slippery", i.e., $C \wedge S$

Simple Example

- Premises: “If it rains, then it is wet and cloudy”, “If it is wet, then it is slippery”: $(R \Rightarrow W \wedge C) \wedge (W \Rightarrow S)$
- Conclusion: “It is cloudy and slippery”, i.e., $C \wedge S$
- Conclusion doesn’t follow from premises; use abduction to find missing hypothesis

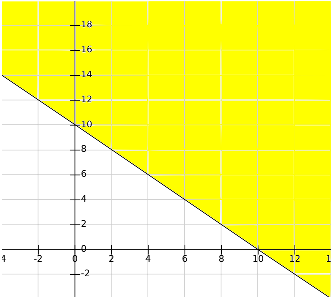
Simple Example

- Premises: "If it rains, then it is wet and cloudy", "If it is wet, then it is slippery": $(R \Rightarrow W \land C) \land (W \Rightarrow S)$
- Conclusion: "It is cloudy and slippery", i.e., $C \land S$
- Conclusion doesn't follow from premises; use abduction to find missing hypothesis
- Possible solution: R, i.e., "It is rainy"

Arithmetic Example

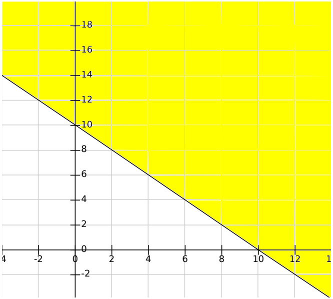
- Suppose we know $x \geq -2$

Arithmetic Example



- Suppose we know $x \geq -2$
- Want to prove: $x + y > 10$

Arithmetic Example



- Suppose we know $x \geq -2$
- Want to prove: $x + y > 10$
- Abductive explanation: $y > 12$

Outline of Talk

1. Properties of desired solutions

Outline of Talk

1. Properties of desired solutions
2. Algorithm for performing abduction in LRA/LIA

Outline of Talk

1. Properties of desired solutions
2. Algorithm for performing abduction in LRA/LIA
3. Loop invariant generation using abduction

Outline of Talk

1. Properties of desired solutions
2. Algorithm for performing abduction in LRA/LIA
3. Loop invariant generation using abduction
4. Compositional verification using abduction

Outline of Talk

1. Properties of desired solutions
2. Algorithm for performing abduction in LRA/LIA
3. Loop invariant generation using abduction
4. Compositional verification using abduction
5. Use of abduction in program synthesis

Outline of Talk

1. Properties of desired solutions
2. Algorithm for performing abduction in LRA/LIA
3. Loop invariant generation using abduction
4. Compositional verification using abduction
5. Use of abduction in program synthesis
6. Conclusion and future directions

Properties of Desired Solutions

- In general, the abduction problem $\Gamma \wedge ? \models \phi$ has infinitely many solutions

Properties of Desired Solutions

- In general, the abduction problem $\Gamma \wedge ? \models \phi$ has infinitely many solutions
- Trivial solution: ϕ, but not useful because does not take into account what we know

Properties of Desired Solutions

- In general, the abduction problem $\Gamma \wedge ? \models \phi$ has infinitely many solutions
- Trivial solution: ϕ, but not useful because does not take into account what we know
- So, what kind of solutions do want to compute?

Which Abductive Explanations Are Good?

Guiding Principle:
Occam's Razor

Which Abductive Explanations Are Good?

Guiding Principle:
Occam's Razor

- If there are multiple competing hypotheses, select the one that makes fewest assumptions

Which Abductive Explanations Are Good?

Guiding Principle:
Occam's Razor

- If there are multiple competing hypotheses, select the one that makes fewest assumptions
- Generality: If explanation A is logically weaker than explanation B, always prefer A

Which Abductive Explanations Are Good?

Guiding Principle:
Occam's Razor

- If there are multiple competing hypotheses, select the one that makes fewest assumptions
- Generality: If explanation A is logically weaker than explanation B, always prefer A
- Succinctness: Minimize number of variables

Solutions We Will Compute

Want to compute logically weakest solutions with fewest variables

Solutions We Will Compute

Want to compute logically weakest solutions with fewest variables

- First talk about how to compute solutions with fewest variables

Solutions We Will Compute

Want to compute logically weakest solutions with fewest variables

- First talk about how to compute solutions with fewest variables
- Then talk about how to obtain most general solution containing these variables

Minimum Satisfying Assignments

To find solutions with fewest variables, we use minimum satisfying assignments of formulas

Minimum Satisfying Assignments

To find solutions with fewest variables, we use minimum satisfying assignments of formulas

Minimum satisfying assignment (MSA):

- ✓ assigns values to a subset of variables in formula
- ✓ sufficient to make formula true
- ✓ Among all other partial satisfying assignments, contains fewest variables

Example

- Consider the following formula in linear integer arithmetic:

$$x + y + w > 0 \lor x + y + z + w < 5$$

Example

- Consider the following formula in linear integer arithmetic:

$$x + y + w > 0 \vee x + y + z + w < 5$$

- Minimum satisfying assignment: $z = 0$

Example

- Consider the following formula in linear integer arithmetic:

$$x + y + w > 0 \vee x + y + z + w < 5$$

- Minimum satisfying assignment: $z = 0$
- Note: Algorithm for computing MSAs given in our CAV'12 paper, "Minimum Satisfying Assignments for SMT"

Why Are MSAs Useful for Abduction?

- Given facts Γ and conclusion ϕ, MSA σ of $\Gamma \Rightarrow \phi$ consistent with Γ is a solution to abduction problem:

$$\sigma \models \Gamma \Rightarrow \phi \quad \text{hence} \quad \sigma \wedge \Gamma \models \phi$$

Why Are MSAs Useful for Abduction?

- Given facts Γ and conclusion ϕ, MSA σ of $\Gamma \Rightarrow \phi$ consistent with Γ is a solution to abduction problem:

$$\sigma \models \Gamma \Rightarrow \phi \quad \text{hence} \quad \sigma \wedge \Gamma \models \phi$$

- Furthermore, it uses a fewest number of variables

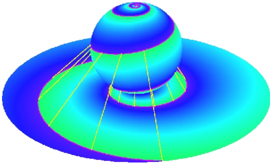
Why Are MSAs Useful for Abduction?

- Given facts Γ and conclusion ϕ, MSA σ of $\Gamma \Rightarrow \phi$ consistent with Γ is a solution to abduction problem:

$$\sigma \models \Gamma \Rightarrow \phi \quad \text{hence} \quad \sigma \wedge \Gamma \models \phi$$

- Furthermore, it uses a fewest number of variables
- But it is not the most general solution

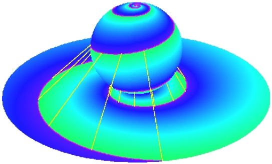
Finding Most General Solutions



Key idea:
Quantifier elimination

- To find most general solution containing variables in the MSA, universally quantify all other variables $\overline{V}$ and apply quantifier elimination to $\forall \overline{V}.\ \Gamma \Rightarrow \phi$

Finding Most General Solutions



Key idea:
Quantifier elimination

- To find most general solution containing variables in the MSA, universally quantify all other variables $\overline{V}$ and apply quantifier elimination to $\forall \overline{V}.\ \Gamma \Rightarrow \phi$
- This yields most general solution with fewest variables

Abduction Algorithm

- abduce yields formula ψ such that

$$\Gamma \wedge \psi \models \phi$$

and ψ is consistent with Γ and θ

$\text{abduce}(\Gamma, \phi, \theta)\{$

$\}$

Abduction Algorithm

- abduce yields formula ψ such that

$$\Gamma \wedge \psi \models \phi$$

  and ψ is consistent with Γ and θ

- First, compute all variables in MSA of $\Gamma \Rightarrow \phi$ consistent with Γ, θ

$$
\begin{array}{l}
\mathrm{abduce}(\Gamma, \phi, \theta)\{ \\
\quad V = \mathrm{msa}(\Gamma \Rightarrow \phi, \theta \cup \{\Gamma\}) \\
\\
\}
\end{array}
$$

Abduction Algorithm

- abduce yields formula ψ such that

$$\Gamma \wedge \psi \models \phi$$

  and ψ is consistent with Γ and θ

- First, compute all variables in MSA of $\Gamma \Rightarrow \phi$ consistent with Γ, θ

- $\forall$-quantify variables not in the MSA and apply quantifier elimination

$\text{abduce}(\Gamma, \phi, \theta)\{$

$\quad V = \text{msa}(\Gamma \Rightarrow \phi, \theta \cup \{\Gamma\})$

$\quad \psi = \text{QE}(\forall \overline{V}.(\Gamma \Rightarrow \phi))$

$\}$

Abduction Algorithm

- abduce yields formula ψ such that

$$\Gamma \wedge \psi \models \phi$$

  and ψ is consistent with Γ and θ

- First, compute all variables in MSA of $\Gamma \Rightarrow \phi$ consistent with Γ, θ
- $\forall$-quantify variables not in the MSA and apply quantifier elimination
- Remove subparts of ψ implied or contradicted by Γ (SAS'10)

$$
\begin{array}{l}
\text{abduce}(\Gamma, \phi, \theta)\{ \\
\quad V = \text{msa}(\Gamma \Rightarrow \phi, \theta \cup \{\Gamma\}) \\
\quad \psi = \text{QE}(\forall \overline{V}.(\Gamma \Rightarrow \phi)) \\
\quad \psi' = \text{simplify}(\psi, \Gamma) \\
\}
\end{array}
$$

Abduction Algorithm

- abduce yields formula ψ such that

$$\Gamma \wedge \psi \models \phi$$

  and ψ is consistent with Γ and θ

- First, compute all variables in MSA of $\Gamma \Rightarrow \phi$ consistent with Γ, θ
- $\forall$-quantify variables not in the MSA and apply quantifier elimination
- Remove subparts of ψ implied or contradicted by Γ (SAS'10)

$$
\begin{array}{l}
\text{abduce}(\Gamma, \phi, \theta)\{ \\
\quad V = \text{msa}(\Gamma \Rightarrow \phi, \theta \cup \{\Gamma\}) \\
\quad \psi = \text{QE}(\forall \overline{V}.(\Gamma \Rightarrow \phi)) \\
\quad \psi' = \text{simplify}(\psi, \Gamma) \\
\quad \text{return } \psi' \\
\}
\end{array}
$$

Simple Example

- Consider abduction problem in LIA defined by conclusion $\phi: \; x + y > 10$ and known facts $\Gamma : x \geq -2 \wedge x = y$

Simple Example

- Consider abduction problem in LIA defined by conclusion $\phi: \; x + y > 10$ and known facts $\Gamma : x \geq -2 \land x = y$
- Compute MSA for $\Gamma \Rightarrow \phi$:

Simple Example

- Consider abduction problem in LIA defined by conclusion $\phi : \; x + y > 10$ and known facts $\Gamma : x \geq -2 \wedge x = y$
- Compute MSA for $\Gamma \Rightarrow \phi$: $y = 15$

Simple Example

- Consider abduction problem in LIA defined by conclusion $\phi: \; x + y > 10$ and known facts $\Gamma : x \geq -2 \land x = y$
- Compute MSA for $\Gamma \Rightarrow \phi$: $y = 15$
- Universally quantify x and eliminate from $\Gamma \Rightarrow \phi$:

Simple Example

- Consider abduction problem in LIA defined by conclusion $\phi: \; x + y > 10$ and known facts $\Gamma : x \geq -2 \wedge x = y$
- Compute MSA for $\Gamma \Rightarrow \phi$: $y = 15$
- Universally quantify x and eliminate from $\Gamma \Rightarrow \phi$: $y < -2 \vee y > 5$

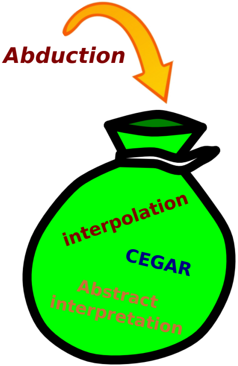
Simple Example

- Consider abduction problem in LIA defined by conclusion $\phi: \; x + y > 10$ and known facts $\Gamma : x \geq -2 \wedge x = y$
- Compute MSA for $\Gamma \Rightarrow \phi$: $y = 15$
- Universally quantify x and eliminate from $\Gamma \Rightarrow \phi$: $y < -2 \vee y > 5$
- Simplify with respect to assumptions:

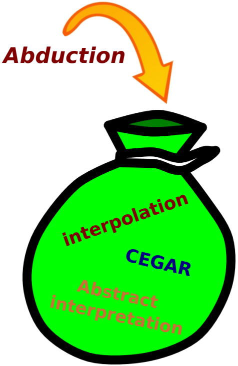
Simple Example

- Consider abduction problem in LIA defined by conclusion $\phi: \ x + y > 10$ and known facts $\Gamma : x \geq -2 \wedge x = y$
- Compute MSA for $\Gamma \Rightarrow \phi$: $y = 15$
- Universally quantify x and eliminate from $\Gamma \Rightarrow \phi$: $y < -2 \vee y > 5$
- Simplify with respect to assumptions: $y > 5$

Applications

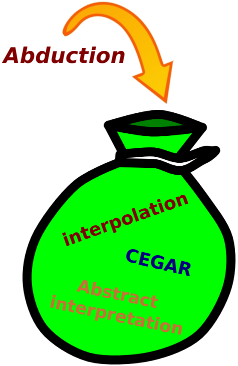


Applications



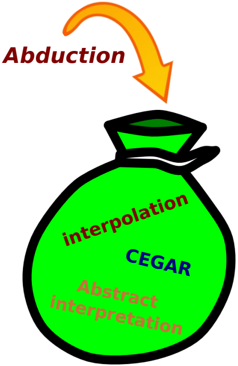
- Loop invariant generation (OOPSLA'13)

Applications



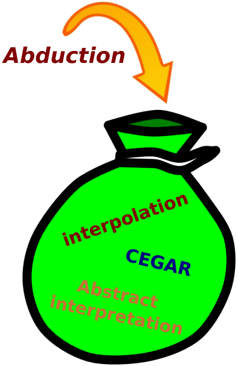
- Loop invariant generation (OOPSLA'13)
- Compositional program verification (TACAS'13)

Applications



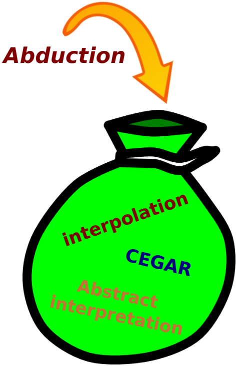
- Loop invariant generation (OOPSLA'13)
- Compositional program verification (TACAS'13)
- Inference of missing library specifications (APLAS'13)

Applications



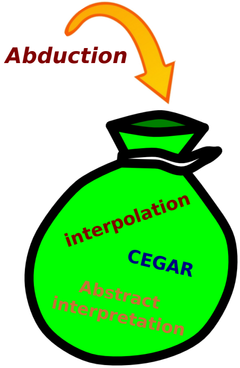
- Loop invariant generation (OOPSLA'13)
- Compositional program verification (TACAS'13)
- Inference of missing library specifications (APLAS'13)
- Diagnosis of static analysis warnings (PLDI'12)

Applications



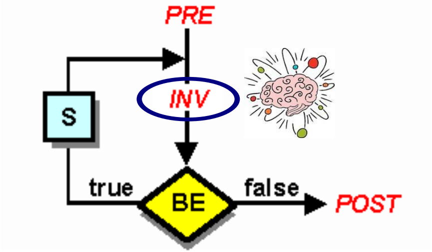
- Loop invariant generation (OOPSLA'13)
- Compositional program verification (TACAS'13)
- Inference of missing library specifications (APLAS'13)
- Diagnosis of static analysis warnings (PLDI'12)
- Synthesis of missing guards (CAV'14)

Applications

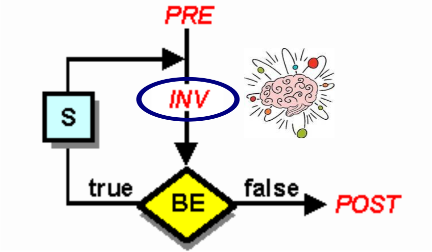


- Loop invariant generation (OOPSLA'13)
- Compositional program verification (TACAS'13)
- Inference of missing library specifications (APLAS'13)
- Diagnosis of static analysis warnings (PLDI'12)
- Synthesis of missing guards (CAV'14)

Application #1: Loop Invariant Generation

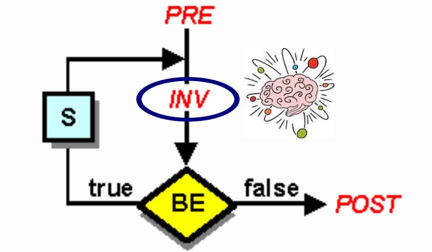


Application #1: Loop Invariant Generation



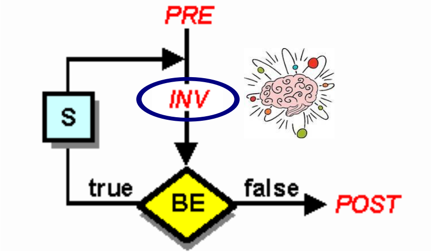
- Most challenging aspect of program verification: **loop invariant generation**

Application #1: Loop Invariant Generation



- Most challenging aspect of program verification: **loop invariant generation**
- Inductive loop invariant **Inv** is implied by **Pre** and preserved in each iteration assuming only **Inv**

Application #1: Loop Invariant Generation



- Most challenging aspect of program verification: **loop invariant generation**
- Inductive loop invariant **Inv** is implied by **Pre** and preserved in each iteration assuming only **Inv**
- But **Inv** is only useful if it is sufficient to prove **Post**

High Level Idea

Key idea 1: Given loop L and postcondition Post, use **abduction** to speculate candidate invariants

High Level Idea

Key idea 1: Given loop L and postcondition Post, use **abduction** to speculate candidate invariants

```
assume(P);
while(C) {
   S;
}
assert(Q);
```

High Level Idea

Key idea 1: Given loop L and postcondition Post, use **abduction** to speculate candidate invariants

- Use abduction to speculate an invariant I that implies post-condition Q:

$$(\neg C \wedge ?) \Rightarrow Q$$

```
assume(P);
while(C) {
  S;
}
assert(Q);
```

High Level Idea

Key idea 1: Given loop L and postcondition Post, use **abduction** to speculate candidate invariants

```
assume(P);
while(C) {
   S;
}
assert(Q);
```

- Use abduction to speculate an invariant I that implies post-condition Q:

$$(\neg C \wedge ?) \Rightarrow Q$$

- Three possibilities:

Key idea 1: Given loop L and postcondition Post, use **abduction** to speculate candidate invariants

```
assume(P);
while(C) {
  S;
}
assert(Q);
```

- Use abduction to speculate an invariant I that implies post-condition Q:

$$(\neg C \land ?) \Rightarrow Q$$

- Three possibilities:
  - ☺ I is inductive: $I \land C \Rightarrow wp(S, I)$, $P \Rightarrow I$

High Level Idea

Key idea 1: Given loop L and postcondition Post, use **abduction** to speculate candidate invariants

```
assume(P);
while(C) {
   S;
}
assert(Q);
```

- Use abduction to speculate an invariant I that implies post-condition Q:

$$(\neg C \wedge ?) \Rightarrow Q$$

- Three possibilities:
  - ☺ I is inductive: $I \wedge C \Rightarrow wp(S, I), \;\; P \Rightarrow I$
  - 😐 I is an invariant, but not inductive

High Level Idea

Key idea 1: Given loop L and postcondition Post, use **abduction** to speculate candidate invariants

```
assume(P);
while(C) {
   S;
}
assert(Q);
```

- Use abduction to speculate an invariant I that implies post-condition Q:

$$(\neg C \land ?) \Rightarrow Q$$

- Three possibilities:
  - ☺ I is inductive: $I \land C \Rightarrow wp(S, I), \ \ P \Rightarrow I$
  - 😐 I is an invariant, but not inductive
  - ☹ I is not an invariant

Candidate Invariant not Inductive

- If candidate invariant I is not inductive, our algorithm tries to **strengthen** it

Candidate Invariant not Inductive

- If candidate invariant I is not inductive, our algorithm tries to **strengthen** it
- Use abduction again to find a strengthening I' of I:

$$(?? \wedge (I \wedge C)) \Rightarrow wp(S, I)$$

Candidate Invariant not Inductive

- If candidate invariant I is not inductive, our algorithm tries to **strengthen** it
- Use abduction again to find a strengthening I' of I:

$$(?? \land (I \land C)) \Rightarrow wp(S, I)$$

- If I' is an invariant, then so is I, i.e.,

I is inductive **relative to** *I*'

Candidate Invariant not Inductive

- If candidate invariant I is not inductive, our algorithm tries to **strengthen** it
- Use abduction again to find a strengthening I' of I:

$$(?? \land (I \land C)) \Rightarrow wp(S, I)$$

- If I' is an invariant, then so is I, i.e.,

I is inductive **relative to** *I'*

- Now, check if $I \land I'$ is inductive, if not, keep strengthening using (i) proof goes through, or (ii) get contradiction

Simple Example

- Start by solving abduction problem:

$$i > n \wedge ?? \Rightarrow i \geq 1$$

```
{i=1, j = 0, n<5}

while(i <= n) {
  j := j+i;
  i := i+j;
}

{i >= 1}
```

Simple Example

```
{i=1, j = 0, n<5}

while(i <= n) {
  j := j+i;
  i := i+j;
}

{i >= 1}
```

- Start by solving abduction problem:

$$i > n \land ?? \Rightarrow i \geq 1$$

- Algorithm returns solution $i \geq 1$

Simple Example

```
{i=1, j = 0, n<5}

while(i <= n) {
  j := j+i;
  i := i+j;
}

{i >= 1}
```

- Start by solving abduction problem:

$$i > n \land ?? \Rightarrow i \geq 1$$

- Algorithm returns solution $i \geq 1$

- Not inductive; so try strengthening:

$$i \leq n \land i \geq 1 \land ?? \Rightarrow j + 2i \geq 1$$

Simple Example

```
{i=1, j = 0, n<5}

while(i <= n) {
   j := j+i;
   i := i+j;
}

{i >= 1}
```

- Start by solving abduction problem:

$$i > n \land ?? \Rightarrow i \geq 1$$

- Algorithm returns solution $i \geq 1$
- Not inductive; so try strengthening:

$$i \leq n \land i \geq 1 \land ?? \Rightarrow j + 2i \geq 1$$

- Solution is $j \geq -1$, so new candidate invariant:

$$i \geq 1 \land j \geq -1$$

Simple Example

```
{i=1, j = 0, n<5}

while(i <= n) {
  j := j+i;
  i := i+j;
}
{i >= 1}
```

- Start by solving abduction problem:

$$i > n \land ?? \Rightarrow i \geq 1$$

- Algorithm returns solution $i \geq 1$
- Not inductive; so try strengthening:

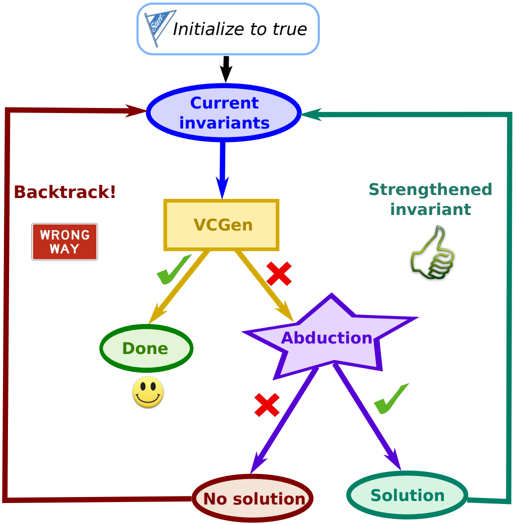
$$i \leq n \land i \geq 1 \land ?? \Rightarrow j + 2i \geq 1$$

- Solution is $j \geq -1$, so new candidate invariant:

$$i \geq 1 \land j \geq -1$$

- This is inductive, so algorithm terminates

Overall Algorithm



How to Perform Backtracking

- Recall: Abduction procedure takes Γ, ϕ, and set θ

$\text{abduce}(\Gamma, \phi, \theta)$

How to Perform Backtracking

- Recall: Abduction procedure takes Γ, ϕ, and set θ
- Solution must be consistent with every $\varphi \in \theta$

$\text{abduce}(\Gamma, \phi, \theta)$

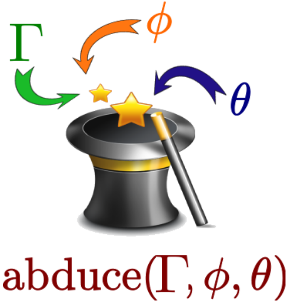
How to Perform Backtracking

- Recall: Abduction procedure takes Γ, ϕ, and set θ
- Solution must be consistent with every $\varphi \in \theta$
- If I is refuted, add $\neg I$ to θ to obtain different solution

$\text{abduce}(\Gamma, \phi, \theta)$

How to Perform Backtracking

- Recall: Abduction procedure takes Γ, ϕ, and set θ
- Solution must be consistent with every $\varphi \in \theta$
- If I is refuted, add $\neg I$ to θ to obtain different solution



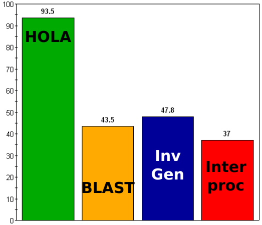
Algorithm lazily generates abductive explanations

Some Experimental Results

- Evaluated this technique on 46 loop invariant benchmarks

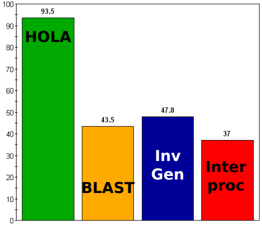
Some Experimental Results

- Evaluated this technique on 46 loop invariant benchmarks
- Compared our results against BLAST, InvGen, and Interproc:



Some Experimental Results

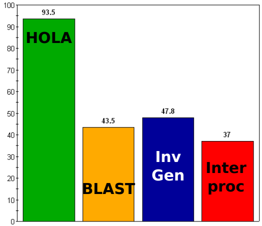
- Evaluated this technique on 46 loop invariant benchmarks
- Compared our results against BLAST, InvGen, and Interproc:



- Can verify 13 benchmarks that no other tool can verify, but cannot prove two benchmarks at least one tool can show

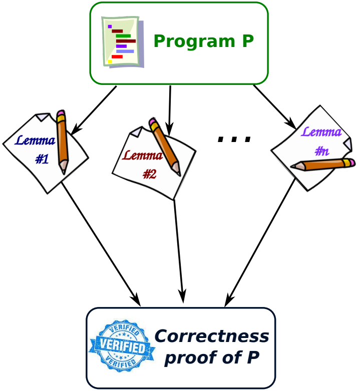
Some Experimental Results

- Evaluated this technique on 46 loop invariant benchmarks
- Compared our results against BLAST, InvGen, and Interproc:

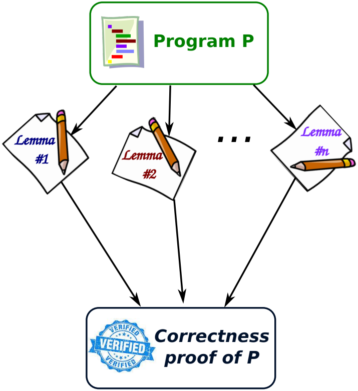


- Can verify 13 benchmarks that no other tool can verify, but cannot prove two benchmarks at least one tool can show
- No termination guarantees

Application #2: Compositional Verification

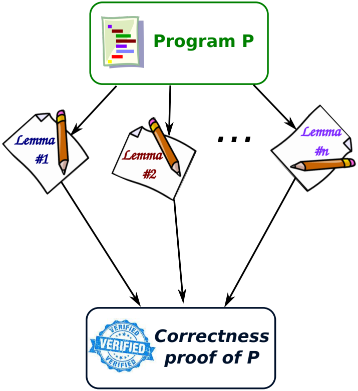


Application #2: Compositional Verification



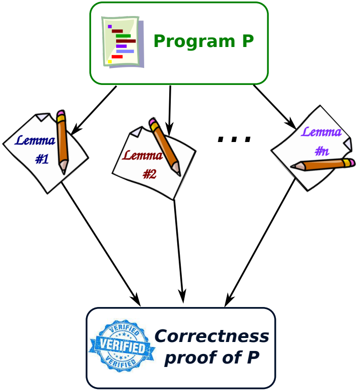
- Compositional approaches decompose proof into lemmas

Application #2: Compositional Verification



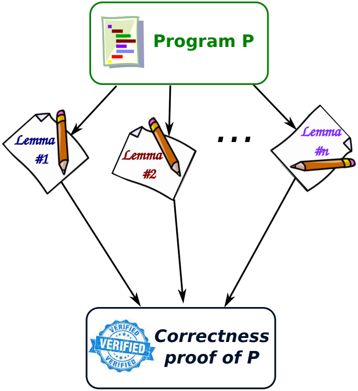
- Compositional approaches decompose proof into lemmas
- Two key advantages:

Application #2: Compositional Verification



- Compositional approaches decompose proof into lemmas
- Two key advantages:
  1. **Scalability:** Each lemma concerns small syntactic part $\Rightarrow$ reason about program fragments in isolation

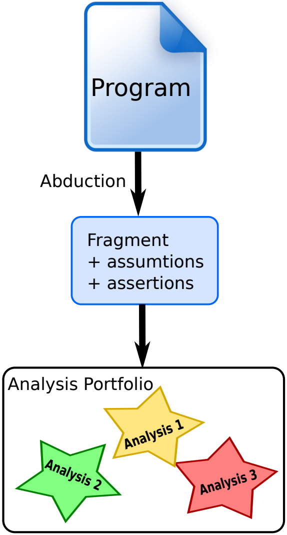
Application #2: Compositional Verification



- Compositional approaches decompose proof into lemmas
- Two key advantages:
  1. **Scalability:** Each lemma concerns small syntactic part $\Rightarrow$ reason about program fragments in isolation
  2. **Abstraction:** Each lemma can be proven using a different abstraction $\Rightarrow$ combine strengths of different techniques

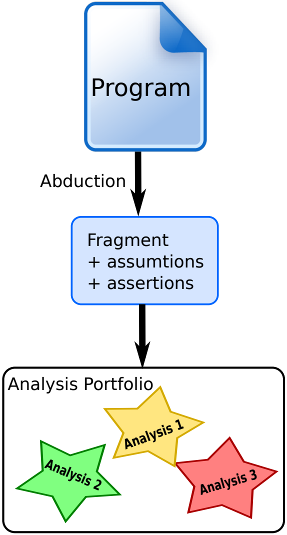
Overview of Compositional Verification Approach

- **Key idea:** Use **abduction** to decompose proof into auxiliary lemmas



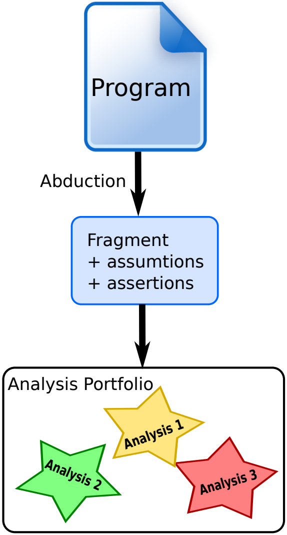
Overview of Compositional Verification Approach

- **Key idea:** Use **abduction** to decompose proof into auxiliary lemmas
  - Lemmas are snippets annotated with assertions and assumptions



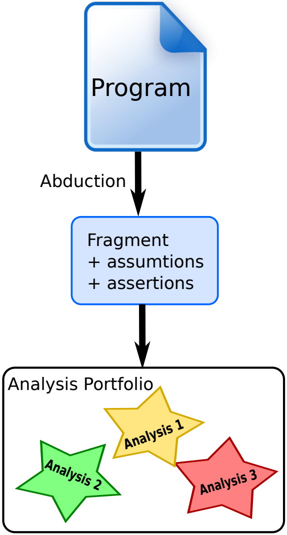
Overview of Compositional Verification Approach

- **Key idea:** Use **abduction** to decompose proof into auxiliary lemmas
  - Lemmas are snippets annotated with assertions and assumptions
- Lemmas are discharged using portfolio of client analyses



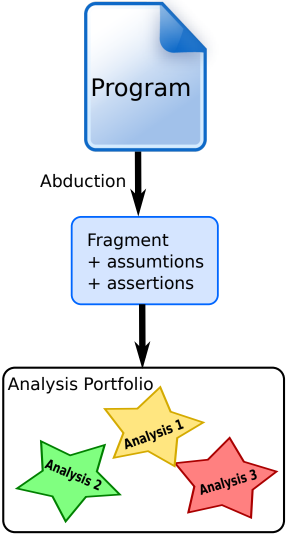
Overview of Compositional Verification Approach

- **Key idea:** Use **abduction** to decompose proof into auxiliary lemmas
  - Lemmas are snippets annotated with assertions and assumptions
- Lemmas are discharged using portfolio of client analyses
- Combine lemmas into overall proof using circular compositional reasoning



Overview of Compositional Verification Approach

- **Key idea:** Use **abduction** to decompose proof into auxiliary lemmas
  - Lemmas are snippets annotated with assertions and assumptions
- Lemmas are discharged using portfolio of client analyses
- Combine lemmas into overall proof using circular compositional reasoning
  - Each lemma can assume correctness of all other lemmas



Example

- Consider this code snippet

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
while(*) {
  assert(x==y);

  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

Example

- Consider this code snippet
- Want to reason about two fragments in isolation

Fragment 1

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
```

```
int x=0, y=0, w=0;
while(*) {
  assert(x==y);

  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

Fragment 2

Example

- Consider this code snippet
- Want to reason about two fragments in isolation
- Focus on fragment containing assertion

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
while(*) {
  assert(x==y);

  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

Example

- Consider this code snippet
- Want to reason about two fragments in isolation
- Focus on fragment containing assertion
- Cannot verify it yet because need precondition "z is odd"

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
while(*) {
  assert(x==y);

  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

Example

- Consider this code snippet
- Want to reason about two fragments in isolation
- Focus on fragment containing assertion
- Cannot verify it yet because need precondition "z is odd"
- Want to automatically infer such missing assumptions!

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
while(*) {
  assert(x==y);

  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

Example cont.

Program Decoration and VC Generation

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(φ1);
while(*) {
  assert(x==y);
  assume(φ2);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

- **Idea:** Decorate program with assume statements containing placeholders (e.g., ϕ_1, ϕ_2)

Example cont.

Program Decoration and VC Generation

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(φ1);
while(*) {
  assert(x==y);
  assume(φ2);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

- **Idea:** Decorate program with assume statements containing placeholders (e.g., ϕ_1, ϕ_2)
- Generate VCs over unknowns ϕ_1 and ϕ_2

Example cont.
Program Decoration and VC Generation

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(φ1);
while(*) {
  assert(x==y);
  assume(φ2);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

- **Idea:** Decorate program with assume statements containing placeholders (e.g., ϕ_1, ϕ_2)
- Generate VCs over unknowns ϕ_1 and ϕ_2
- VC 1:

$$(x = 0 \wedge y = 0 \\ \wedge w = 0 \wedge \phi_1) \Rightarrow x = y$$

Example cont.

Program Decoration and VC Generation

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(φ1);
while(*) {
    assert(x==y);
    assume(φ2);
σ   z+=x+y+w;
    y++;
    x+=z%2;
    w+=2;
}
```

- **Idea:** Decorate program with assume statements containing placeholders (e.g., ϕ_1, ϕ_2)
- Generate VCs over unknowns ϕ_1 and ϕ_2
- VC 1:

$$(x = 0 \land y = 0 \land w = 0 \land \phi_1) \Rightarrow x = y$$

- VC 2:

$$(\phi_2 \land x = y) \Rightarrow wp(\sigma, x = y)$$

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(φ1);
while(*) {
    assert(x==y);
    assume(φ2);
σ   z+=x+y+w;
    y++;
    x+=z%2;
    w+=2;
}
```

- **Idea:** Decorate program with assume statements containing placeholders (e.g., ϕ_1, ϕ_2)
- Generate VCs over unknowns ϕ_1 and ϕ_2
- VC 1: VALID

$$(x = 0 \land y = 0 \\ \land w = 0 \land \phi_1) \Rightarrow x = y$$

- VC 2:

$$(\phi_2 \land x = y) \Rightarrow wp(\sigma, x = y)$$

Example cont.

Program Decoration and VC Generation

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(φ1);
while(*) {
    assert(x==y);
    assume(φ2);
    z+=x+y+w;
    y++;
    x+=z%2;
    w+=2;
}
```

σ

- **Idea:** Decorate program with assume statements containing placeholders (e.g., ϕ_1, ϕ_2)
- Generate VCs over unknowns ϕ_1 and ϕ_2
- VC 1: VALID

$$(x = 0 \land y = 0 \land w = 0 \land \phi_1) \Rightarrow x = y$$

- VC 2: NOT VALID

$$(\phi_2 \land x = y) \Rightarrow wp(\sigma, x = y)$$

- Fix VC2 using abduction:

$$\phi_2 : (w + z)\%2 = 1$$

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(ϕ1);
while(*) {
  assert(x==y);
  assume(ϕ2);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

σ

- Fix VC2 using abduction:

$$\phi_2 : (w+z)\%2 = 1$$

- Now use **circular compositional reasoning**

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(φ1);
while(*) {
  assert(x==y);
  assume(φ2);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

σ

Example, cont.
Lemma Inference using Abduction

- Fix VC2 using abduction:

$$\phi_2 : (w + z)\%2 = 1$$

- Now use **circular compositional reasoning**

- **Subgoal 1:** Prove $x = y$ using $(w + z)\%2 = 1$

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;

while(*) {
   assert(x==y);
   assume((w+z)%2=1);
   z+=x+y+w;
   y++;
   x+=z%2;
   w+=2;
}
```

Example, cont.
Lemma Inference using Abduction

- Fix VC2 using abduction:

$$\phi_2 : (w + z)\%2 = 1$$

- Now use **circular compositional reasoning**

- **Subgoal 1:** Prove $x = y$ using $(w + z)\%2 = 1$

- **Subgoal 2:** Prove ϕ_2 assuming $x = y$

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;

while(*) {
  assume(x==y);
  assert((w+z)%2=1);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

Example, cont.

Lemma Inference using Abduction

- Fix VC2 using abduction:

$$\phi_2 : (w + z)\%2 = 1$$

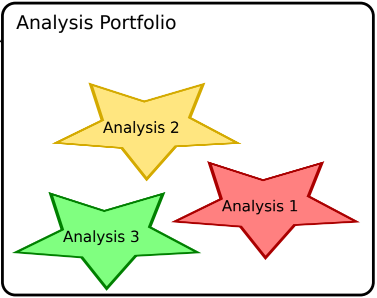
- Now use **circular compositional reasoning**
- **Subgoal 1:** Prove $x = y$ using $(w + z)\%2 = 1$
- **Subgoal 2:** Prove ϕ_2 assuming $x = y$
- Subgoal 1 is immediately discharged; focus on subgoal 2

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;

while(*) {
  assume(x==y);
  assert((w+z)%2=1);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

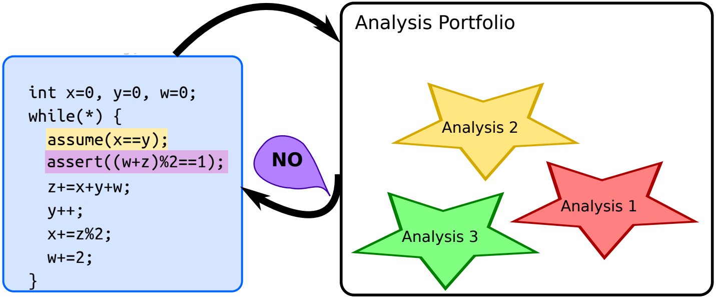
Example, cont.

```
int x=0, y=0, w=0;
while(*) {
  assume(x==y);
  assert((w+z)%2==1);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```



- Invoke client analyses to discharge proof subgoal

Example, cont.



- Invoke client analyses to discharge proof subgoal
- No client can prove it because initial value of z unconstrained

Example, cont.

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(φ1);
while(*) {
   assume(x==y);
   assert((w+z)%2==1);
   z+=x+y+w;
   y++;
   x+=z%2;
   w+=2;
}
```

- Go back to lemma inference and annotate program with unknown precondition

Example, cont.

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(φ1);
while(*) {
   assume(x==y);
   assert((w+z)%2==1);
   z+=x+y+w;
   y++;
   x+=z%2;
   w+=2;
}
```

- Go back to lemma inference and annotate program with unknown precondition
- Generate VC and solve for unknown ϕ_1:

$$\phi_1 : z\%2 = 1$$

Example, cont.

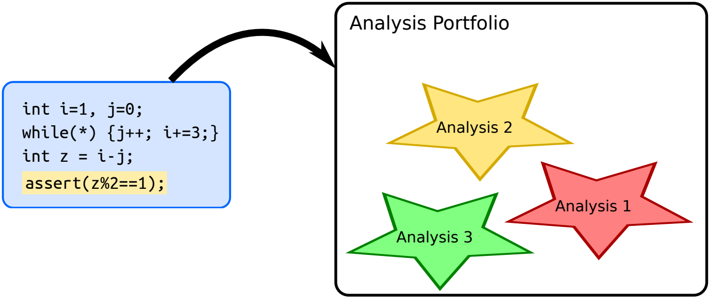
```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(φ1);
while(*) {
  assume(x==y);
  assert((w+z)%2==1);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

- Go back to lemma inference and annotate program with unknown precondition
- Generate VC and solve for unknown ϕ_1:

$$\phi_1 : z\%2 = 1$$

- Now, ϕ_1 becomes a lemma (assertion) to be proven

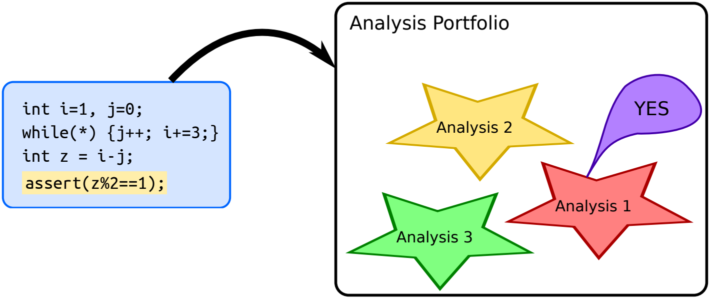
Example, cont.
Invoking Client Analyses



- Now, annotate first fragment with assertion and invoke clients

Example, cont.

Invoking Client Analyses



- Now, annotate first fragment with assertion and invoke clients
- Can be shown by any client analysis that can establish $i = 3j + 1$

Example, cont.

Final Piece

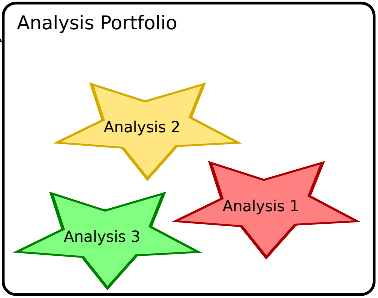
```
int x=0, y=0, w=0;
assume(z%2==1);
while(*) {
  assume(x==y);
  assert((w+z)%2==1);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

- Now, add this as assumption to second fragment

Example, cont.

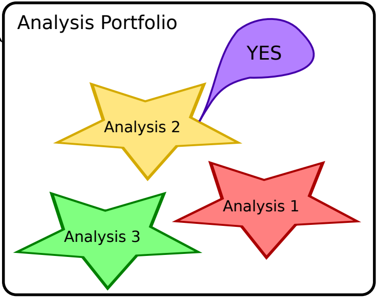
Final Piece

```
int x=0, y=0, w=0;
assume(z%2==1);
while(*) {
  assume(x==y);
  assert((w+z)%2==1);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```



- Now, add this as assumption to second fragment
- Again, invoke client analyses to verify second fragment

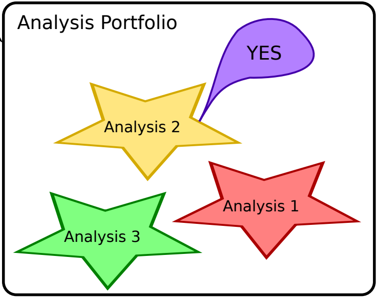
```
int x=0, y=0, w=0;
assume(z%2==1);
while(*) {
  assume(x==y);
  assert((w+z)%2==1);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```



- Now, add this as assumption to second fragment
- Again, invoke client analyses to verify second fragment – can be proven using linear congruences

Example, cont.
Final Piece

```
int x=0, y=0, w=0;
assume(z%2==1);
while(*) {
  assume(x==y);
  assert((w+z)%2==1);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```



- Now, add this as assumption to second fragment
- Again, invoke client analyses to verify second fragment – can be proven using linear congruences

We have now proven the original assertion!

Essence of Technique

- Approach involves two key ingredients: **assertion elimination** and **assertion introduction**

Essence of Technique

- Approach involves two key ingredients: **assertion elimination** and **assertion introduction**
- Assertions introduced using abductive inference

Essence of Technique

- Approach involves two key ingredients: **assertion elimination** and **assertion introduction**
- Assertions introduced using abductive inference
- Assertions eliminated using client analyses and circular compositional reasoning

Essence of Technique

- Approach involves two key ingredients: **assertion elimination** and **assertion introduction**
- Assertions introduced using abductive inference
- Assertions eliminated using client analyses and circular compositional reasoning
- Similar to SMT solver – core part performs VC gen + abduction

Essence of Technique

- Approach involves two key ingredients: **assertion elimination** and **assertion introduction**
- Assertions introduced using abductive inference
- Assertions eliminated using client analyses and circular compositional reasoning
- Similar to SMT solver – core part performs VC gen + abduction

- Client analyses similar to theory solvers

Experiments

- Used this technique to verify safety properties in C programs

Experiments

- Used this technique to verify safety properties in C programs
- Used four different static analysis tools as clients

Experiments

- Used this technique to verify safety properties in C programs
- Used four different static analysis tools as clients

Name	LOC	Time (s)	# queries	Avg # vars in query	Avg LOC in query
Wizardpen Linux Driver	1242	3.8	5	1.5	29
OpenSSH clientloop	1987	2.8	3	2.3	5
Coreutils su	1057	3.0	5	1.7	6
GSL Histogram	526	0.6	4	3.6	15
GSL Matrix	7233	16.9	8	1.8	7

Experiments

- Used this technique to verify safety properties in C programs
- Used four different static analysis tools as clients

Name	LOC	Time (s)	# queries	Avg # vars in query	Avg LOC in query
Wizardpen Linux Driver	1242	3.8	5	1.5	29
OpenSSH clientloop	1987	2.8	3	2.3	5
Coreutils su	1057	3.0	5	1.7	6
GSL Histogram	526	0.6	4	3.6	15
GSL Matrix	7233	16.9	8	1.8	7

Property can be proven using our technique, but not using individual clients

Experiments

- Used this technique to verify safety properties in C programs
- Used four different static analysis tools as clients

Name	LOC	Time (s)	# queries	Avg # vars in query	Avg LOC in query
Wizardpen Linux Driver	1242	3.8	5	1.5	29
OpenSSH clientloop	1987	2.8	3	2.3	5
Coreutils su	1057	3.0	5	1.7	6
GSL Histogram	526	0.6	4	3.6	15
GSL Matrix	7233	16.9	8	1.8	7

Verification time reasonable (0.6-16.9s)

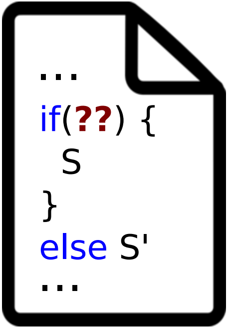
Experiments

- Used this technique to verify safety properties in C programs
- Used four different static analysis tools as clients

Name	LOC	Time (s)	# queries	Avg # vars in query	Avg LOC in query
Wizardpen Linux Driver	1242	3.8	5	1.5	29
OpenSSH clientloop	1987	2.8	3	2.3	5
Coreutils su	1057	3.0	5	1.7	6
GSL Histogram	526	0.6	4	3.6	15
GSL Matrix	7233	16.9	8	1.8	7

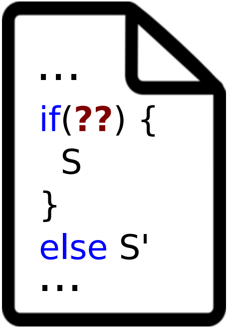
Fragments extracted for queries small in practice

Application #3: Automated Guard Synthesis



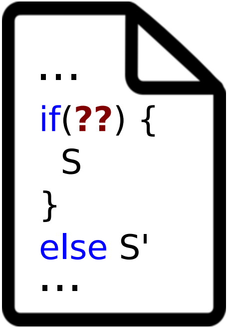
- In **program sketching**, programmer writes a draft program with "holes"

Application #3: Automated Guard Synthesis



- In **program sketching**, programmer writes a draft program with "holes"
- Program synthesizer completes the holes in a way that satisfies specification

Application #3: Automated Guard Synthesis



- In **program sketching**, programmer writes a draft program with "holes"
- Program synthesizer completes the holes in a way that satisfies specification
- Abduction is useful for synthesizing unknown guards in program sketches

Concrete Use Case: Memory Safety

- Programmers often write checks to prevent memory safety errors (buffer overruns, null dereferences, ...)

```
if(C) {R} else { /* handle error */}
```

Concrete Use Case: Memory Safety

- Programmers often write checks to prevent memory safety errors (buffer overruns, null dereferences, ...)

```
if(C) {R} else { /* handle error */}
```

- Such checks are tedious to write and error-prone (e.g, off-by-one errors common cause of buffer overflows)

Guard Synthesis for Memory Safety

Key Idea: Program synthesis to guarantee memory safety

```
if(???)  {R} else { /* handle error */}
```

Guard Synthesis for Memory Safety

Key Idea: Program synthesis to guarantee memory safety

```
if(???)  {R} else { /* handle error */}
```

1. Programmer specifies which parts of the program should be protected and how to handle error

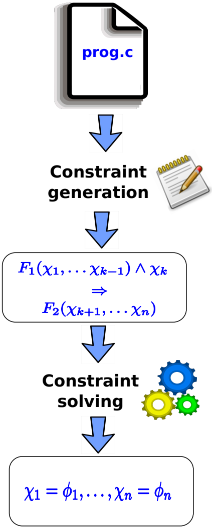
Guard Synthesis for Memory Safety

Key Idea: Program synthesis to guarantee memory safety

```
if(???)  {R} else { /* handle error */}
```

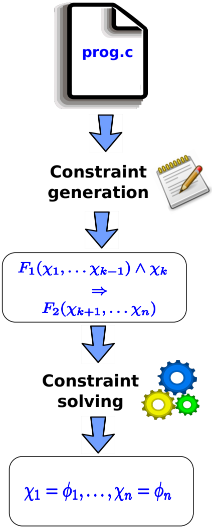
1. Programmer specifies which parts of the program should be protected and how to handle error
2. Technique synthesizes guards that guarantee memory safety
   - Guards should be as permissive and concise as possible
   - Key ingredient of synthesis algorithm is abduction

Solution Overview



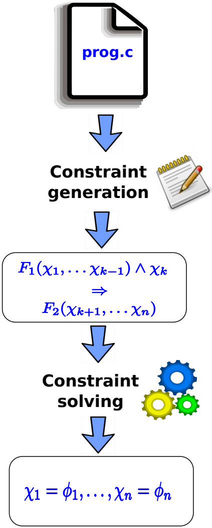
1. Constraint Generation:

Solution Overview



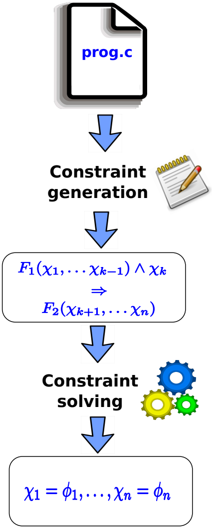
1. Constraint Generation:
   - Represent unknown guards using placeholders

Solution Overview



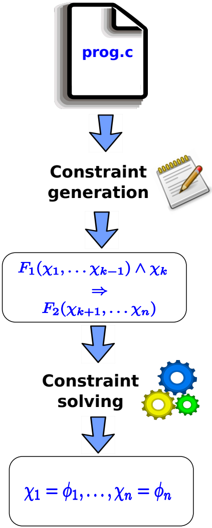
1. Constraint Generation:
   - Represent unknown guards using placeholders
   - Perform dual forward and backward analysis to generate constraint for each unknown

Solution Overview



1. Constraint Generation:
   - Represent unknown guards using placeholders
   - Perform dual forward and backward analysis to generate constraint for each unknown
2. Constraint Solving:

Solution Overview



1. Constraint Generation:
    - Represent unknown guards using placeholders
    - Perform dual forward and backward analysis to generate constraint for each unknown
2. Constraint Solving:
    - An extended abduction algorithm for solving constraint system with multiple unknowns

In More Detail

- Generate one constraint per unknown

ϕ
```
{
    ...
}
```

ψ
```
if(??)
{
    ...
}
```

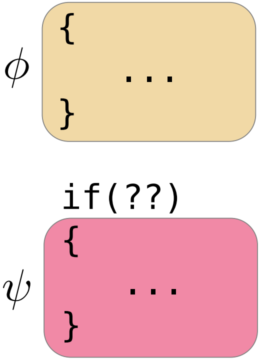
In More Detail

ϕ
```
{
    ...
}
```

```
if(??)
```
ψ
```
{
    ...
}
```

- Generate one constraint per unknown
- Compute postcondition ϕ of code before unknown

In More Detail



- Generate one constraint per unknown
- Compute postcondition ϕ of code before unknown
- Compute safety precondition ψ of code nested inside unknown

In More Detail

```
φ  {
       ...
   }

   if(??)
ψ  {
       ...
   }
```

- Generate one constraint per unknown
- Compute postcondition ϕ of code before unknown
- Compute safety precondition ψ of code nested inside unknown
- To guarantee memory safery, find ?? such that $\phi \wedge\ ?? \models \psi$

In More Detail

ϕ

```
{
    ...
}
```

ψ

```
if(??)
{
    ...
}
```

- Generate one constraint per unknown
- Compute postcondition ϕ of code before unknown
- Compute safety precondition ψ of code nested inside unknown
- To guarantee memory safery, find ?? such that $\phi \wedge ?? \models \psi$
- This is almost an abduction problem, but ϕ, ψ can have other unknowns

In More Detail

ϕ
```
{
    ...
}
```

```
if(??)
{
    ...
}
```
ψ

- Generate one constraint per unknown
- Compute postcondition ϕ of code before unknown
- Compute safety precondition ψ of code nested inside unknown
- To guarantee memory safery, find $??$ such that $\phi \land\ ?? \models \psi$
- This is almost an abduction problem, but ϕ, ψ can have other unknowns
- Impose ordering on constraints and reduce to standard abduction

Example

- Code snippet from Unix Coreutils with protected memory access

```
int main(int argc,
    char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;

  optind=0;
  while(...) {
    optind++;
    if(*) {argv++;
           argc--;}
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```

Example

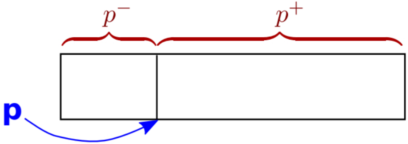
- Code snippet from Unix Coreutils with protected memory access
- Convention: For pointer p:

```
int main(int argc,
   char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;

  optind=0;
  while(...) {
    optind++;
    if(*) {argv++;
           argc--;}
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```

Example

- Code snippet from Unix Coreutils with protected memory access
- Convention: For pointer p:
  - p^+ represents distance to end of memory block

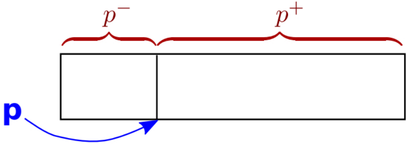



```
int main(int argc,
    char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;

  optind=0;
  while(...) {
    optind++;
    if(*) {argv++;
           argc--;}
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```

Example

- Code snippet from Unix Coreutils with protected memory access
- Convention: For pointer p:
  - p^+ represents distance to end of memory block
  - p^- represents distance from beginning of memory block




```
int main(int argc,
    char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;

  optind=0;
  while(...) {
    optind++;
    if(*) {argv++;
           argc--;}
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```

Example Cont.

- First Step: Compute what is known at ?? $\Rightarrow$ postcondition ϕ

```
int main(int argc,
   char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;

  optind=0;
  while(...) {
    optind++;
    if(*) {argv++;
           argc--;}
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```

Example Cont.

- First Step: Compute what is known at ?? $\Rightarrow$ postcondition ϕ
  - From language semantics:

$$argv^{+} = argc \wedge argv^{-} = 0$$

```
int main(int argc,
   char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;

  optind=0;
  while(...) {
    optind++;
    if(*) {argv++;
           argc--;}
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```

Example Cont.

- First Step: Compute what is known at ?? $\Rightarrow$ postcondition ϕ
  - From language semantics:

    $$argv^{+} = argc \wedge argv^{-} = 0$$

  - From computing the strongest postcondition:

    $$argv^{+} = argc \wedge argv^{-} \geq 1 \;\wedge\; optind \geq 0$$

```
int main(int argc,
   char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;

  optind=0;
  while(...) {
    optind++;
    if(*) {argv++;
           argc--;}
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```

Example Cont.

- Second Step: Compute what needs to hold at ?? to ensure memory safety
  $\Rightarrow$ precondition ψ

```
int main(int argc,
   char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;

  optind=0;
  while(...) {
    optind++;
    if(*) {argv++;
           argc--;}
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```

Example Cont.

- Second Step: Compute what needs to hold at ?? to ensure memory safety
  $\Rightarrow$ precondition ψ

- Buffer access:

$$optind + 1 < argv^{+} \wedge$$
$$optind + 1 \geq -argv^{-}$$

```
int main(int argc,
   char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;

  optind=0;
  while(...) {
    optind++;
    if(*) {argv++;
           argc--;}
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```

Example Cont.

- Solve abduction problem
  $\phi \wedge ?? \models \psi$ where

$$\phi : \begin{array}{c} argv^{+} = argc \wedge \\ argv^{-} \geq 1 \ \wedge \ optind \geq 0 \end{array}$$

$$\psi : \begin{array}{l} optind + 1 < argv^{+} \wedge \\ optind + 1 \geq -argv^{-} \end{array}$$

```
int main(int argc,
   char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;

  optind=0;
  while(...) {
    optind++;
    if(*) {argv++;
           argc--;}
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```

Example Cont.

- Solve abduction problem
  $\phi \wedge ?? \models \psi$ where

$$\phi : \begin{array}{c} argv^{+} = argc \wedge \\ argv^{-} \geq 1 \ \wedge \ optind \geq 0 \end{array}$$

$$\psi : \begin{array}{l} optind + 1 < argv^{+} \wedge \\ optind + 1 \geq -argv^{-} \end{array}$$

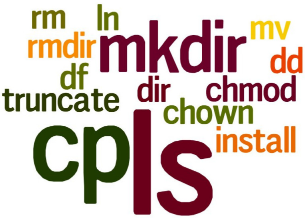
- Solution: $argc - optind > 1$

```
int main(int argc,
   char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;

  optind=0;
  while(...) {
    optind++;
    if(*) {argv++;
           argc--;}
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```

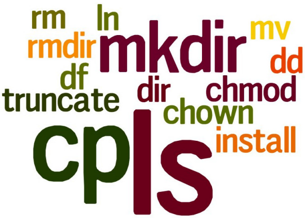
Experiments

- Evaluated technique on the Unix Coreutils and parts of OpenSSH



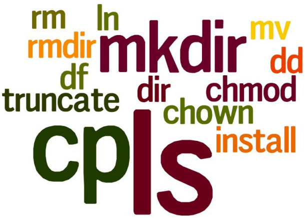
Experiments

- Evaluated technique on the Unix Coreutils and parts of OpenSSH
- Removed conditionals used to prevent memory safety errors



Experiments

- Evaluated technique on the Unix Coreutils and parts of OpenSSH
- Removed conditionals used to prevent memory safety errors
- Used our new technique to synthesize the missing guards



Experiments Cont.

Program	Lines	# holes	Time (s)	Memory	Synthesis successful?	Bug?
Coreutils hostname	160	1	0.15	10 MB	Yes	No
Coreutils tee	223	1	0.84	10 MB	Yes	Yes
Coreutils runcon	265	2	0.81	12 MB	Yes	No
Coreutils chroot	279	2	0.53	23 MB	Yes	No
Coreutils remove	710	2	1.38	66MB	Yes	No
Coreutils nl	758	3	2.07	80 MB	Yes	No
SSH - sshconnect	810	3	1.43	81 MB	Yes	No
Coreutils mv	929	4	2.03	42 MB	Yes	No
SSH - do_authentication	1,904	4	3.92	86 MB	Yes	Yes
SSH - ssh_session	2,260	5	4.35	81 MB	Yes	No

Used technique to synthesize 27 unknown guards in real C programs

Experiments Cont.

Program	Lines	# holes	Time (s)	Memory	Synthesis successful?	Bug?
Coreutils hostname	160	1	0.15	10 MB	Yes	No
Coreutils tee	223	1	0.84	10 MB	Yes	Yes
Coreutils runcon	265	2	0.81	12 MB	Yes	No
Coreutils chroot	279	2	0.53	23 MB	Yes	No
Coreutils remove	710	2	1.38	66MB	Yes	No
Coreutils nl	758	3	2.07	80 MB	Yes	No
SSH - sshconnect	810	3	1.43	81 MB	Yes	No
Coreutils mv	929	4	2.03	42 MB	Yes	No
SSH - do_authentication	1,904	4	3.92	86 MB	Yes	Yes
SSH - ssh_session	2,260	5	4.35	81 MB	Yes	No

In 21 out of 27 cases, tool inferred same predicate as programmer

Experiments Cont.

Program	Lines	# holes	Time (s)	Memory	Synthesis successful?	Bug?
Coreutils hostname	160	1	0.15	10 MB	Yes	No
Coreutils tee	223	1	0.84	10 MB	Yes	Yes
Coreutils runcon	265	2	0.81	12 MB	Yes	No
Coreutils chroot	279	2	0.53	23 MB	Yes	No
Coreutils remove	710	2	1.38	66MB	Yes	No
Coreutils nl	758	3	2.07	80 MB	Yes	No
SSH - sshconnect	810	3	1.43	81 MB	Yes	No
Coreutils mv	929	4	2.03	42 MB	Yes	No
SSH - do_authentication	1,904	4	3.92	86 MB	Yes	Yes
SSH - ssh_session	2,260	5	4.35	81 MB	Yes	No

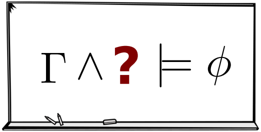
In 4 cases, syntactically different, but semantically equivalent guards

Experiments Cont.

Program	Lines	# holes	Time (s)	Memory	Synthesis successful?	Bug?
Coreutils hostname	160	1	0.15	10 MB	Yes	No
Coreutils tee	223	1	0.84	10 MB	Yes	Yes
Coreutils runcon	265	2	0.81	12 MB	Yes	No
Coreutils chroot	279	2	0.53	23 MB	Yes	No
Coreutils remove	710	2	1.38	66MB	Yes	No
Coreutils nl	758	3	2.07	80 MB	Yes	No
SSH - sshconnect	810	3	1.43	81 MB	Yes	No
Coreutils mv	929	4	2.03	42 MB	Yes	No
SSH - do_authentication	1,904	4	3.92	86 MB	Yes	Yes
SSH - ssh_session	2,260	5	4.35	81 MB	Yes	No

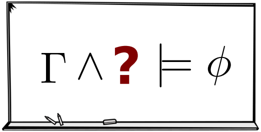
In 2 cases, guards did not match
$\Rightarrow$ bug in original program!

Summary



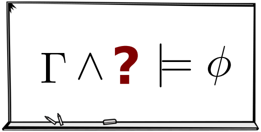
- Abduction $=$ logical formulation of “guessing”

Summary



- Abduction $=$ logical formulation of “guessing”
- Lots of uses in automated reasoning about programs, particularly when combined with backtracking search

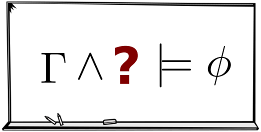
Summary



- Abduction $=$ logical formulation of "guessing"
- Lots of uses in automated reasoning about programs, particularly when combined with backtracking search
- If you are interested in using abduction, check out:

  http://www.cs.utexas.edu/~tdillig/mistral/explain.html

Summary



- Abduction = logical formulation of "guessing"
- Lots of uses in automated reasoning about programs, particularly when combined with backtracking search
- If you are interested in using abduction, check out:

  http://www.cs.utexas.edu/∼tdillig/mistral/explain.html
- Easy to use: `expl = conclusion.abduce(premises);`

Limitations and Future Work

- ⚠ Abduction algorithm uses quantifier elimination $\Rightarrow$ limited scalability and requires to theories that admit QE

Limitations and Future Work

⚠ Abduction algorithm uses quantifier elimination $\Rightarrow$ limited scalability and requires to theories that admit QE

- **Future work:** Alternative algorithms that don't use QE

Limitations and Future Work

- ⚠ Abduction algorithm uses quantifier elimination $\Rightarrow$ limited scalability and requires to theories that admit QE
  - **Future work:** Alternative algorithms that don't use QE
- ⚠ Abduction requires single unknown in LHS, but sometimes there are multiple unknowns

Limitations and Future Work

⚠ Abduction algorithm uses quantifier elimination $\Rightarrow$ limited scalability and requires to theories that admit QE

- **Future work:** Alternative algorithms that don't use QE

⚠ Abduction requires single unknown in LHS, but sometimes there are multiple unknowns

- **On-going work:** Multi-abduction algorithm to simultaneously infer multiple unknowns

Questions?