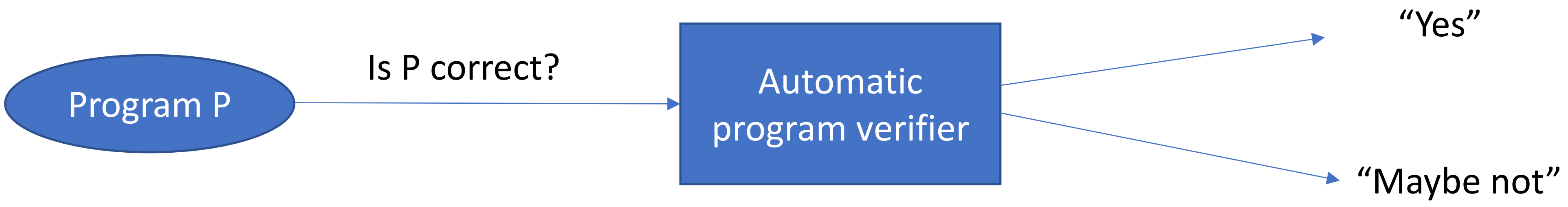
Formal Validation of a Practical Verification Condition Generator

Gaurav Parthasarathy, Peter Müller, Alexander J. Summers

ETH zürich

UBC

Motivation



Program P

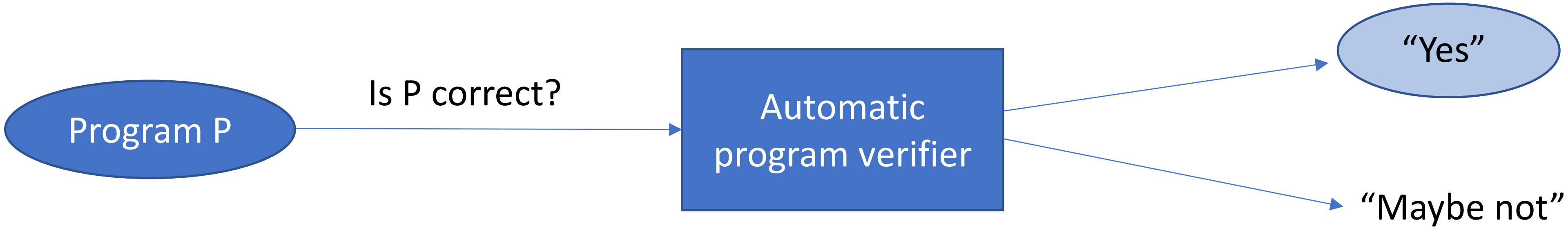
Is P correct?

Automatic program verifier

“Yes”

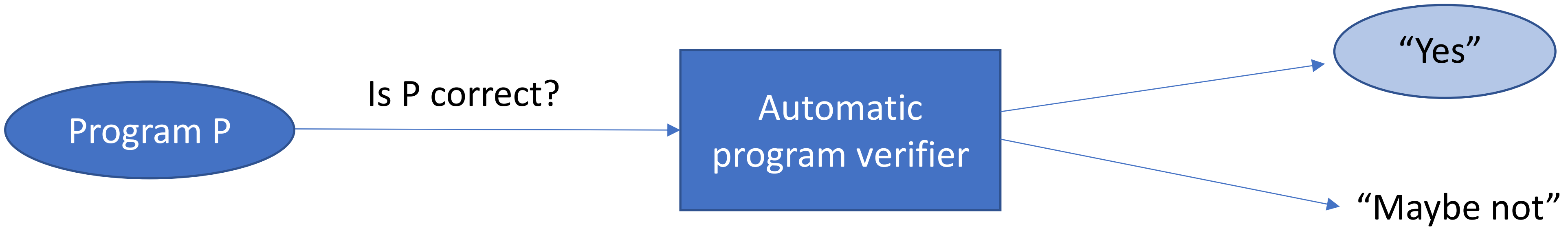
“Maybe not”

Motivation



Soundness of a verifier: If the verifier says “Yes”, then P is indeed correct.

Motivation



Program P

Is P correct?

Automatic program verifier

"Yes"

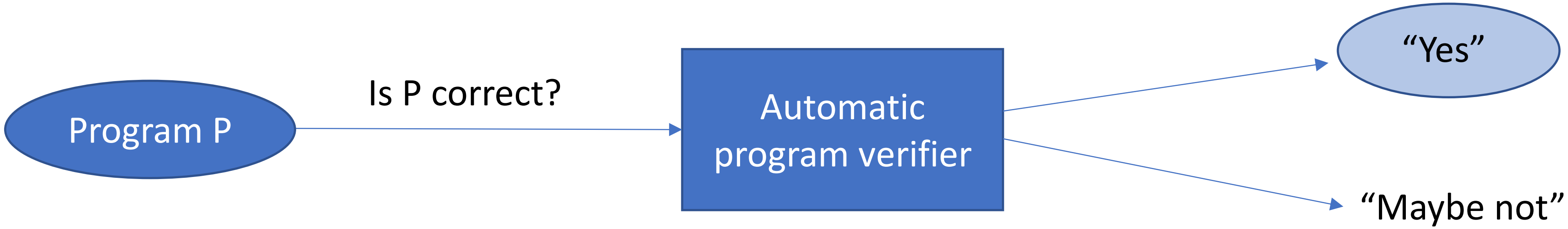
"Maybe not"

Soundness of a verifier: If the verifier says "Yes", then P is indeed correct.

Underlying logic of the verifier

- Sometimes formalised

Motivation



Soundness of a verifier: If the verifier says "Yes", then P is indeed correct.

Underlying logic of the verifier

- Sometimes formalised

Implementation of the verifier

- No formal guarantees
- Consists of many thousands of lines of code

Guarantees for verifier implementations

One possible approach: Prove verifier correct once and for all

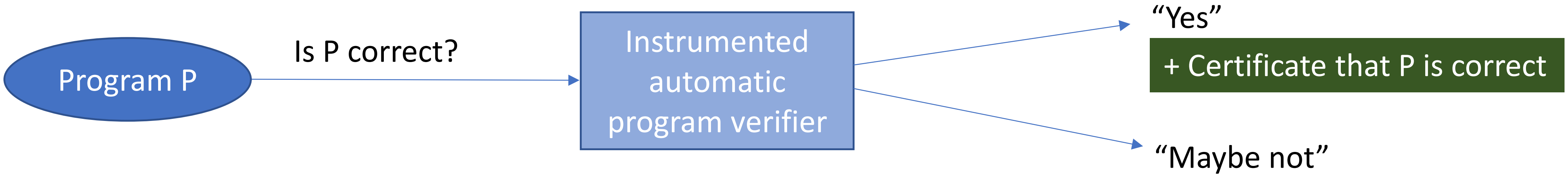
- Impractical for existing verifiers, since implementation languages lack formalisation
- If reimplement in Coq or Isabelle, then lose benefits of modern languages

Guarantees for verifier implementations

One possible approach: Prove verifier correct once and for all

- Impractical for existing verifiers, since implementation languages lack formalisation
- If reimplement in Coq or Isabelle, then lose benefits of modern languages

Our approach: Use a per-run validation strategy

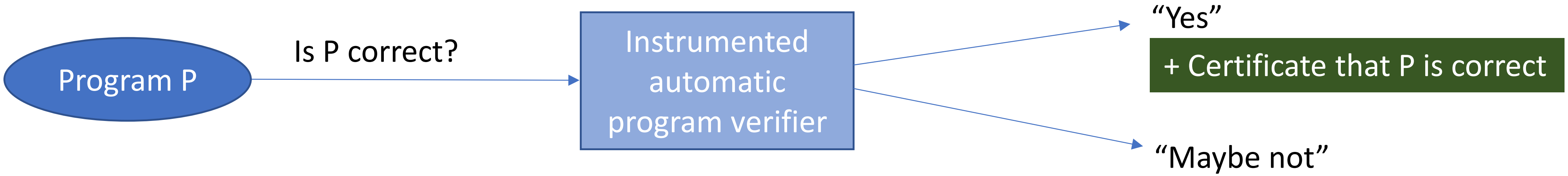

Program P
Is P correct?
Instrumented automatic program verifier
“Yes”
+ Certificate that P is correct
“Maybe not”

Guarantees for verifier implementations

One possible approach: Prove verifier correct once and for all

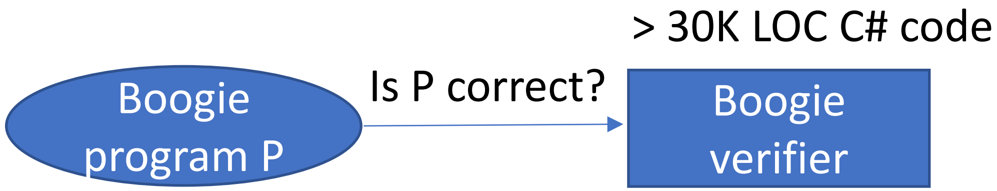
- Impractical for existing verifiers, since implementation languages lack formalisation
- If reimplement in Coq or Isabelle, then lose benefits of modern languages

Our approach: Use a per-run validation strategy

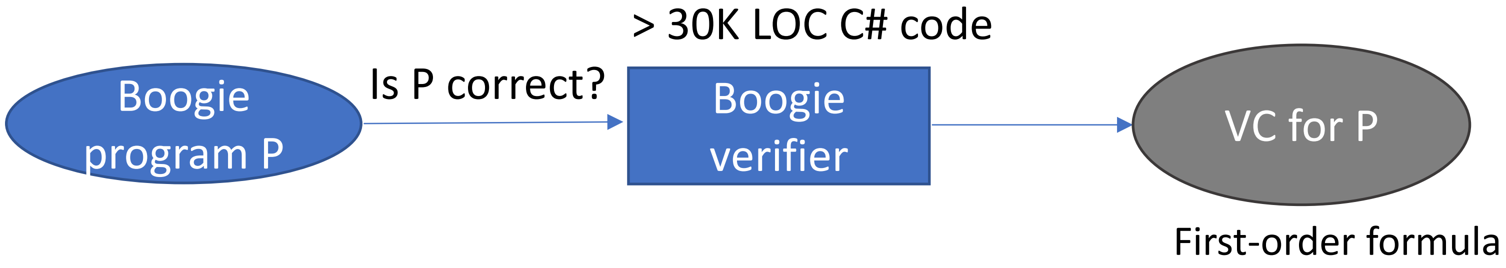

Program P
Is P correct?
Instrumented automatic program verifier
"Yes"
+ Certificate that P is correct
"Maybe not"

- We show it is feasible for an existing verifier
- Certificate generation is robust to implementation changes

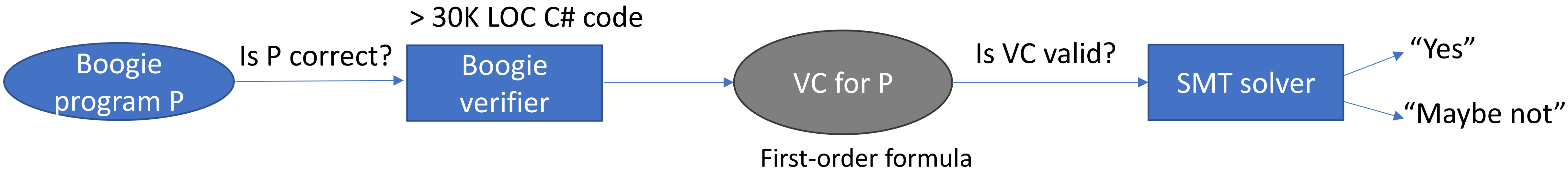
Main contributions



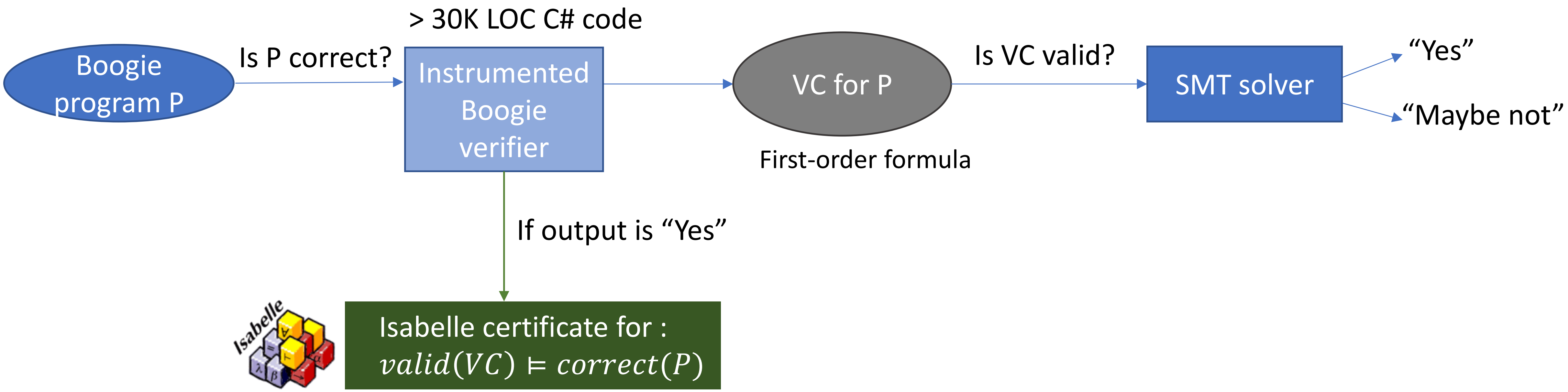
Main contributions



Main contributions



Main contributions

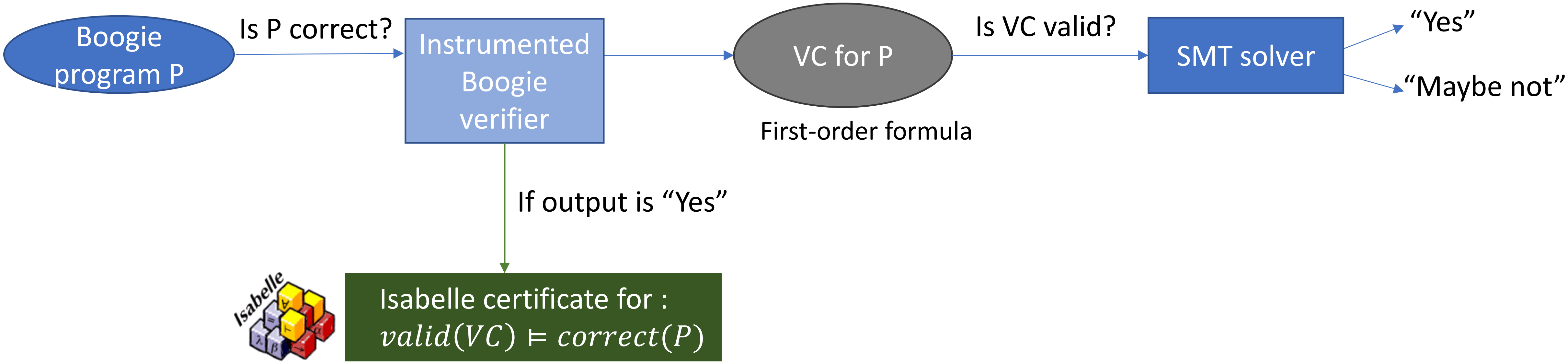


We formalise a subset of the Boogie language

Main contributions

low instrumentation effort (< 250 lines)

> 30K LOC C# code



Boogie program P

Is P correct?

Instrumented Boogie verifier

VC for P

First-order formula

Is VC valid?

SMT solver

“Yes”

“Maybe not”

If output is “Yes”

Isabelle certificate for : $valid(VC) \vDash correct(P)$

We formalise a subset of the Boogie language

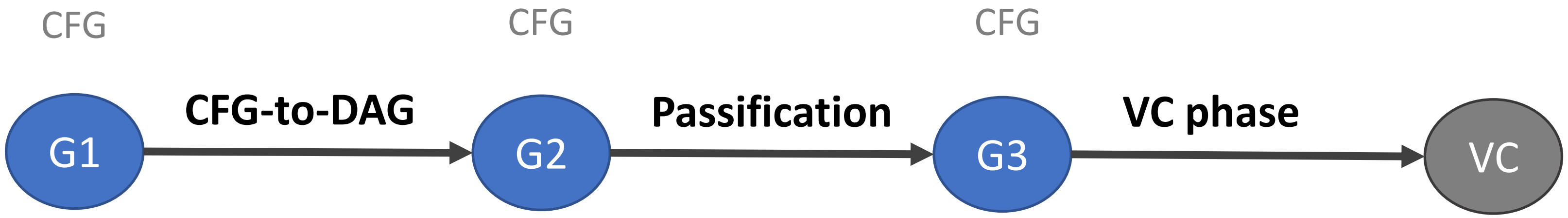
Validating the Boogie verifier

Validating the Boogie verifier

Verification of each procedure is decomposed into phases

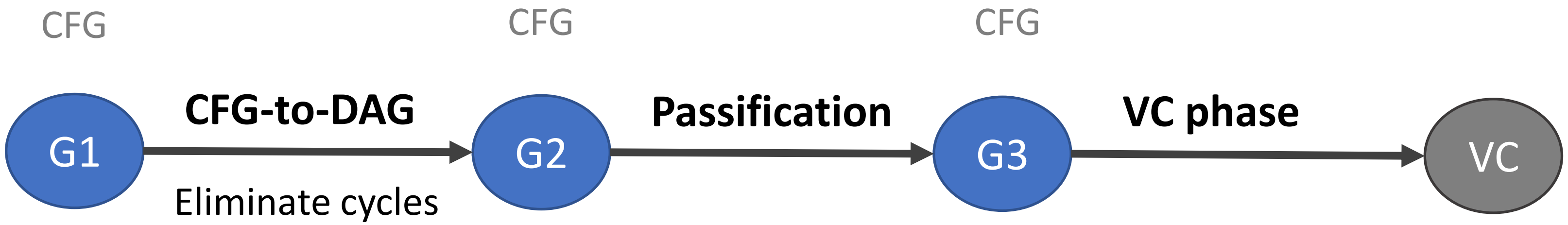
Validating the Boogie verifier

Verification of each procedure is decomposed into phases



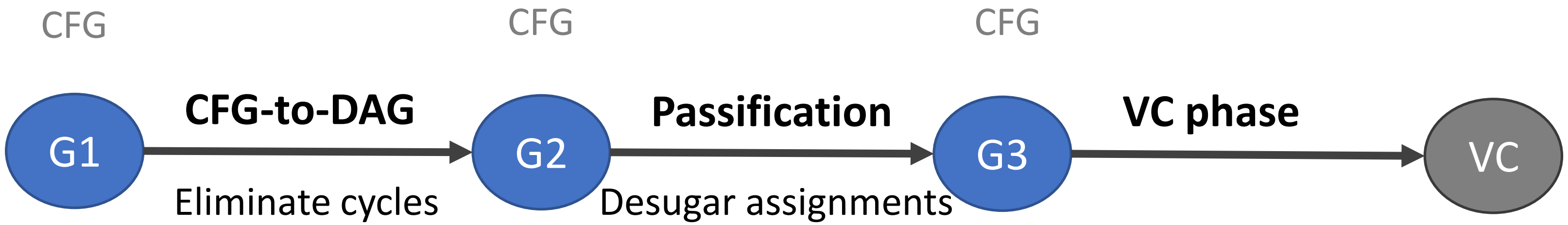
Validating the Boogie verifier

Verification of each procedure is decomposed into phases



Validating the Boogie verifier

Verification of each procedure is decomposed into phases



CFG

G1

CFG-to-DAG

Eliminate cycles

CFG

G2

Passification

Desugar assignments

CFG

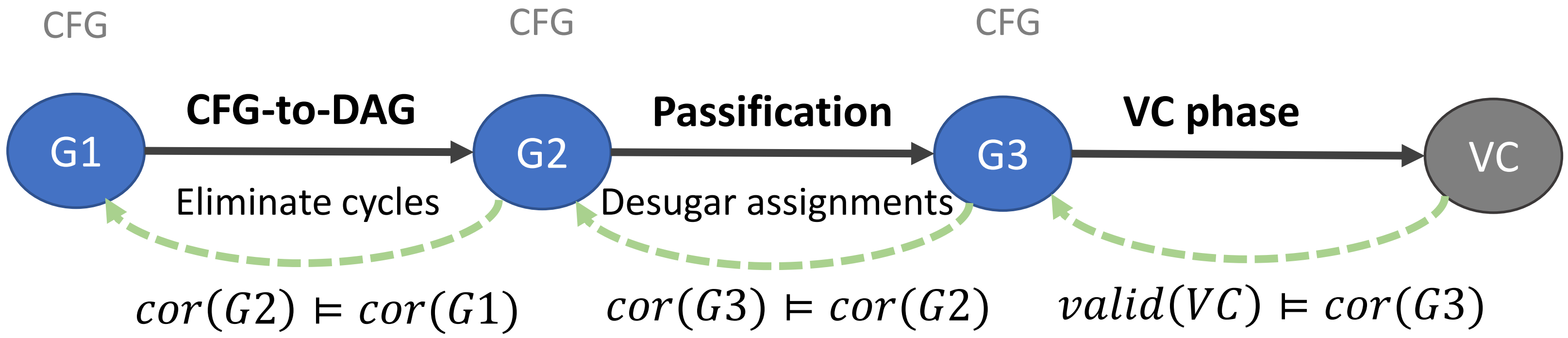
G3

VC phase

VC

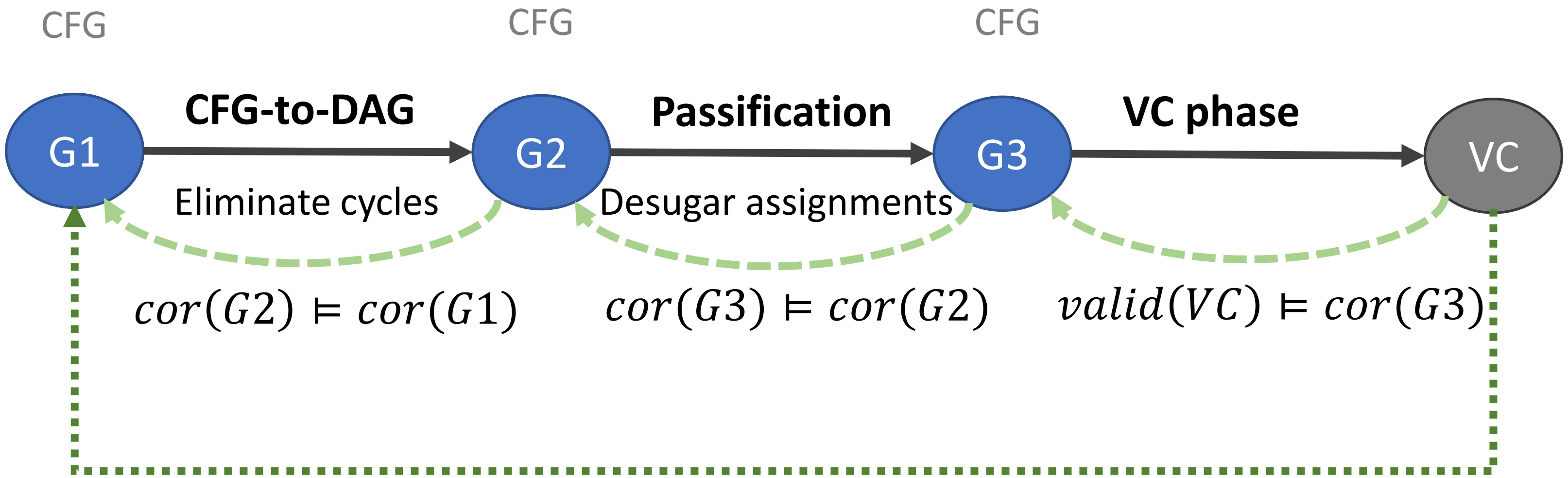
Validating the Boogie verifier

Verification of each procedure is decomposed into phases



Validating the Boogie verifier

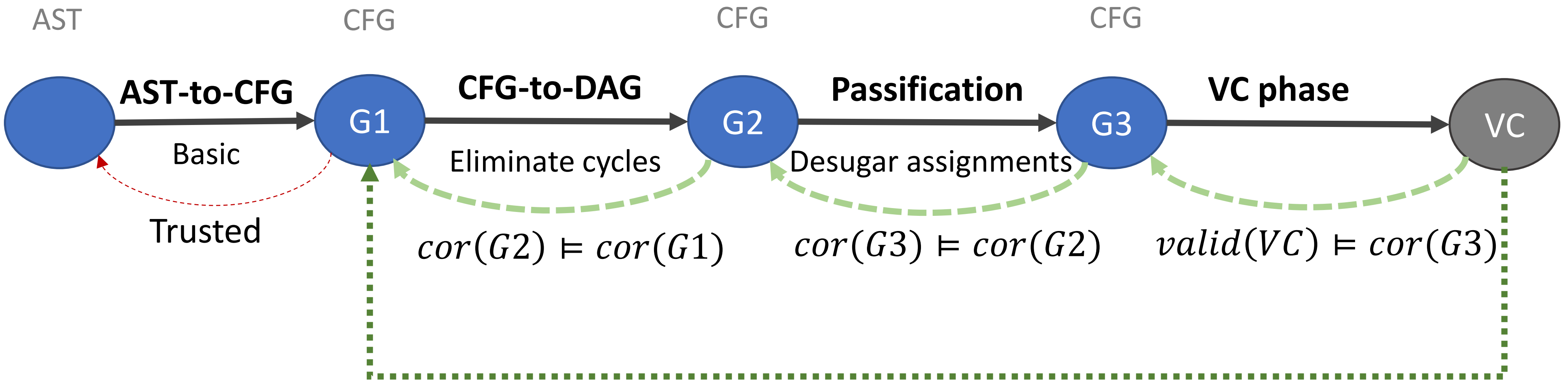
Verification of each procedure is decomposed into phases



Final procedure certificate:
$valid(VC) \vDash cor(G1)$

Validating the Boogie verifier

Verification of each procedure is decomposed into phases



Final procedure certificate:
$valid(VC) \vDash cor(G1)$

Supported Boogie subset

Top-level	Procedures, (poly.) functions, axioms, type constructors, …

Supported Boogie subset

Top-level	Procedures, (poly.) functions, axioms, type constructors, …
Types	Booleans, integers, uninterpreted types

Supported Boogie subset

Top-level	Procedures, (poly.) functions, axioms, type constructors, …
Types	Booleans, integers, uninterpreted types
Expressions	function calls, value and type quantification, …

Supported Boogie subset

Top-level	Procedures, (poly.) functions, axioms, type constructors, …
Types	Booleans, integers, uninterpreted types
Expressions	function calls, value and type quantification, …
Statements	x := E **assert** E **assume** E **havoc** x

Example

```
assume i != 0
j := 0

while (i != 0)
 invariant j >= 0 && (i == 0 ==> j > 0)
{
  j := j+1
  i := i-1
}

assert j > 0
…
```

Example

Consider all possible values for i and j

```
assume i != 0
j := 0

while (i != 0)
 invariant j >= 0 && (i == 0 ==> j > 0)
{
  j := j+1
  i := i-1
}

assert j > 0
…
```

Example

```
assume i != 0
j := 0

while (i != 0)
 invariant j >= 0 && (i == 0 ==> j > 0)
{
  j := j+1
  i := i-1
}

assert j > 0
…
```

Prune executions

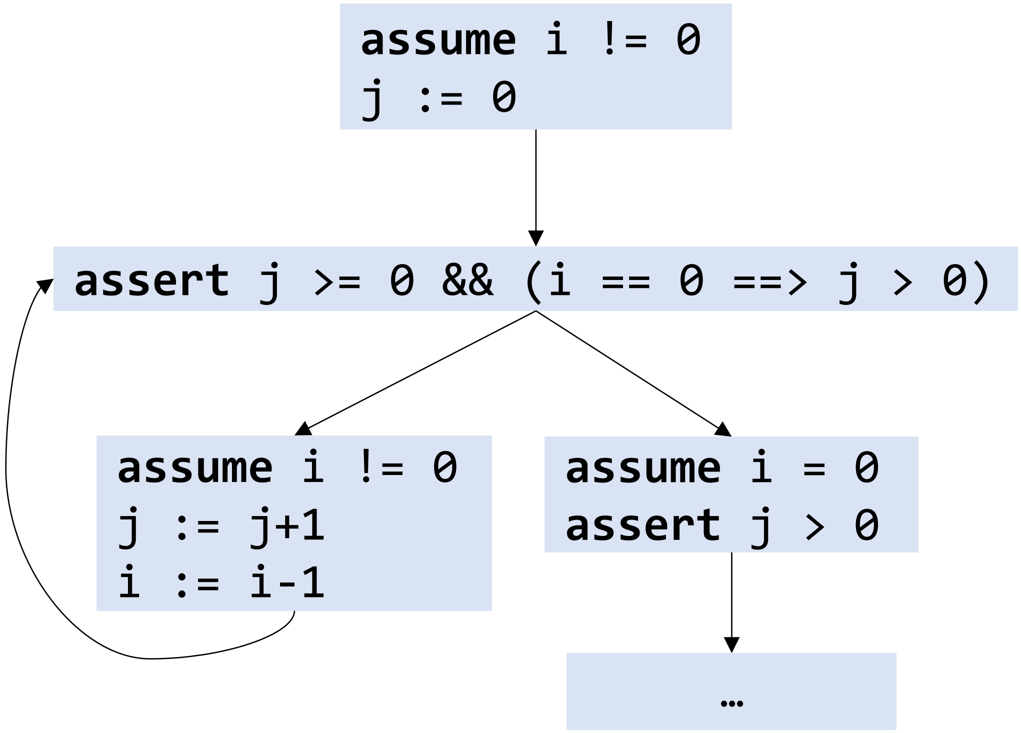
Example

```
assume i != 0
j := 0

while (i != 0)
 invariant j >= 0 && (i == 0 ==> j > 0)
{
  j := j+1
  i := i-1
}

assert j > 0
…
```

Always satisfied here

CFG representation

```
assume i != 0
j := 0

while (i != 0)
 invariant j >= 0 && (i == 0 ==> j > 0)
{
  j := j+1
  i := i-1
}

assert j > 0
…
```



```
assume i != 0
j := 0
```

```
assert j >= 0 && (i == 0 ==> j > 0)
```

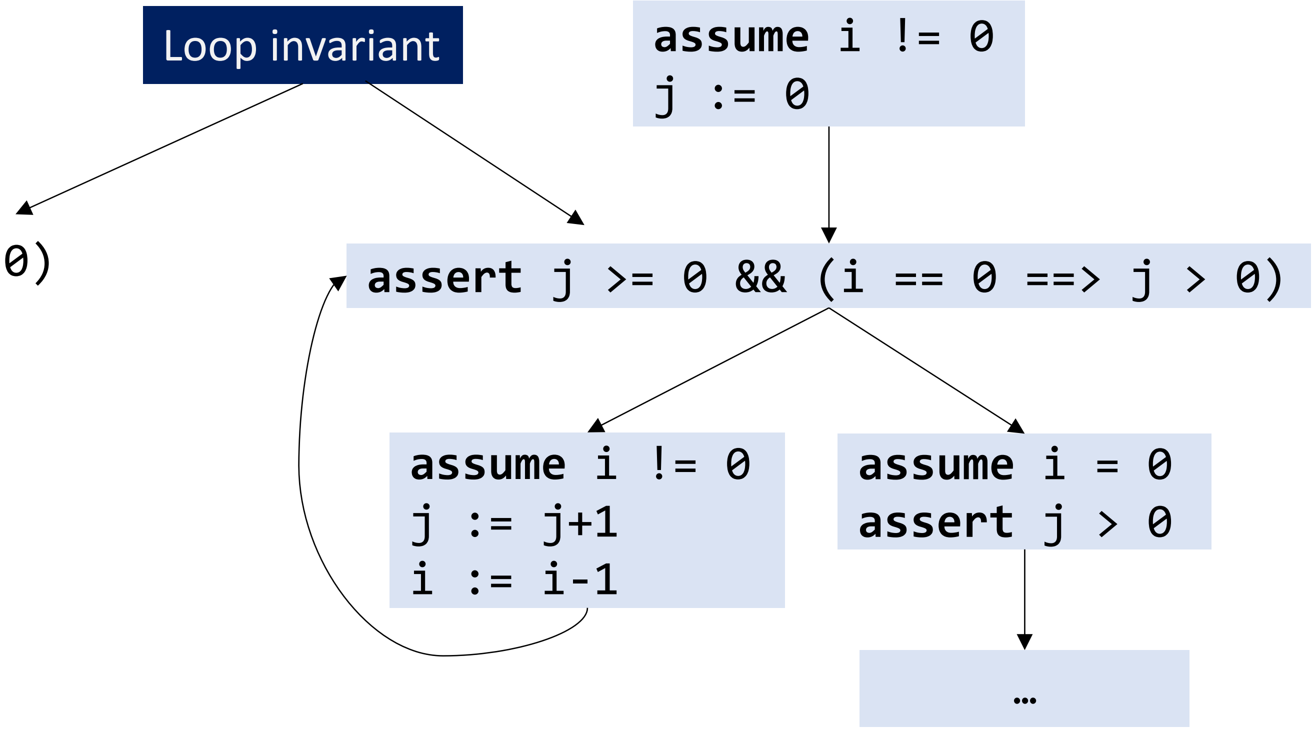
```
assume i != 0
j := j+1
i := i-1
```

```
assume i = 0
assert j > 0
```

```
…
```

CFG representation

```
assume i != 0
j := 0

while (i != 0)
 invariant j >= 0 && (i == 0 ==> j > 0)
{
  j := j+1
  i := i-1
}

assert j > 0
…
```



Loop invariant

```
assume i != 0
j := 0
```

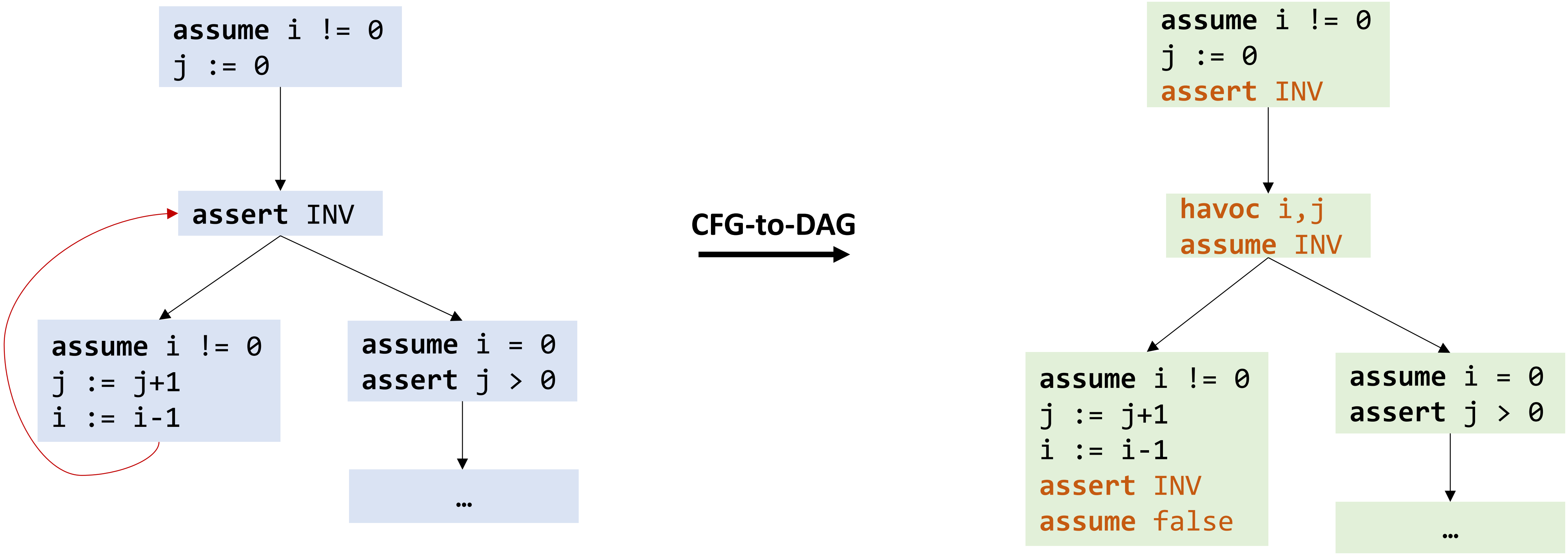
```
assert j >= 0 && (i == 0 ==> j > 0)
```

```
assume i != 0
j := j+1
i := i-1
```

```
assume i = 0
assert j > 0
```

…

CFG-to-DAG phase



```
assume i != 0
j := 0
```

```
assert INV
```

```
assume i != 0
j := j+1
i := i-1
```

```
assume i = 0
assert j > 0
```

```
…
```

CFG-to-DAG

```
assume i != 0
j := 0
assert INV
```

```
havoc i,j
assume INV
```

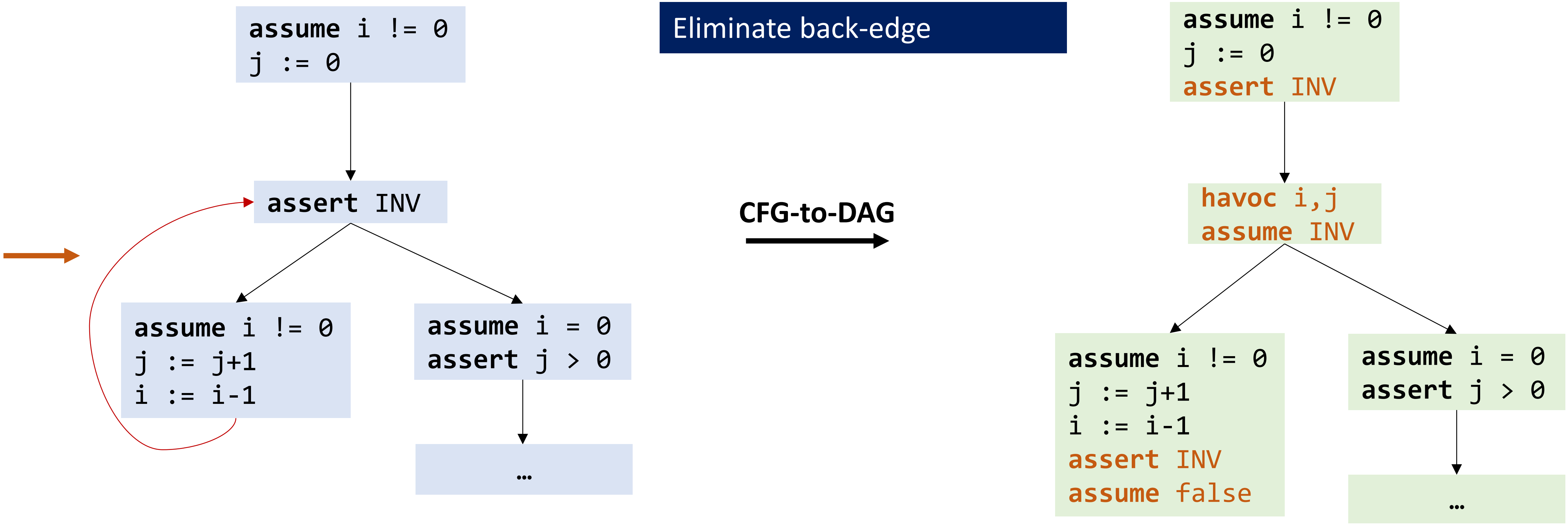
```
assume i != 0
j := j+1
i := i-1
assert INV
assume false
```

```
assume i = 0
assert j > 0
```

```
…
```

CFG-to-DAG phase

Eliminate back-edge



```
assume i != 0
j := 0
```

```
assert INV
```

```
assume i != 0
j := j+1
i := i-1
```

```
assume i = 0
assert j > 0
```

…

CFG-to-DAG

```
assume i != 0
j := 0
assert INV
```

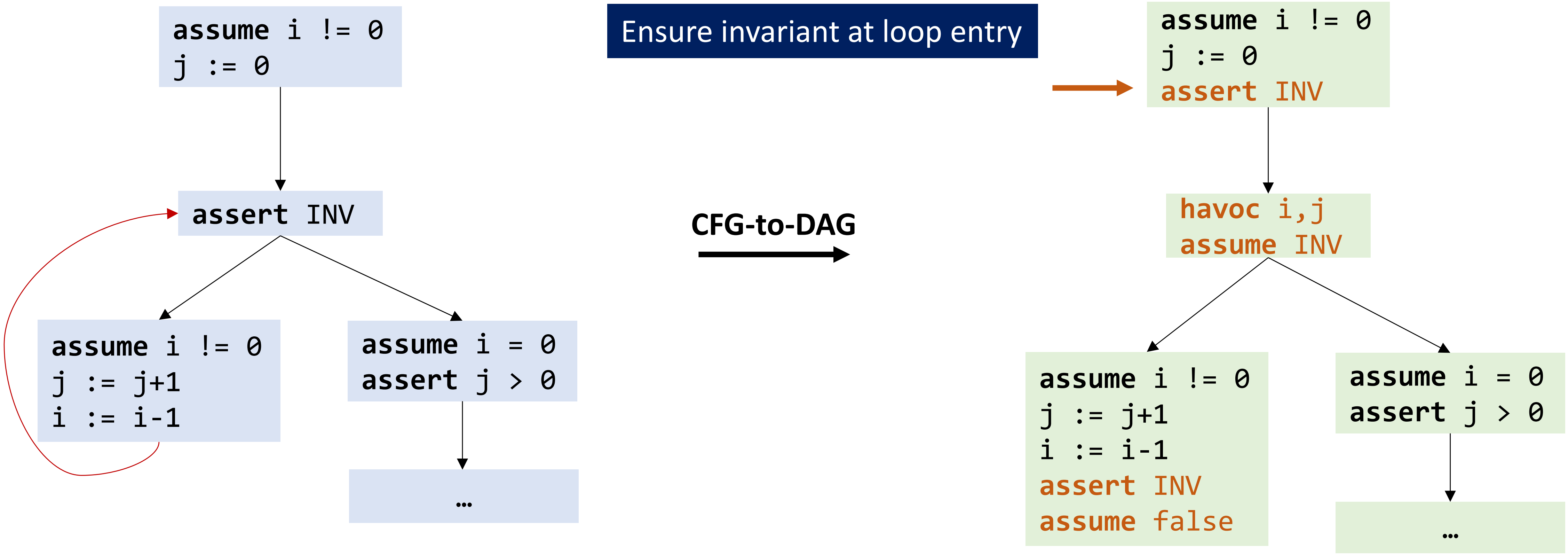
```
havoc i,j
assume INV
```

```
assume i != 0
j := j+1
i := i-1
assert INV
assume false
```

```
assume i = 0
assert j > 0
```

…

CFG-to-DAG phase



Ensure invariant at loop entry

```
assume i != 0
j := 0
```

```
assert INV
```

```
assume i != 0
j := j+1
i := i-1
```

```
assume i = 0
assert j > 0
```

```
…
```

```
assume i != 0
j := 0
assert INV
```

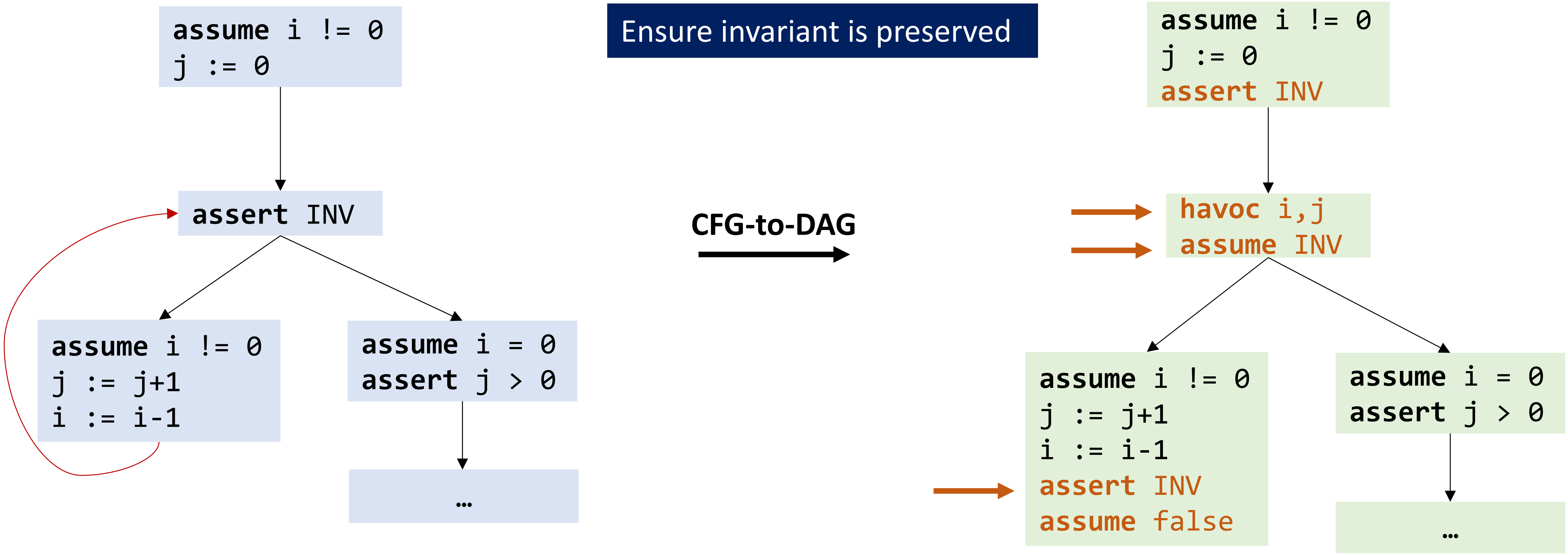
```
havoc i,j
assume INV
```

```
assume i != 0
j := j+1
i := i-1
assert INV
assume false
```

```
assume i = 0
assert j > 0
```

```
…
```

CFG-to-DAG phase



Ensure invariant is preserved

```
assume i != 0
j := 0
```

```
assert INV
```

```
assume i != 0
j := j+1
i := i-1
```

```
assume i = 0
assert j > 0
```

```
…
```

```
assume i != 0
j := 0
assert INV
```

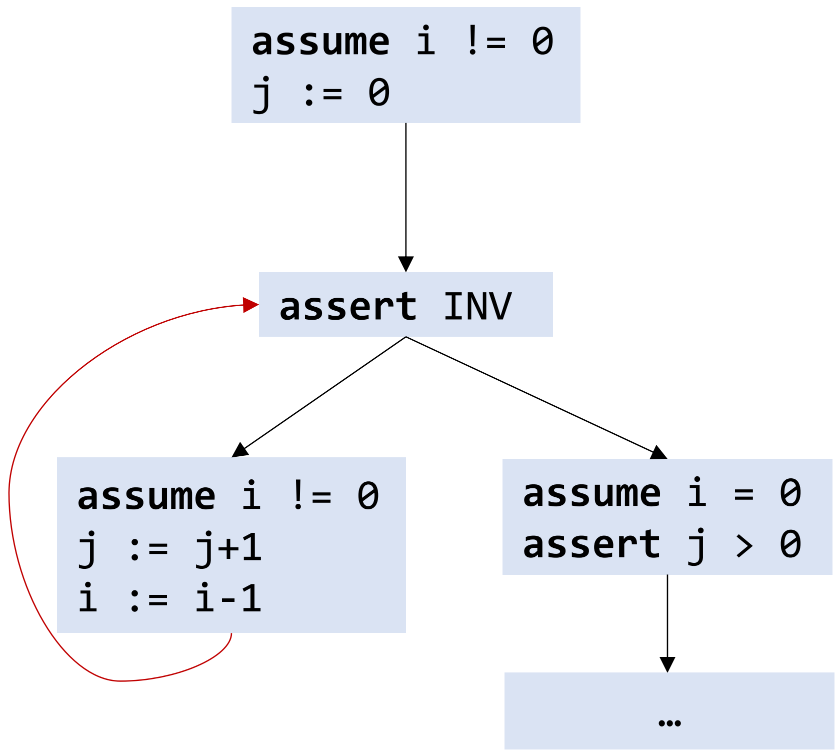
```
havoc i,j
assume INV
```

```
assume i != 0
j := j+1
i := i-1
assert INV
assume false
```

```
assume i = 0
assert j > 0
```

```
…
```

CFG-to-DAG phase



```
assume i != 0
j := 0
```

```
assert INV
```

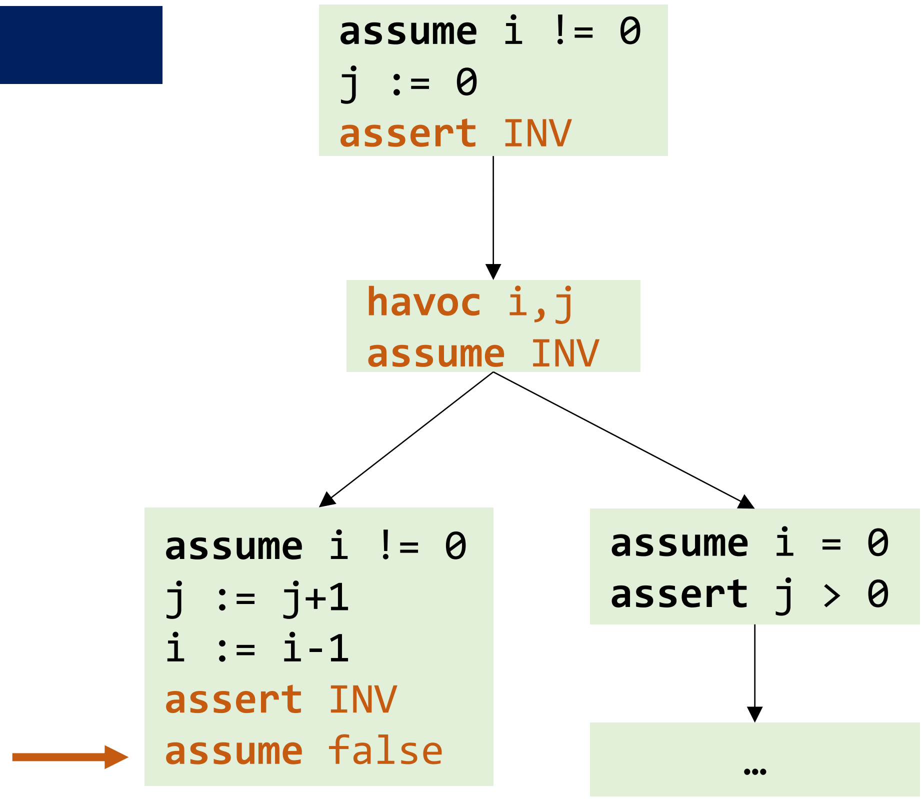
```
assume i != 0
j := j+1
i := i-1
```

```
assume i = 0
assert j > 0
```

```
…
```

Prune loop executions

CFG-to-DAG



```
assume i != 0
j := 0
assert INV
```

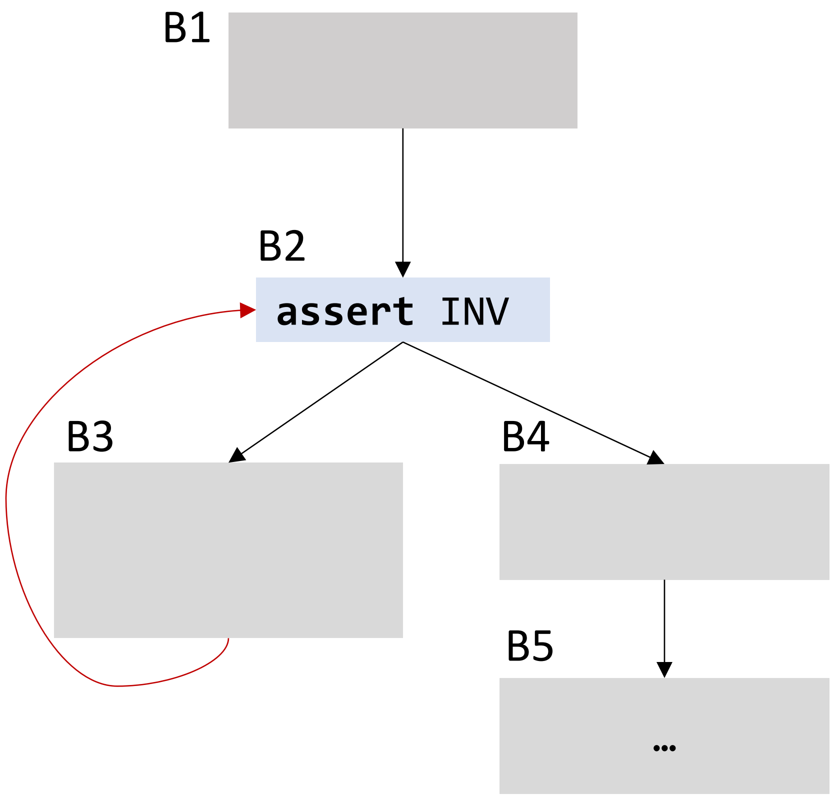
```
havoc i,j
assume INV
```

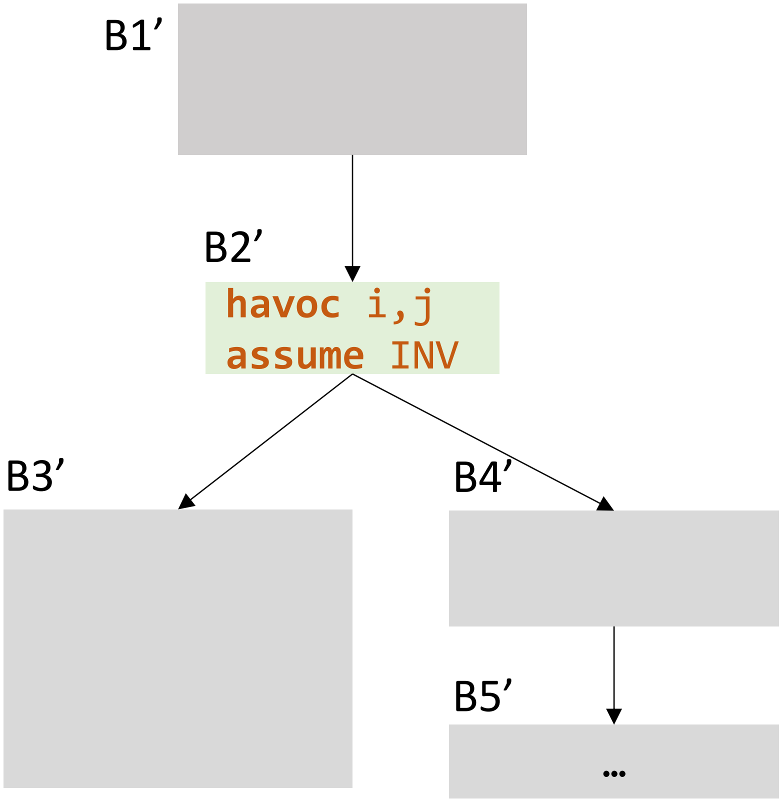
```
assume i != 0
j := j+1
i := i-1
assert INV
assume false
```

```
assume i = 0
assert j > 0
```

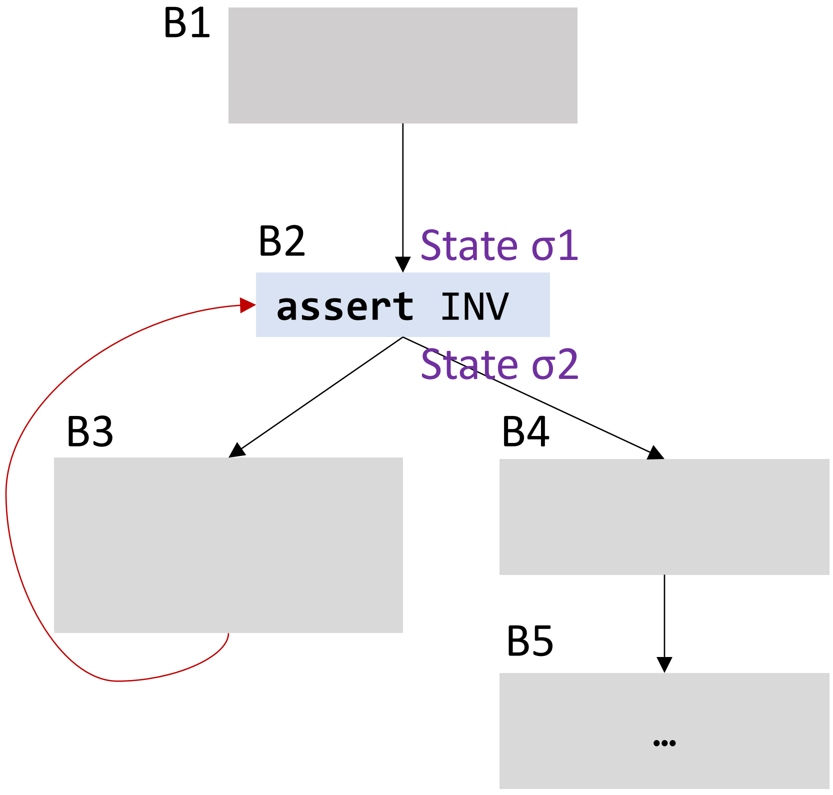
```
…
```

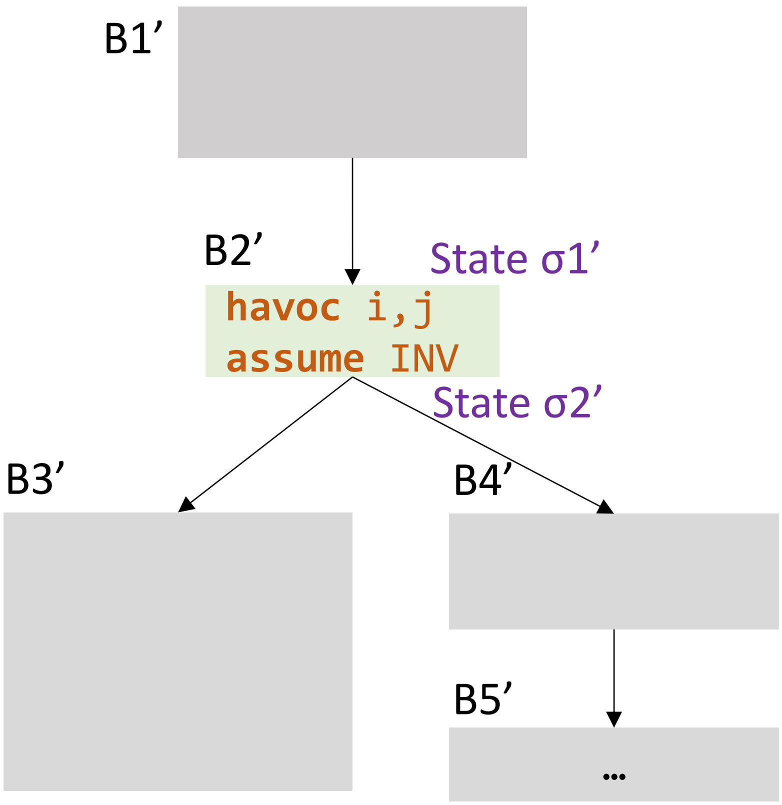
CFG-to-DAG validation: local results



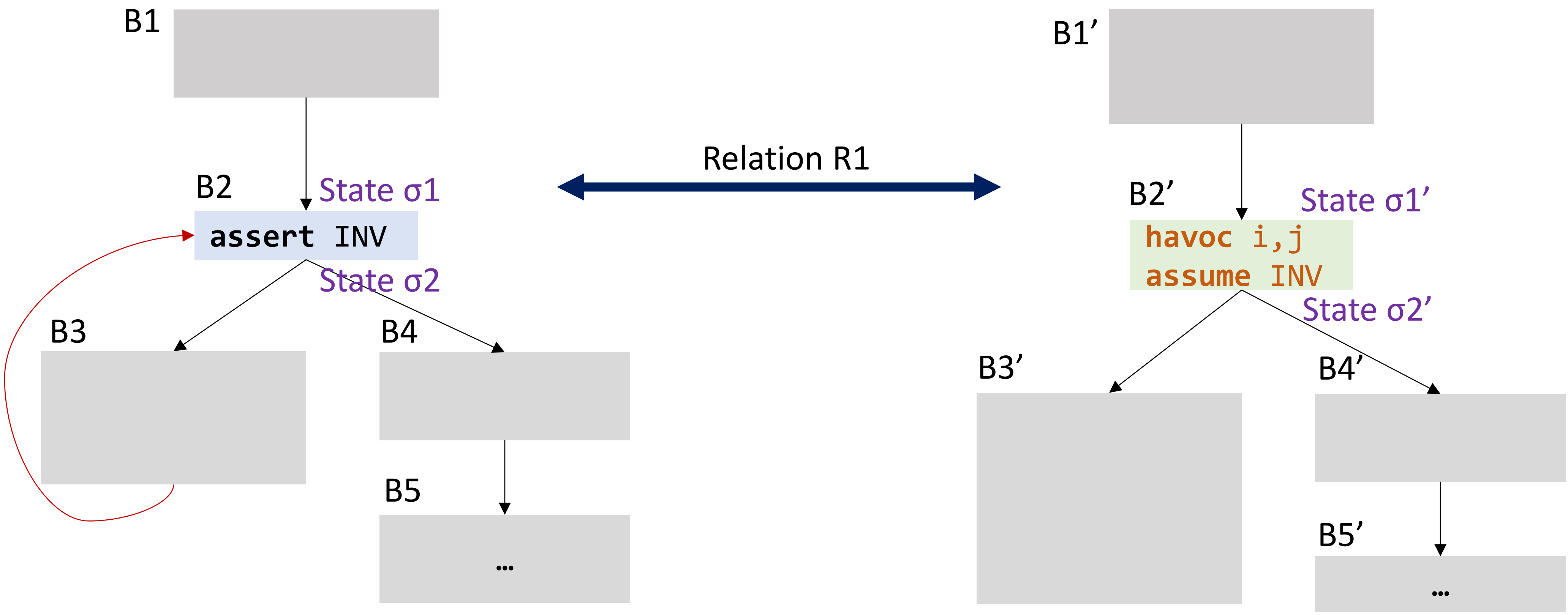


CFG-to-DAG validation: local results

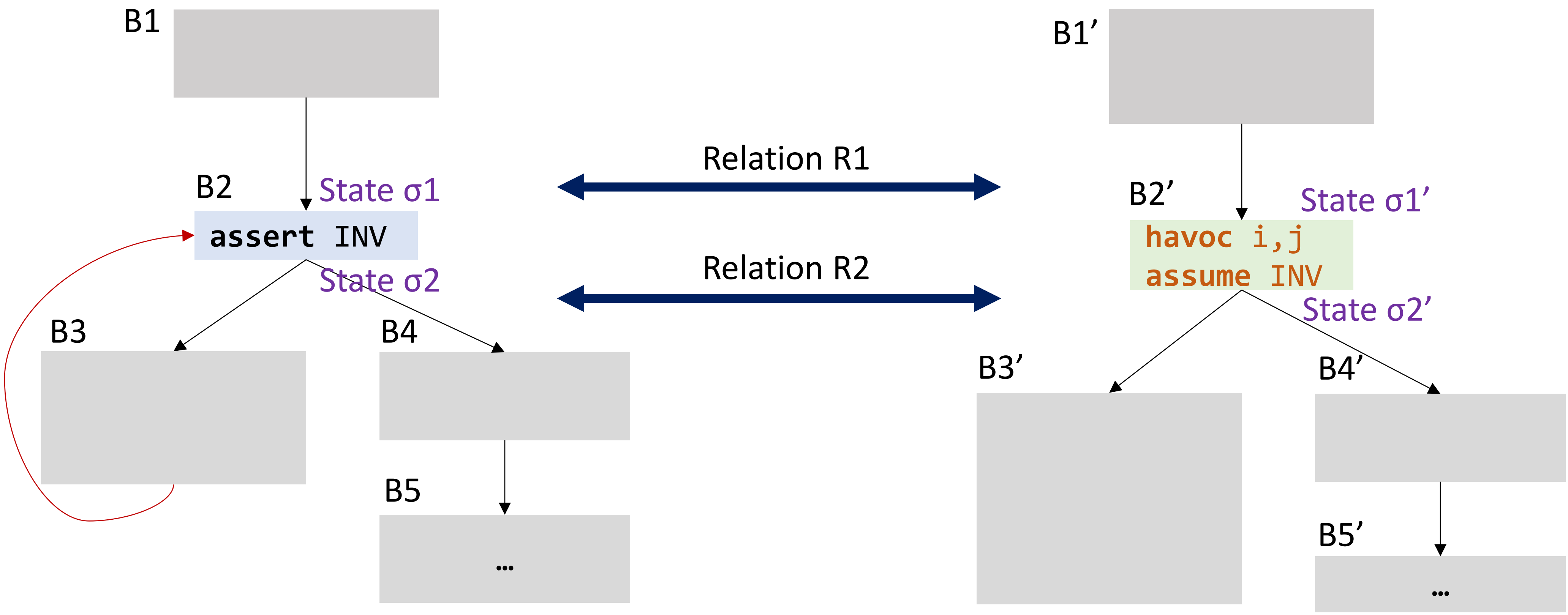




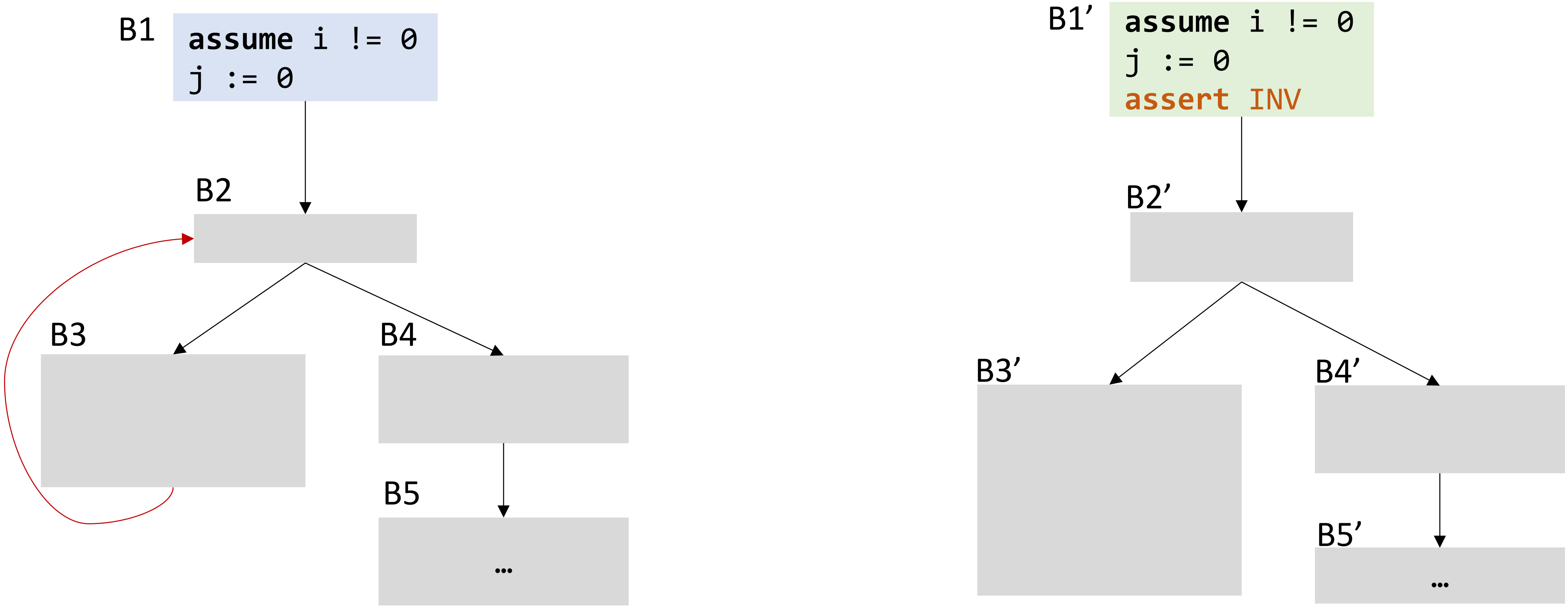
CFG-to-DAG validation: local results



CFG-to-DAG validation: local results



CFG-to-DAG validation: global result



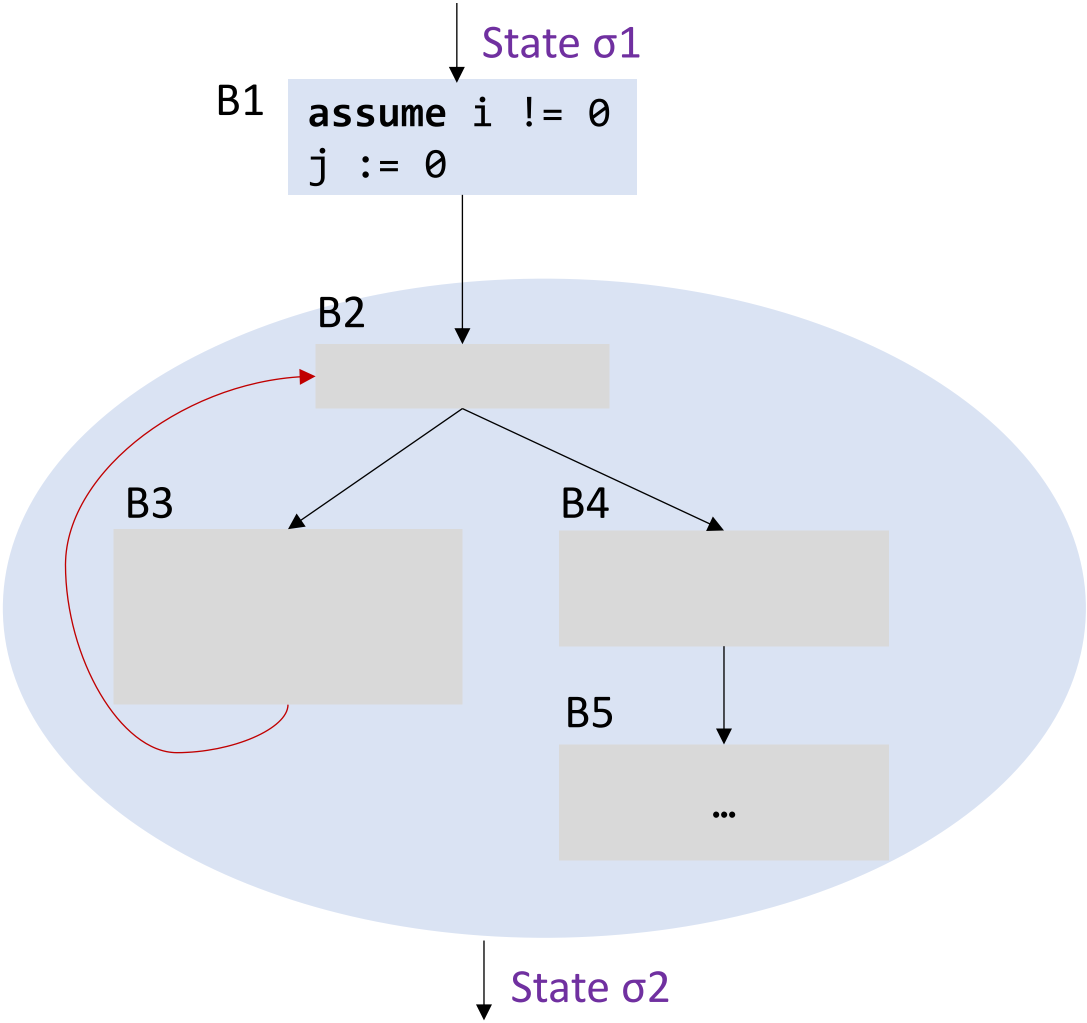
B1

```
assume i != 0
j := 0
```

B1’

```
assume i != 0
j := 0
assert INV
```

CFG-to-DAG validation: global result

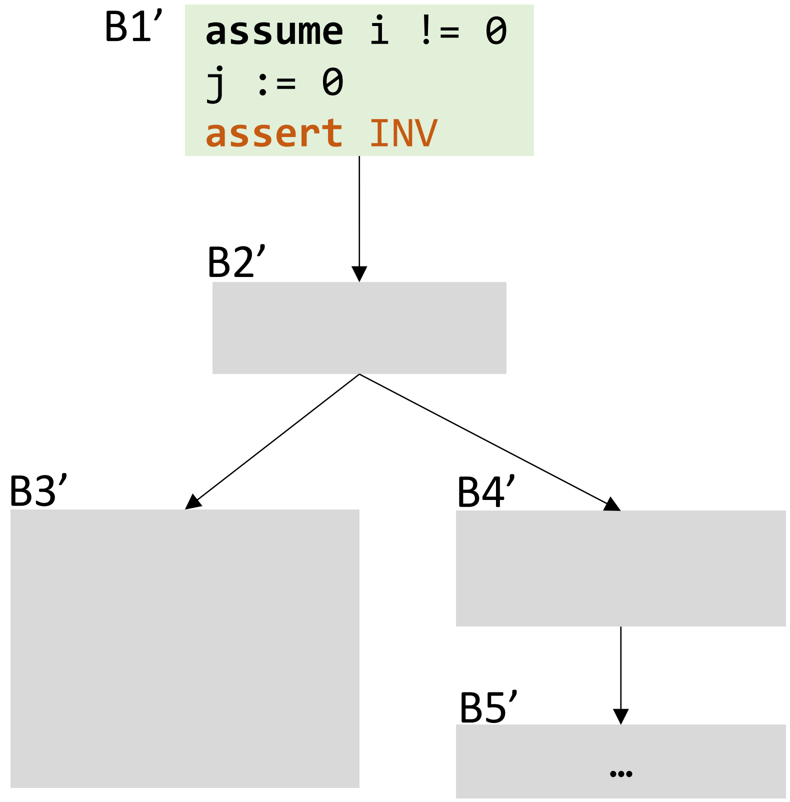


State σ1

B1

```
assume i != 0
j := 0
```

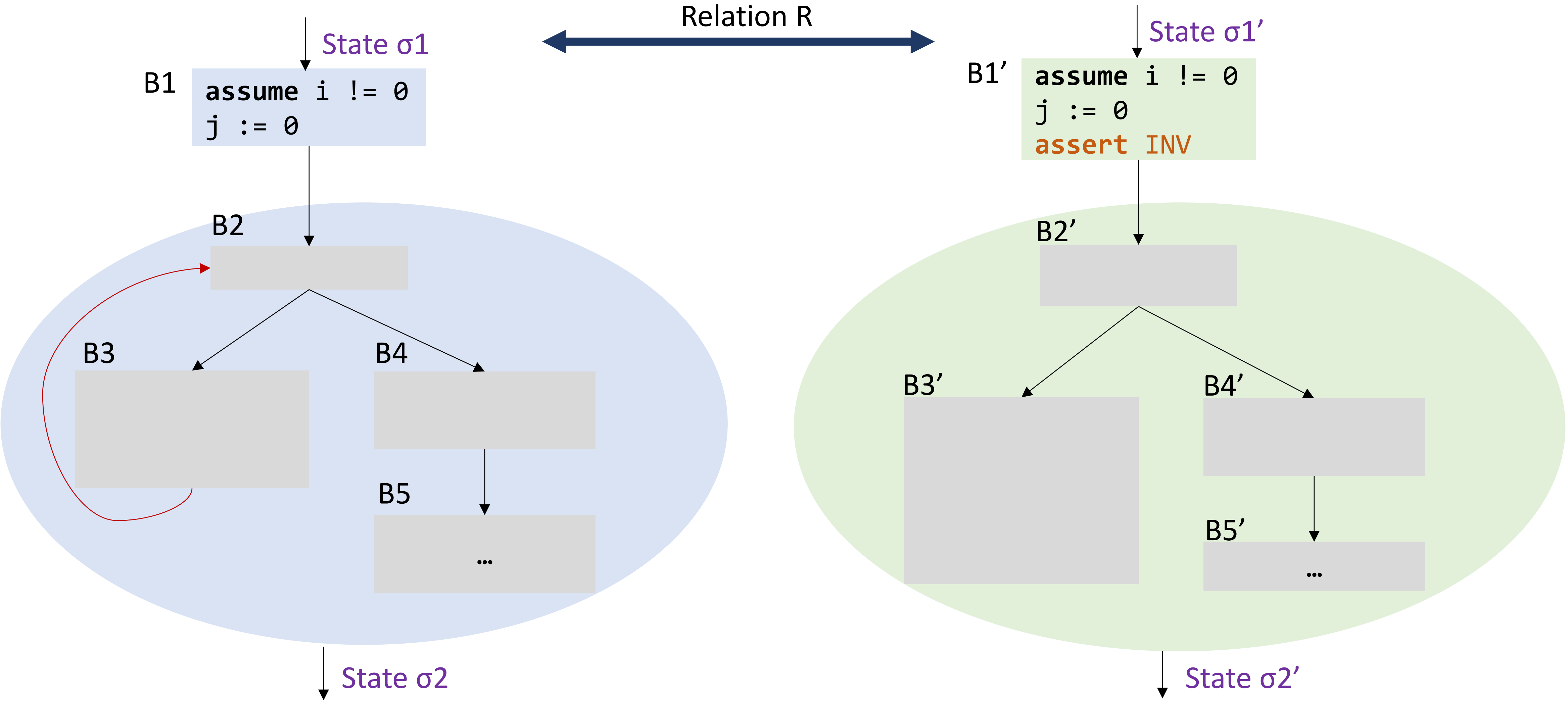
State σ2



B1’

```
assume i != 0
j := 0
assert INV
```

CFG-to-DAG validation: global result

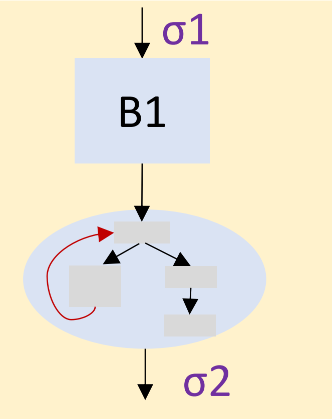


```
assume i != 0
j := 0
```

```
assume i != 0
j := 0
assert INV
```

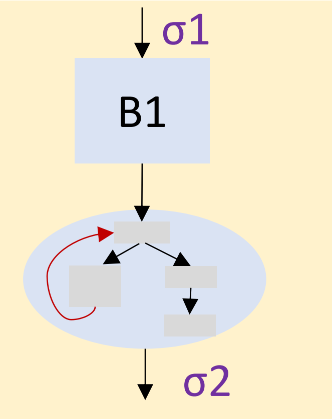
CFG-to-DAG validation: global block theorem

Assumptions



CFG-to-DAG validation: global block theorem

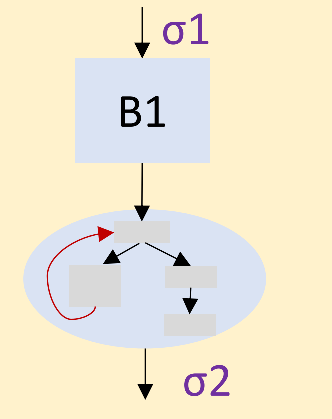
Assumptions



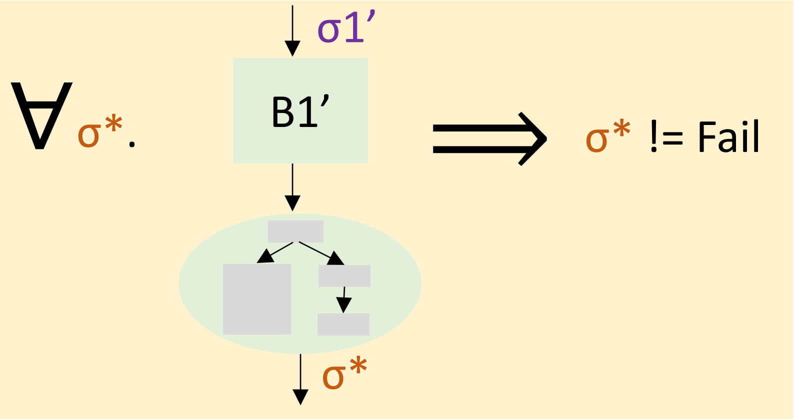
R(σ1, σ1’)

CFG-to-DAG validation: global block theorem

Assumptions

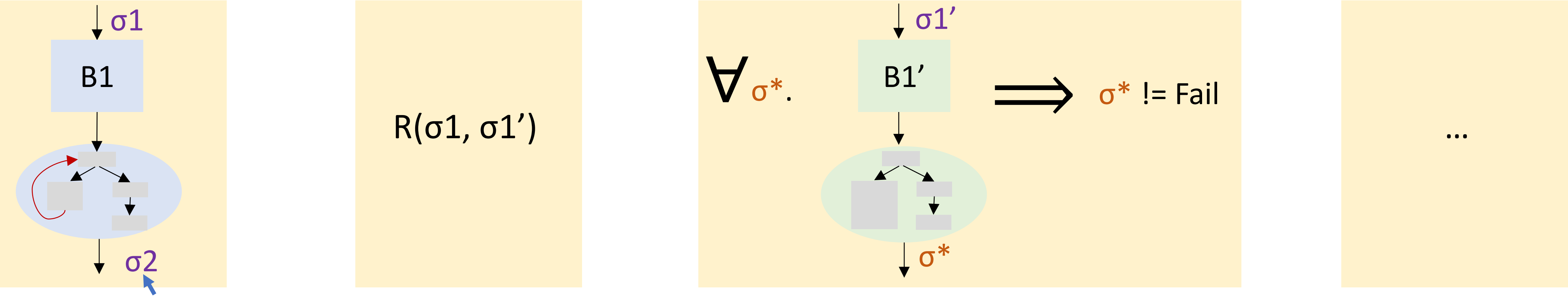


$R(\sigma1, \sigma1')$



CFG-to-DAG validation: global block theorem

Assumptions



σ1

B1

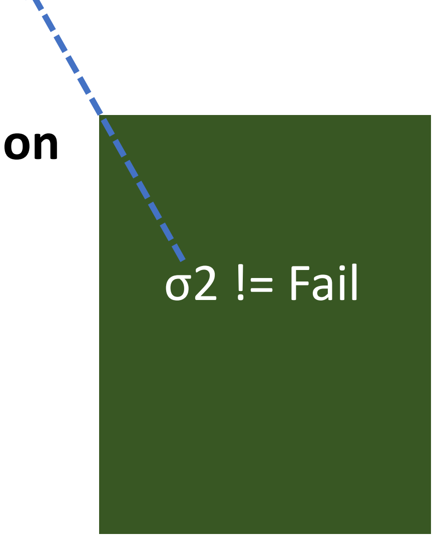
σ2

R(σ1, σ1’)

∀ σ*. σ1’ B1’ σ* ⟹ σ* != Fail

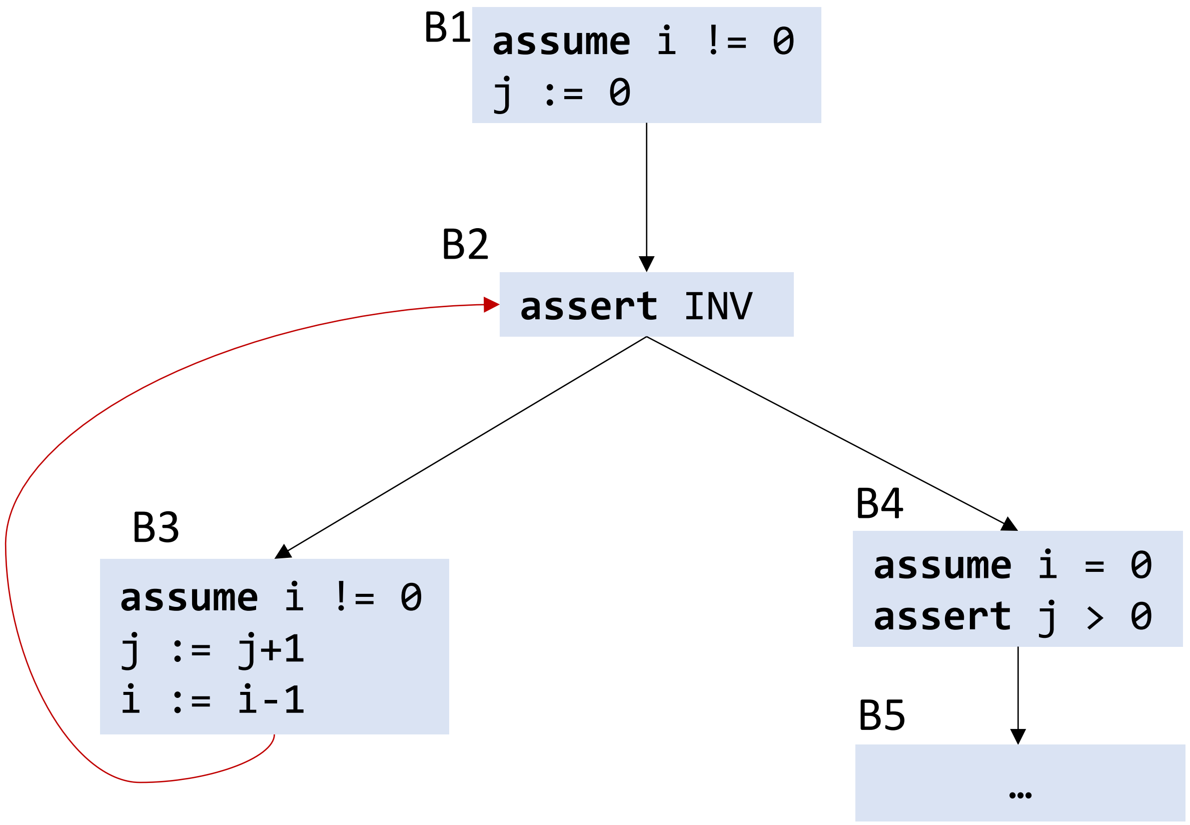
…

Conclusion

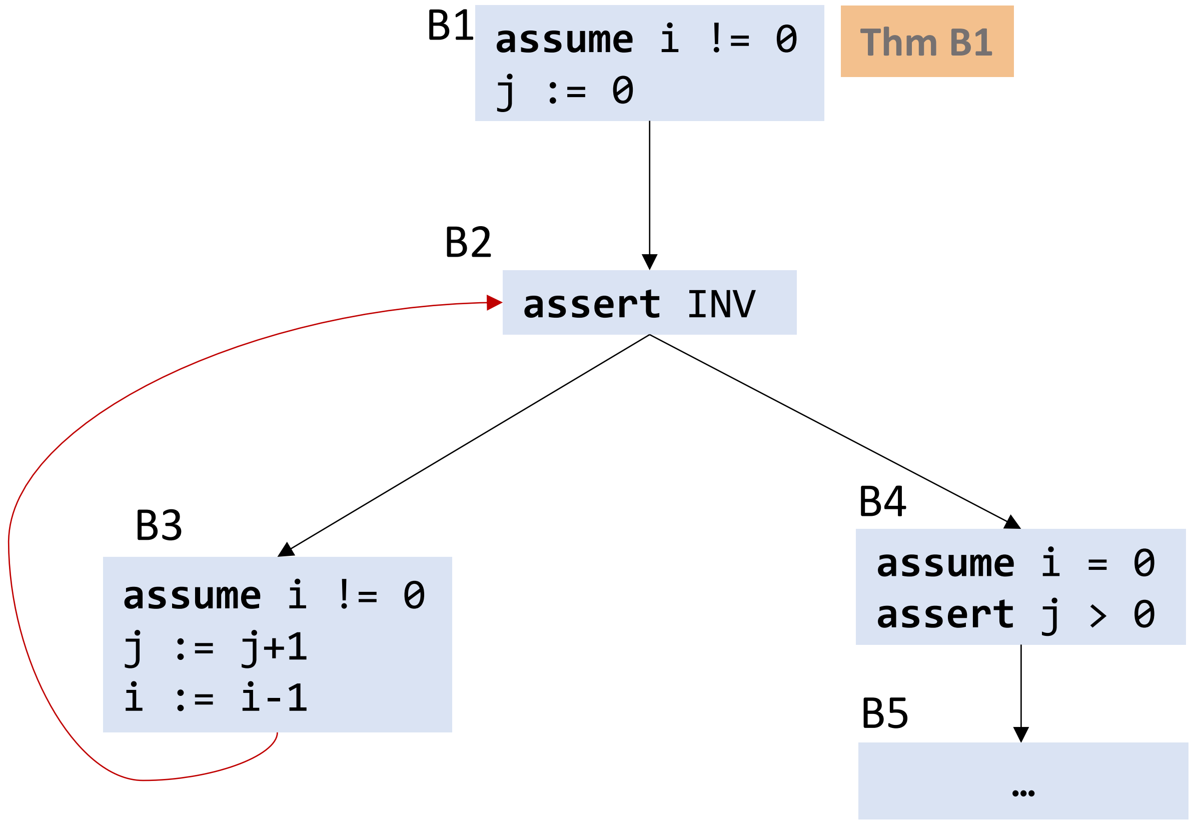


σ2 != Fail

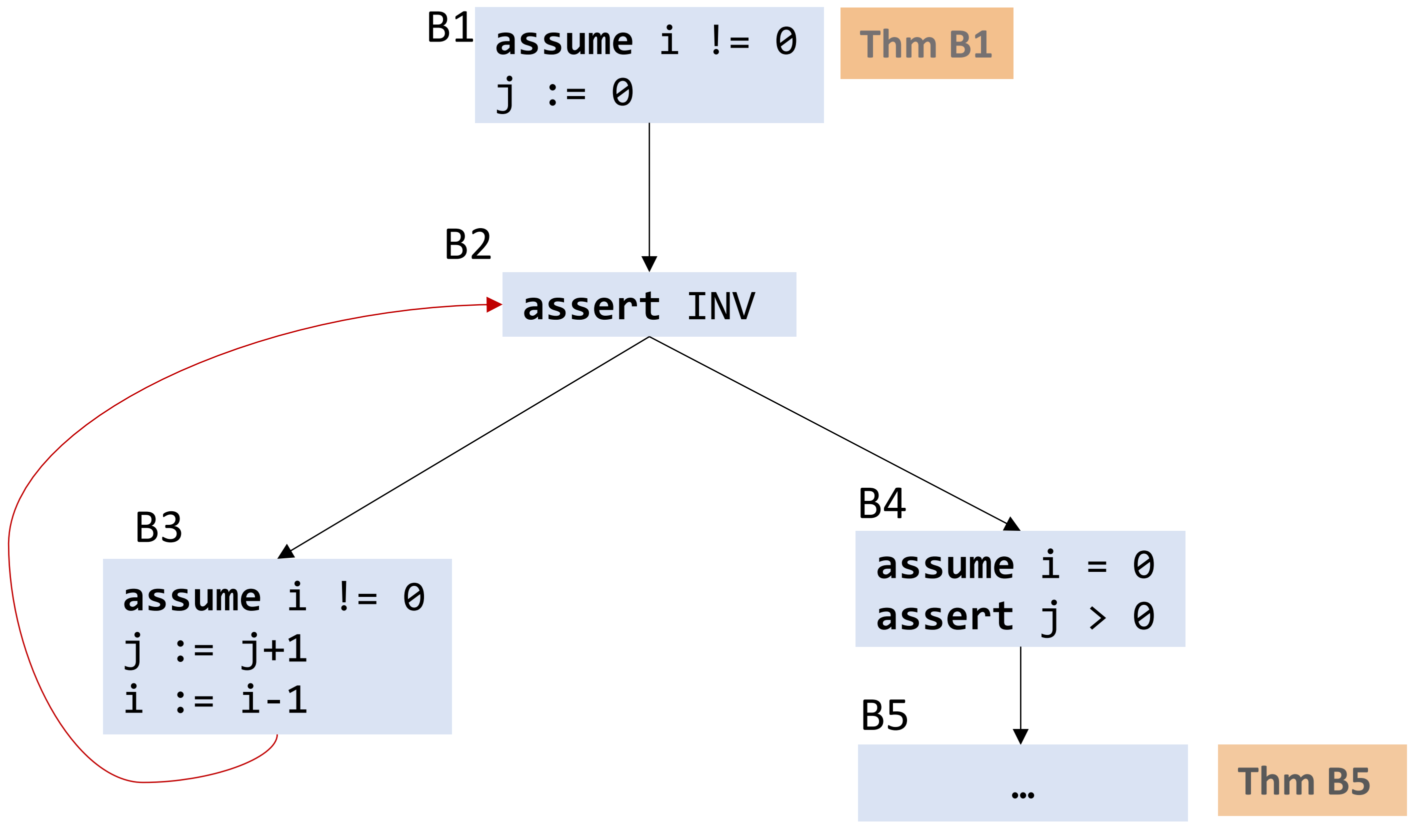
CFG-to-DAG validation: global proof strategy



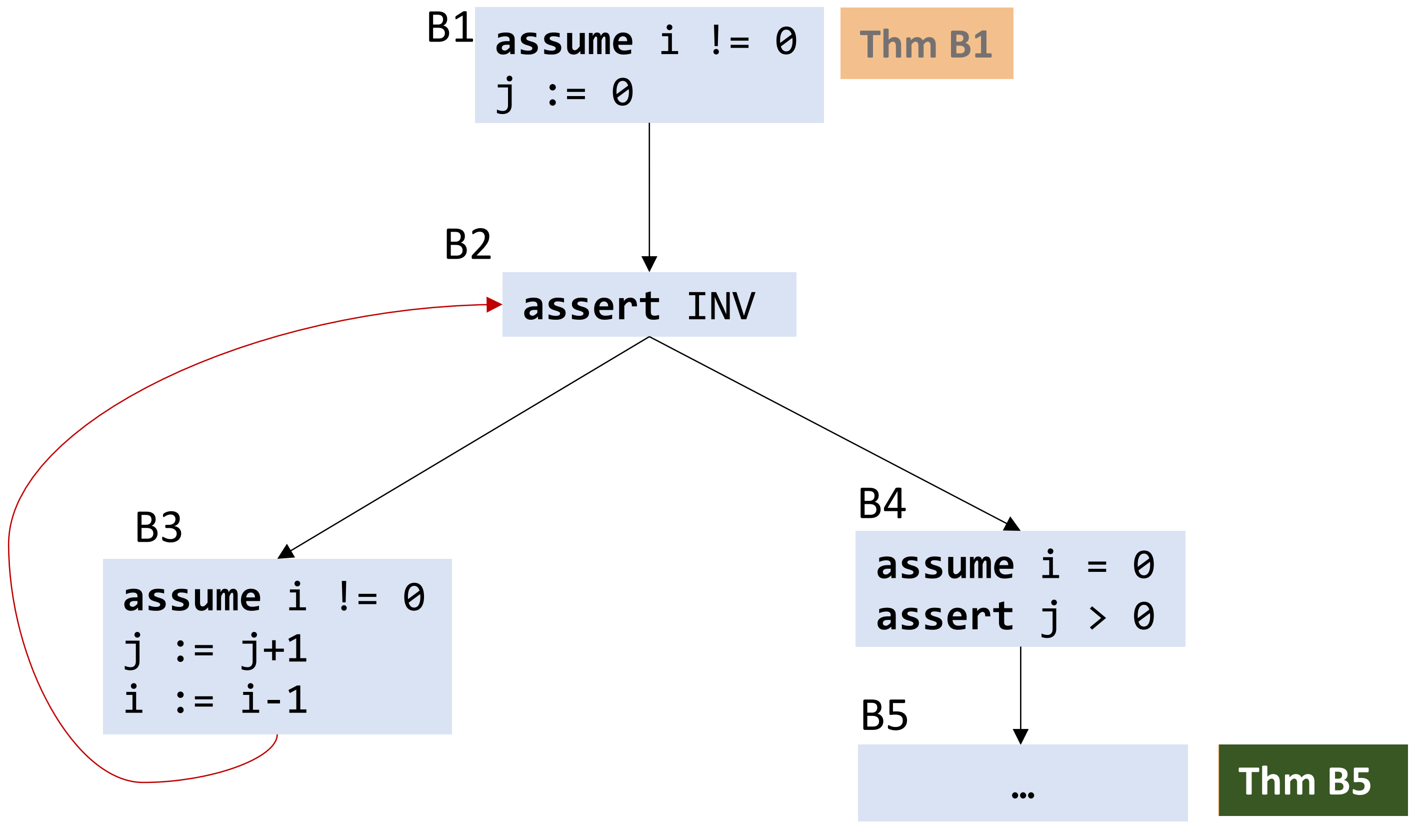
CFG-to-DAG validation: global proof strategy



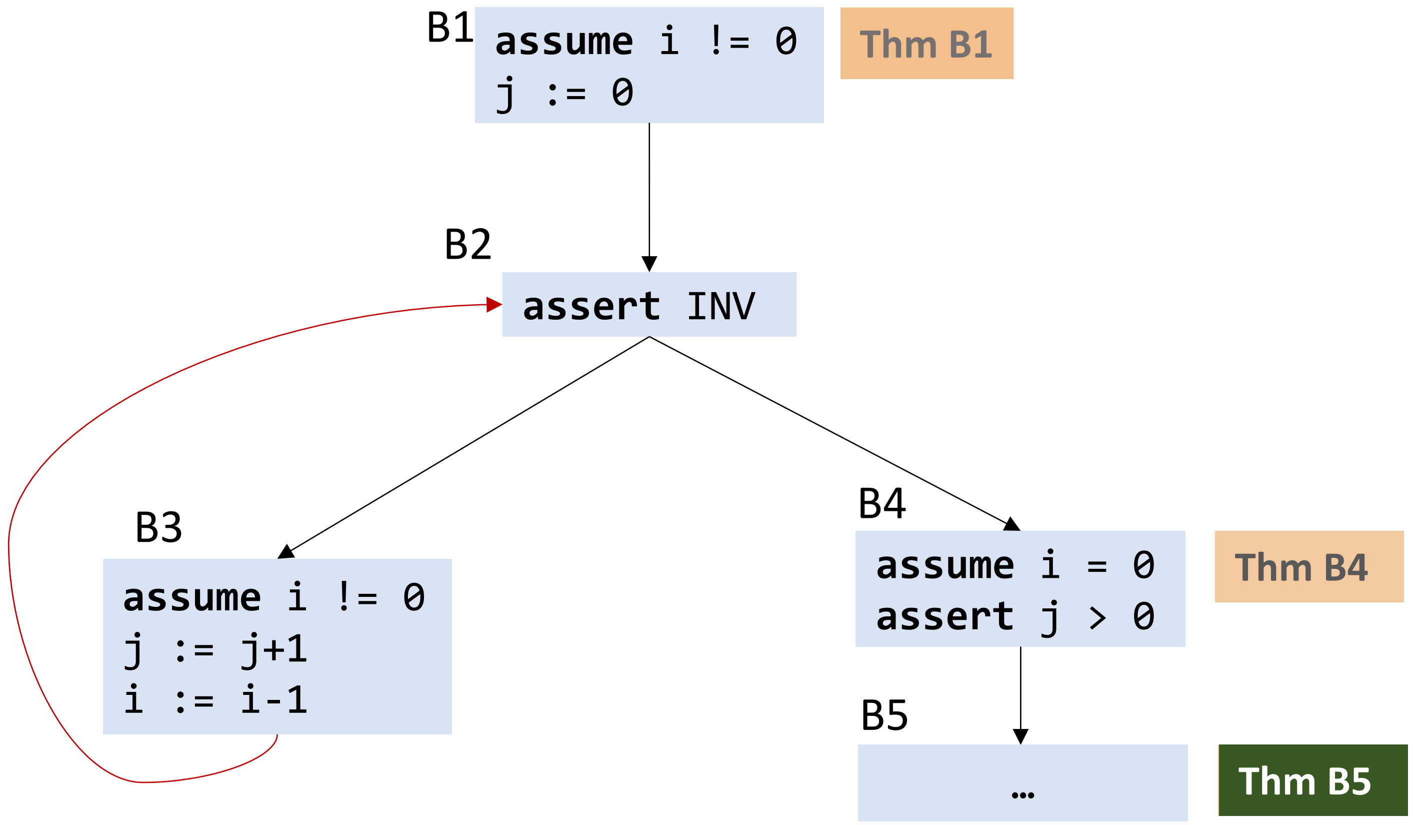
CFG-to-DAG validation: global proof strategy



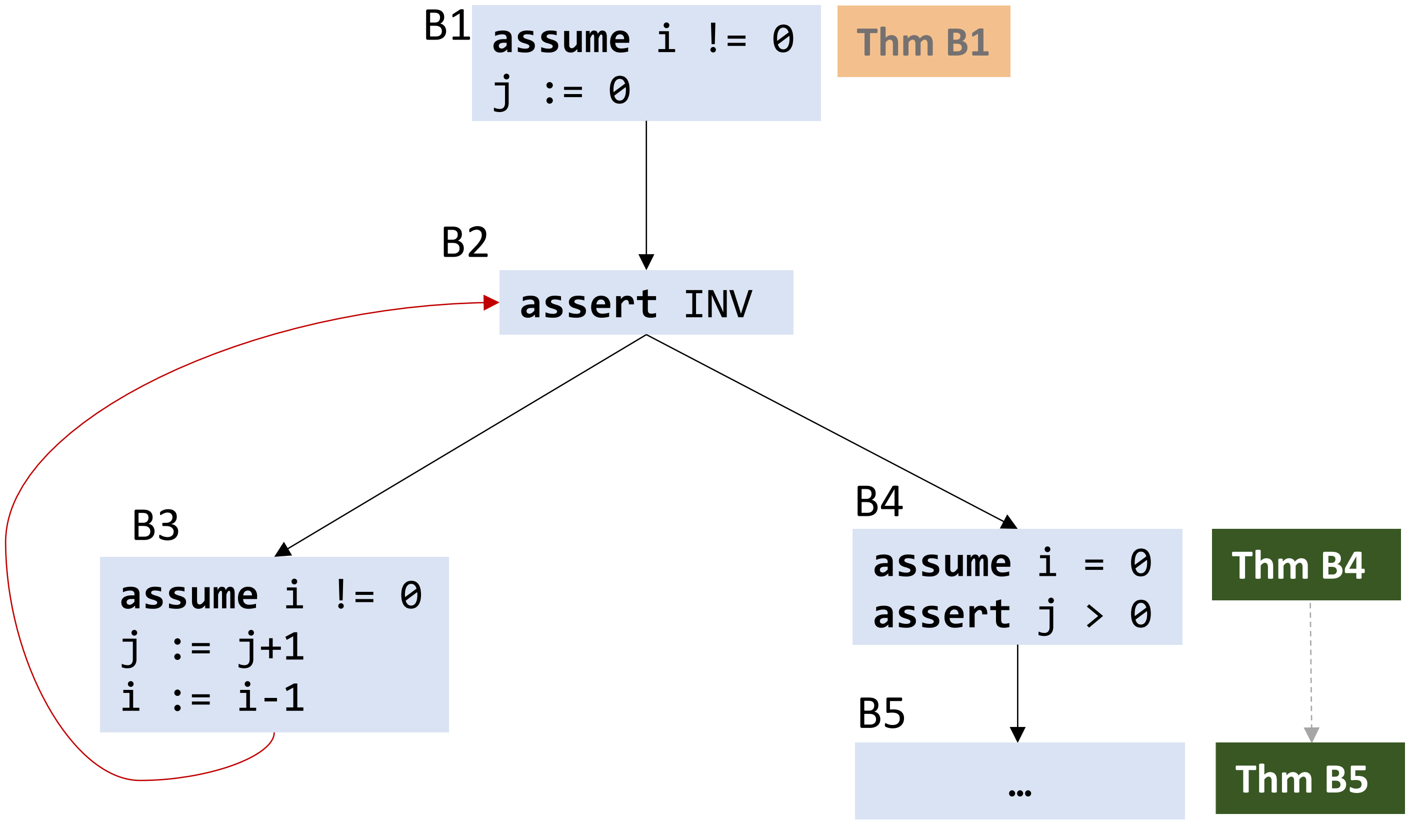
CFG-to-DAG validation: global proof strategy



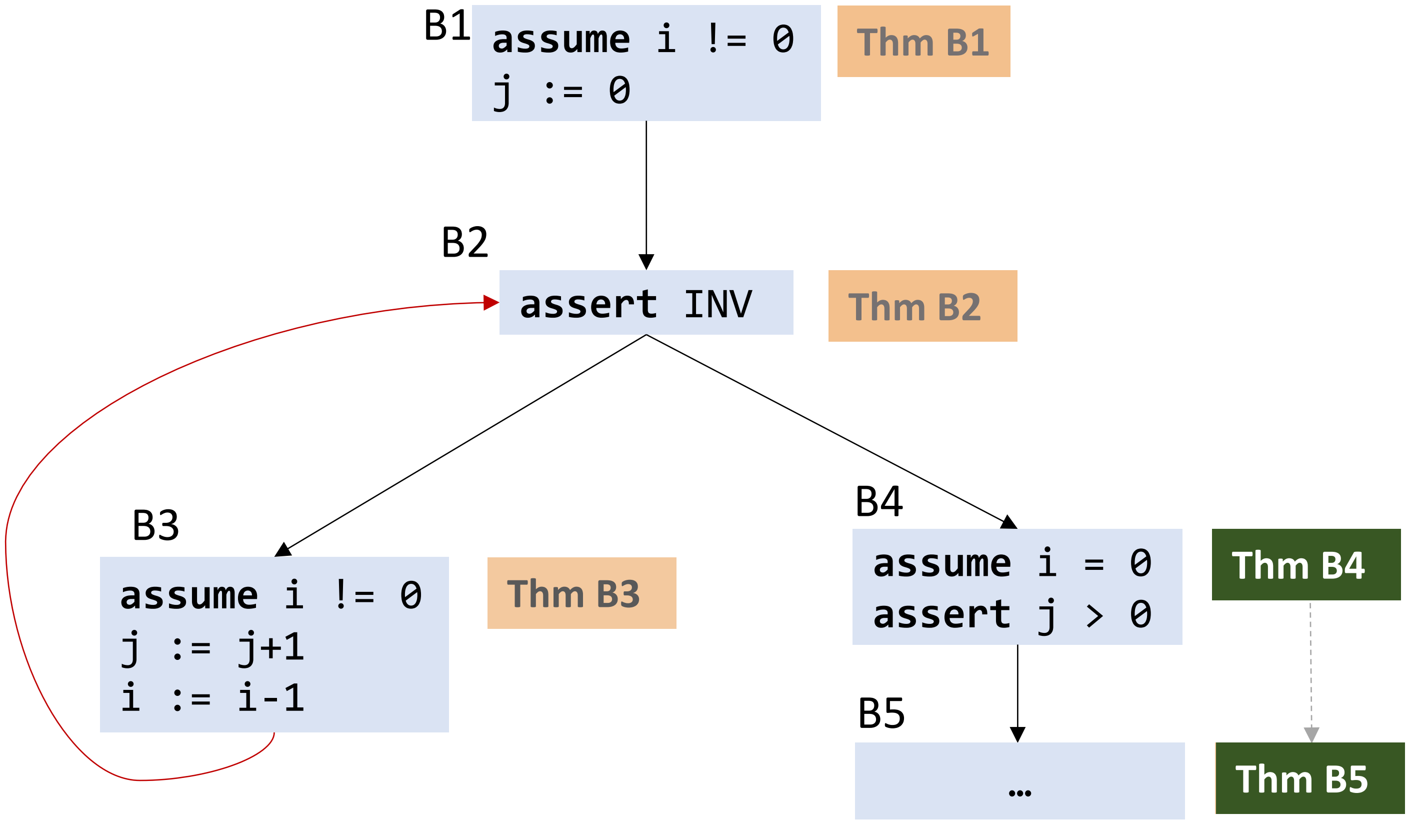
CFG-to-DAG validation: global proof strategy



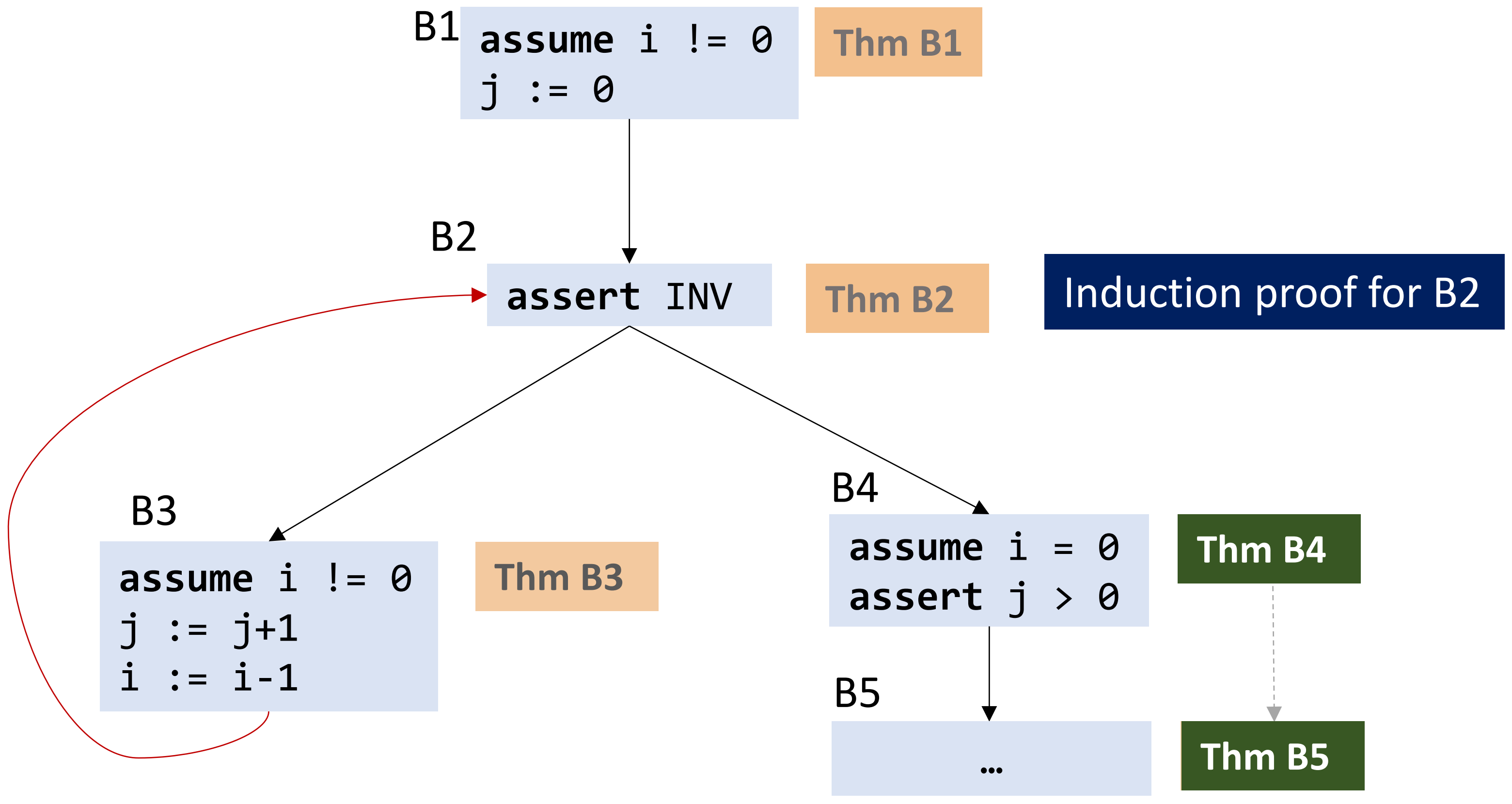
CFG-to-DAG validation: global proof strategy



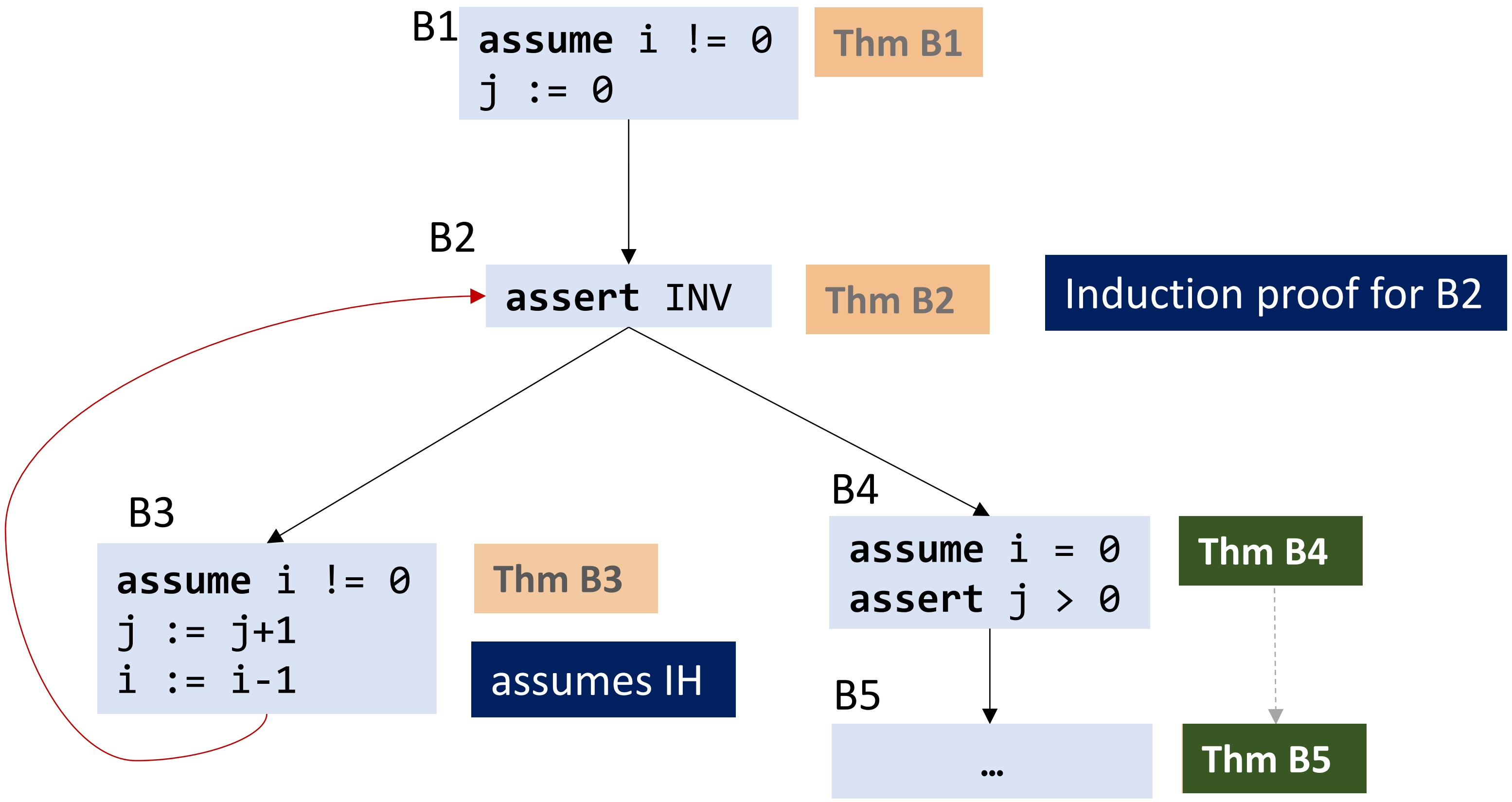
CFG-to-DAG validation: global proof strategy



CFG-to-DAG validation: global proof strategy

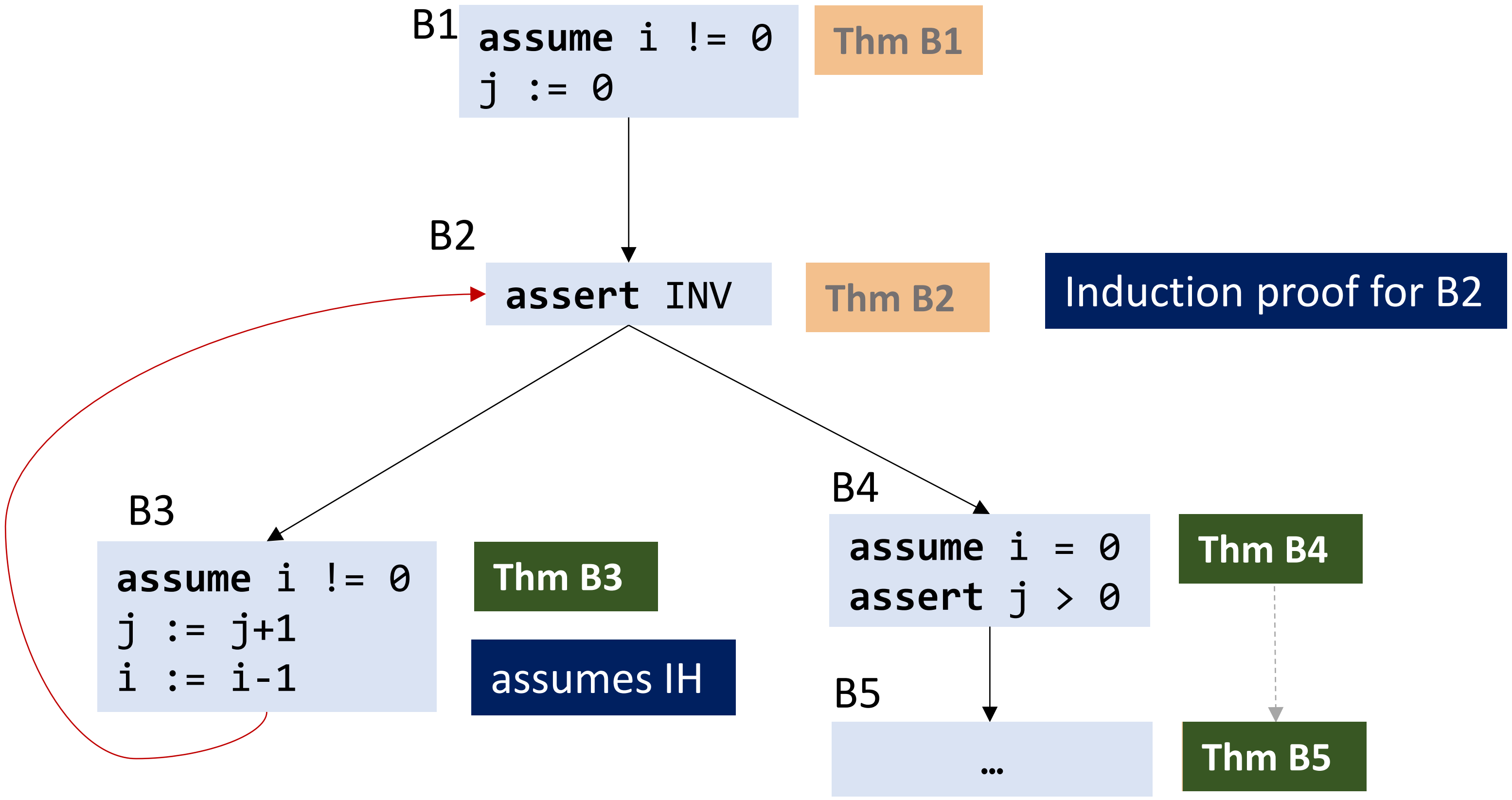


CFG-to-DAG validation: global proof strategy



```
B1:
assume i != 0
j := 0

B2:
assert INV

B3:
assume i != 0
j := j+1
i := i-1

B4:
assume i = 0
assert j > 0

B5:
…
```

CFG-to-DAG validation: global proof strategy



B1
```
assume i != 0
j := 0
```
Thm B1

B2
```
assert INV
```
Thm B2

Induction proof for B2

B3
```
assume i != 0
j := j+1
i := i-1
```
Thm B3

assumes IH

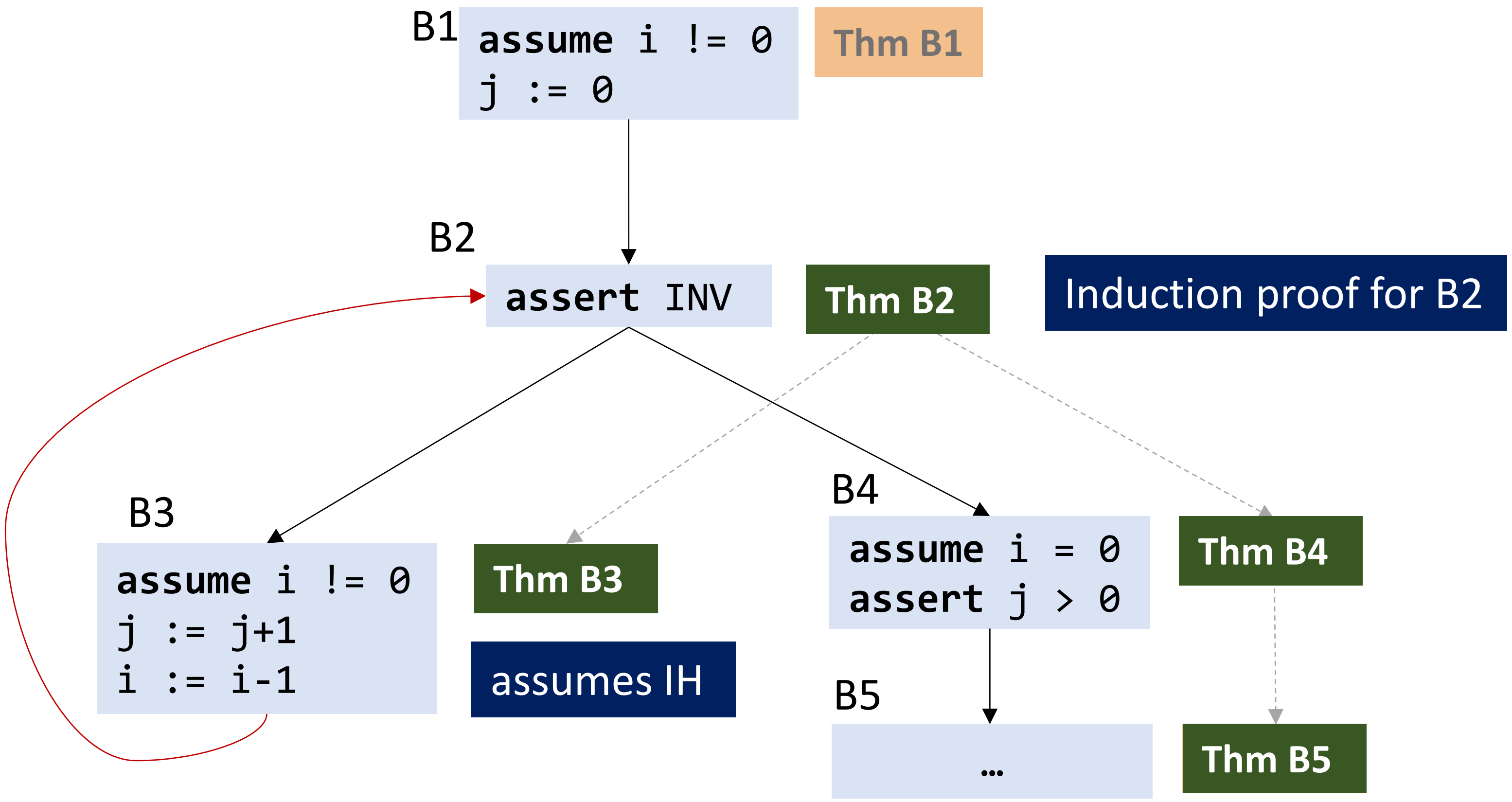
B4
```
assume i = 0
assert j > 0
```
Thm B4

B5

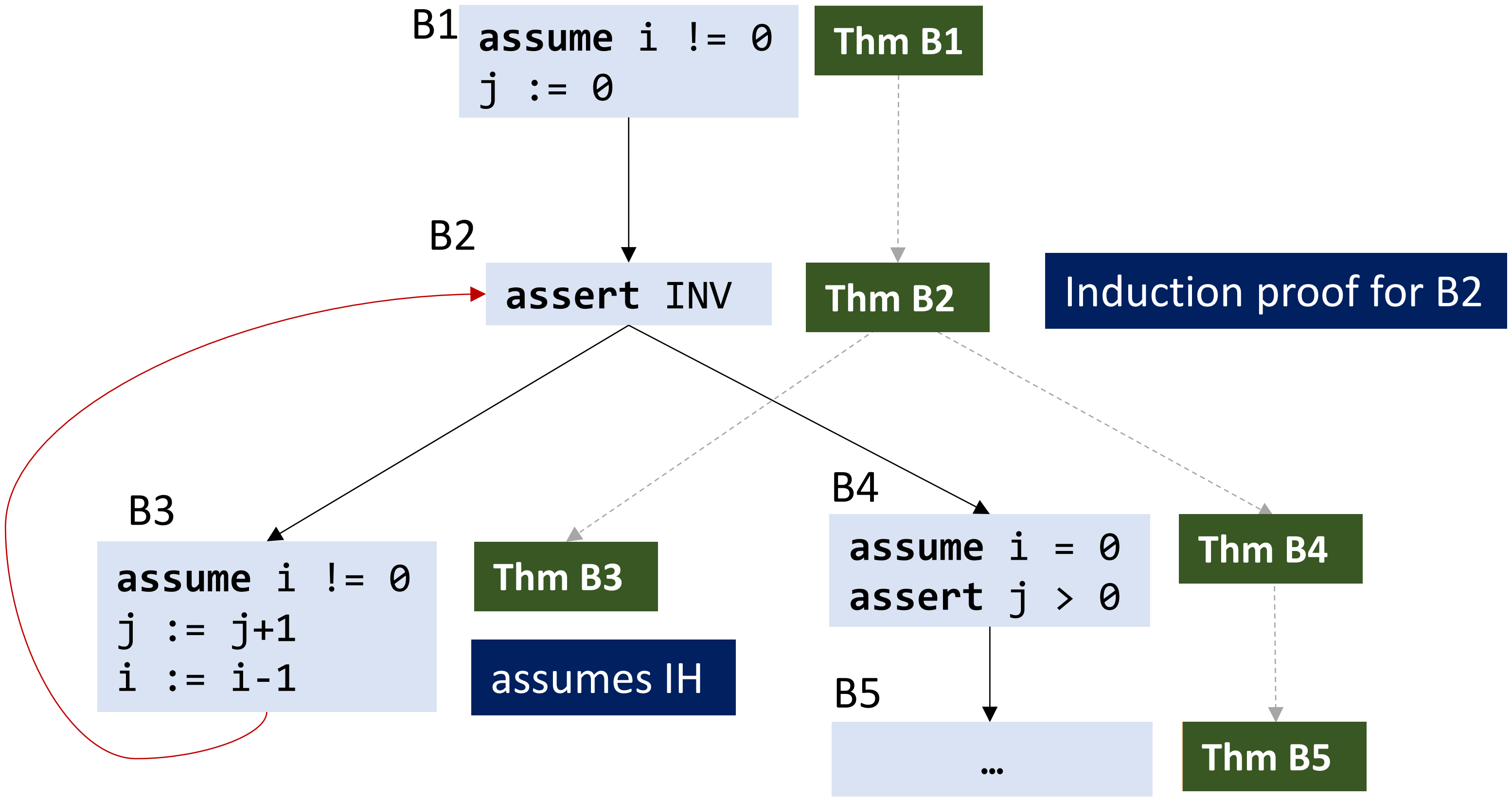
…

Thm B5

CFG-to-DAG validation: global proof strategy



CFG-to-DAG validation: global proof strategy



VC Phase

VC Phase

∀ VC quant . VC assumptions ⟹ CFG WP

VC Phase

$$\forall \underbrace{\text{VC quant}}_{\substack{\text{type encoding parameters,} \\ \text{Boogie variables,} \\ \ldots}} \,.\, \text{VC assumptions} \Longrightarrow \text{CFG WP}$$

VC Phase

$$\forall\ \underbrace{\text{VC quant}}_{\begin{array}{l}\text{type encoding parameters,}\\ \text{Boogie variables,}\\ \text{...}\end{array}}\ .\ \underbrace{\text{VC assumptions}}_{\begin{array}{l}\text{type encoding axiomatisation,}\\ \text{Boogie axioms,}\\ \text{...}\end{array}} \Longrightarrow \text{CFG WP}$$

VC Phase

$$\forall\ \underbrace{\text{VC quant}}_{\substack{\text{type encoding parameters,}\\ \text{Boogie variables,}\\ \text{…}}}\ .\ \underbrace{\text{VC assumptions}}_{\substack{\text{type encoding axiomatisation,}\\ \text{Boogie axioms,}\\ \text{…}}}\ \Longrightarrow\ \underbrace{\text{CFG WP}}_{\text{weakest precondition}}$$

Evaluation

functional
Artifact
CAV AEC
Evaluated
available reusable

Validated 96/100 examples from Boogie’s test suite (that verify and are in our subset)

Evaluation

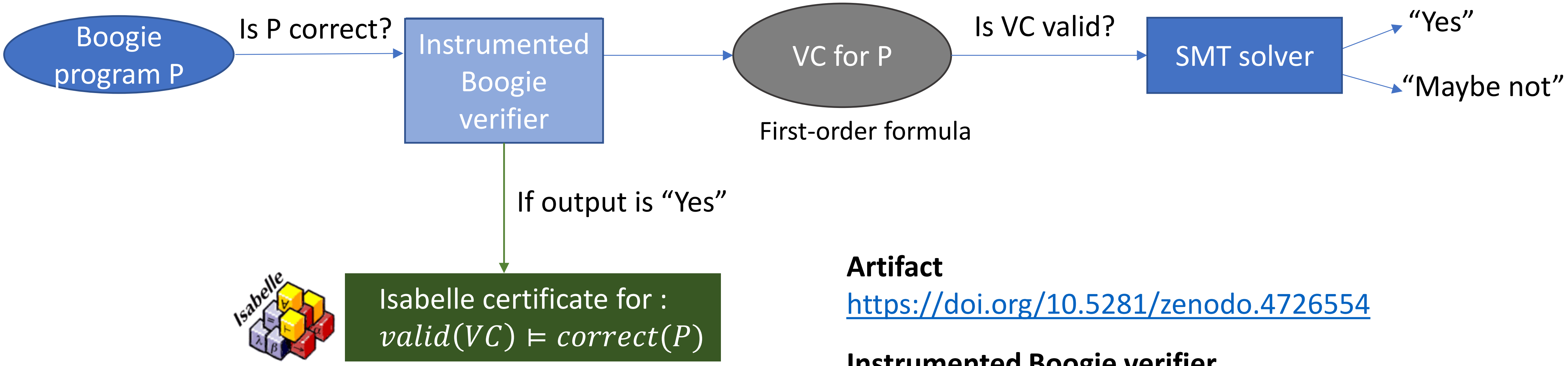
functional Artifact CAV AEC Evaluated available reusable

Validated 96/100 examples from Boogie’s test suite (that verify and are in our subset)

Validated 10 algorithmic examples

Program	LOC	# Procedures	Time [seconds]	Isabelle LOC
MaxOfArray	22	1	19.9	1944
Plateau	50	1	22.9	2019
DutchFlag	76	2	52.8	3994
…	…	…	…	…

Summary



Boogie program P

Is P correct?

Instrumented Boogie verifier

VC for P

First-order formula

Is VC valid?

SMT solver

“Yes”

“Maybe not”

If output is “Yes”

Isabelle certificate for :

$valid(VC) \vDash correct(P)$

Artifact
https://doi.org/10.5281/zenodo.4726554

Instrumented Boogie verifier
https://github.com/gauravpartha/boogie_proofgen/

Boogie formalisation
https://github.com/gauravpartha/foundational_boogie/