

Compiling WebAssembly Concolic Execution with Staging, Continuations, and Snapshots

Dinghong Zhong¹ Alexander Y. Bai² Mikail Khan³ Guannan Wei¹

¹Tufts University

²New York University

³Carnegie Mellon University

OOPSLA 2026

Compiling WebAssembly **Concolic Execution** with Staging, Continuations, and Snapshots

Dinghong Zhong¹ Alexander Y. Bai² Mikail Khan³ Guannan Wei¹

¹Tufts University

²New York University

³Carnegie Mellon University

OOPSLA 2026

Concolic Execution

Concolic execution = **con**crete execution + sym**bol**ic execution

Concolic Execution

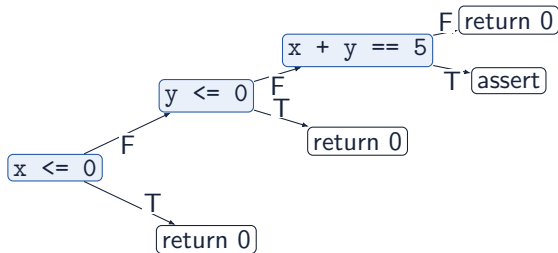
Concolic execution = **con**crete execution + **symbolic** execution

```
int main() {  
    int x = input();  
    int y = input();  
    if (x <= 0)  
        return 0;  
    else if (y <= 0)  
        return 0;  
    else if (x + y == 5)  
        return assert(false);  
    else  
        return 0;  
}
```

Concolic Execution

Concolic execution = **concrete** execution + **symbolic** execution

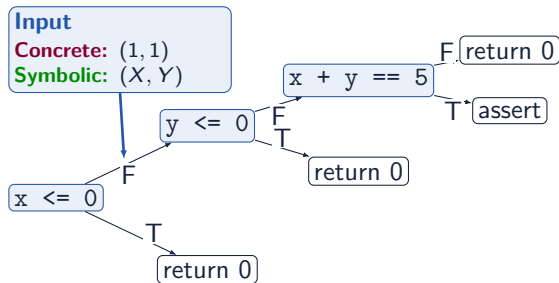
```
int main() {  
    int x = input();  
    int y = input();  
    if (x <= 0)  
        return 0;  
    else if (y <= 0)  
        return 0;  
    else if (x + y == 5)  
        return assert(false);  
    else  
        return 0;  
}
```



Concolic Execution

Concolic execution = **concrete** execution + **symbolic** execution

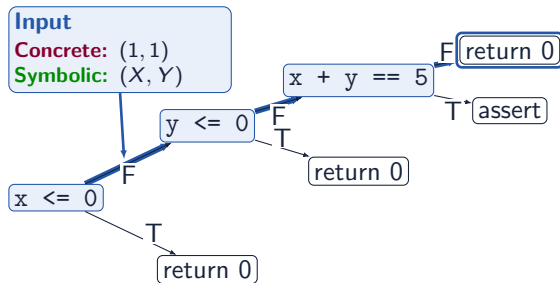
```
int main() {  
    int x = input();  
    int y = input();  
    if (x <= 0)  
        return 0;  
    else if (y <= 0)  
        return 0;  
    else if (x + y == 5)  
        return assert(false);  
    else  
        return 0;  
}
```



Concolic Execution

Concolic execution = **concrete** execution + **symbolic** execution

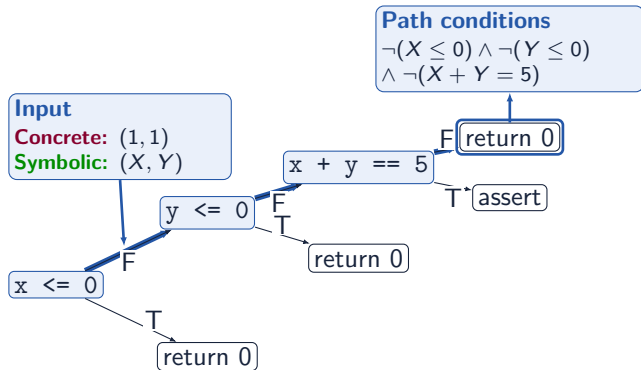
```
int main() {  
    int x = input();  
    int y = input();  
    if (x <= 0)  
        return 0;  
    else if (y <= 0)  
        return 0;  
    else if (x + y == 5)  
        return assert(false);  
    else  
        return 0;  
}
```



Concolic Execution

Concolic execution = **concrete** execution + **symbolic** execution

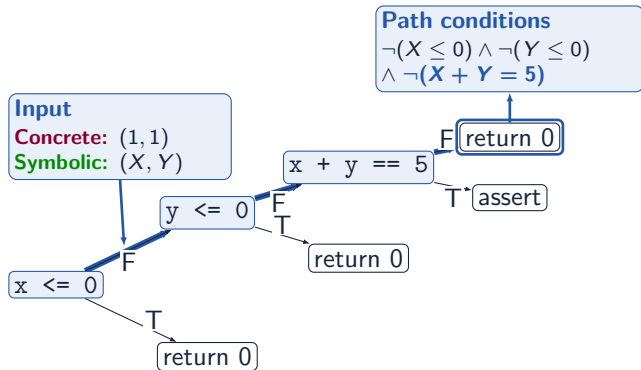
```
int main() {  
    int x = input();  
    int y = input();  
    if (x <= 0)  
        return 0;  
    else if (y <= 0)  
        return 0;  
    else if (x + y == 5)  
        return assert(false);  
    else  
        return 0;  
}
```



Concolic Execution

Concolic execution = **concrete** execution + **symbolic** execution

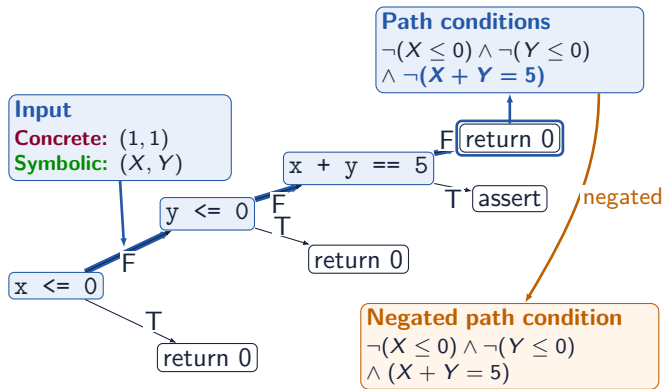
```
int main() {  
    int x = input();  
    int y = input();  
    if (x <= 0)  
        return 0;  
    else if (y <= 0)  
        return 0;  
    else if (x + y == 5)  
        return assert(false);  
    else  
        return 0;  
}
```



Concolic Execution

Concolic execution = **concrete** execution + **symbolic** execution

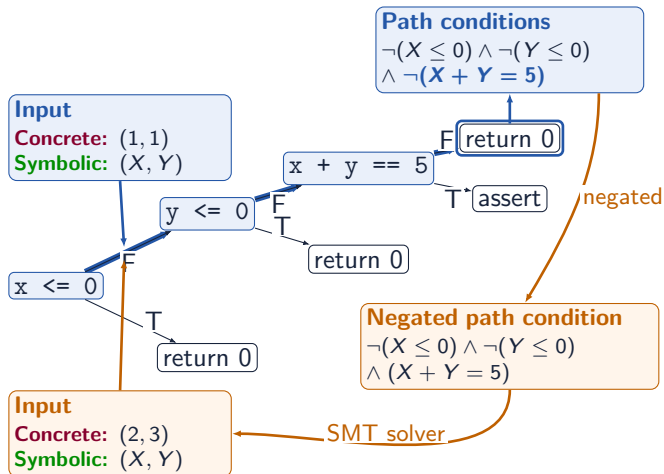
```
int main() {  
    int x = input();  
    int y = input();  
    if (x <= 0)  
        return 0;  
    else if (y <= 0)  
        return 0;  
    else if (x + y == 5)  
        return assert(false);  
    else  
        return 0;  
}
```



Concolic Execution

Concolic execution = **concrete** execution + **symbolic** execution

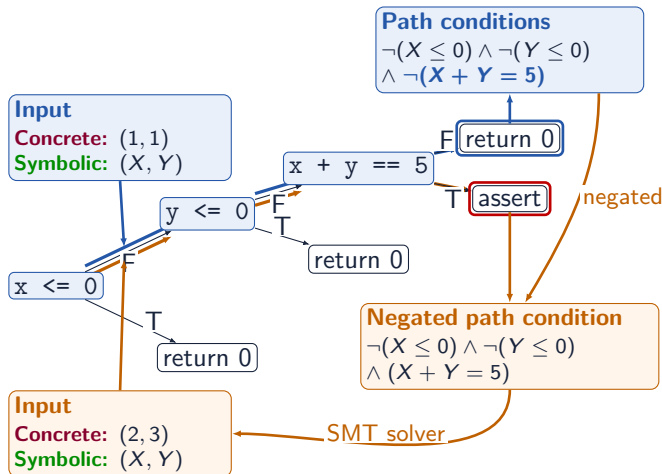
```
int main() {  
    int x = input();  
    int y = input();  
    if (x <= 0)  
        return 0;  
    else if (y <= 0)  
        return 0;  
    else if (x + y == 5)  
        return assert(false);  
    else  
        return 0;  
}
```



Concolic Execution

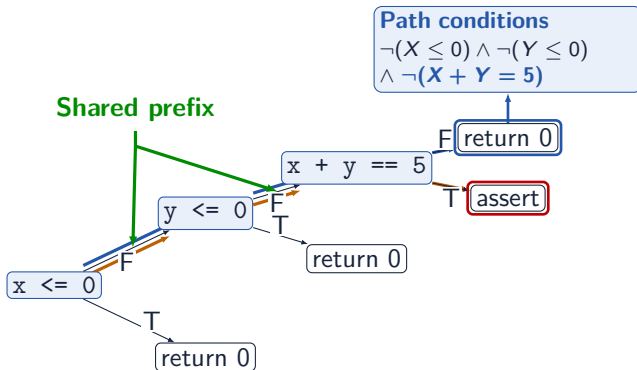
Concolic execution = **concrete** execution + **symbolic** execution

```
int main() {  
    int x = input();  
    int y = input();  
    if (x <= 0)  
        return 0;  
    else if (y <= 0)  
        return 0;  
    else if (x + y == 5)  
        return assert(false);  
    else  
        return 0;  
}
```



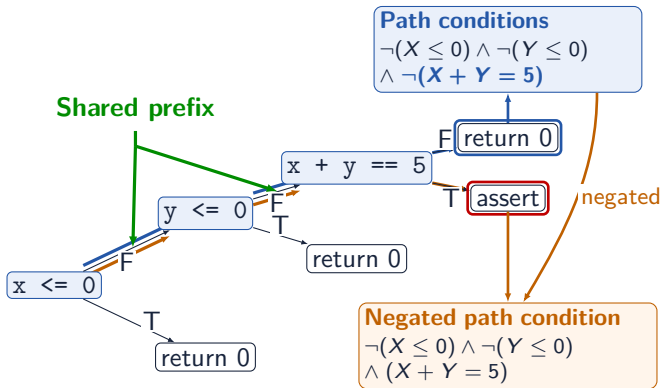
Snapshot Reuse

Explored paths share a prefix.



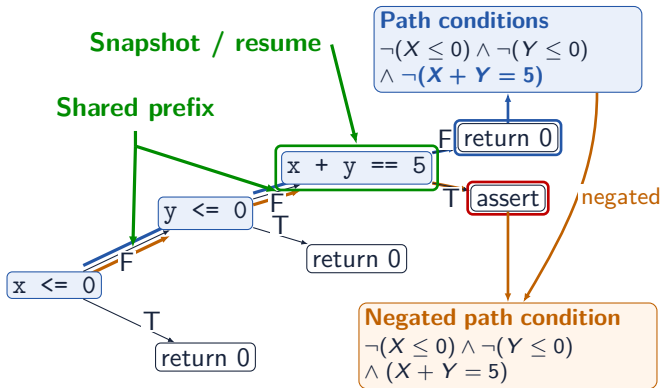
Snapshot Reuse

Explored paths share a prefix.



Snapshot Reuse

Explored paths share a prefix.



Resume from a deeper branch point $\implies$ **longer prefix, more work saved**

Implementing Snapshot Reuse in an Interpreter

$$\text{Snapshot} = \underbrace{\text{Program state} + \text{Control flow}}_{\text{The interpreter has full control}}$$

Implementing Snapshot Reuse in an Interpreter

$$\text{Snapshot} = \underbrace{\text{Program state} + \text{Control flow}}_{\text{The interpreter has full control}}$$



Implementing Snapshot Reuse in an Interpreter

$$\text{Snapshot} = \underbrace{\text{Program state} + \text{Control flow}}_{\text{The interpreter has full control}}$$



However, **it is slow due to interpretation overhead.**

Resolving the Tension Between Two Families

Interpretation-based

Instrumentation-based



- ▶ **Snapshot reuse:** the interpreter owns the state and control.
- ▶ **Slow:** interpretation overhead.

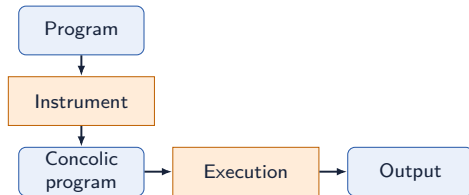
Resolving the Tension Between Two Families

Interpretation-based



- ▶ **Snapshot reuse:** the interpreter owns the state and control.
- ▶ **Slow:** interpretation overhead.

Instrumentation-based



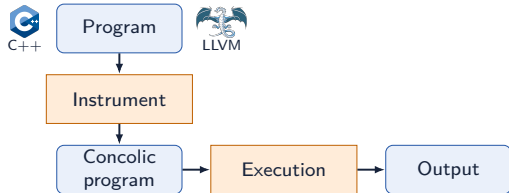
Resolving the Tension Between Two Families

Interpretation-based



- ▶ **Snapshot reuse:** the interpreter owns the state and control.
- ▶ **Slow:** interpretation overhead.

Instrumentation-based



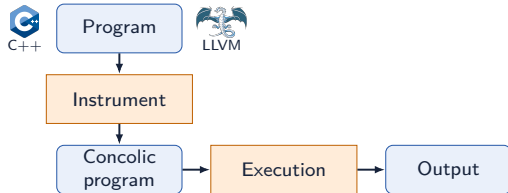
Resolving the Tension Between Two Families

Interpretation-based



- ▶ **Snapshot reuse:** the interpreter owns the state and control.
- ▶ **Slow:** interpretation overhead.

Instrumentation-based



- ▶ **Efficient:** no interpretation overhead.
- ▶ **No snapshot reuse:** starts from the beginning every time.

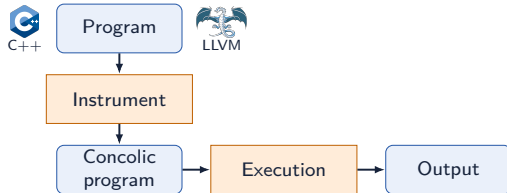
Resolving the Tension Between Two Families

Interpretation-based



- ▶ **Snapshot reuse:** the interpreter owns the state and control.
- ▶ **Slow:** interpretation overhead.

Instrumentation-based



- ▶ **Efficient:** no interpretation overhead.
- ▶ **No snapshot reuse:** starts from the beginning every time.

How can we get the best of both worlds?

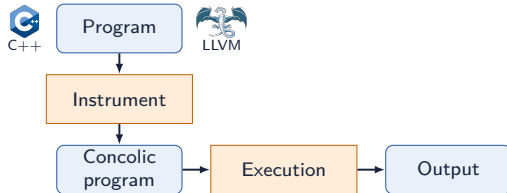
Resolving the Tension Between Two Families

Interpretation-based



- ▶ **Snapshot reuse:** the interpreter owns the state and control.
- ▶ **Slow:** interpretation overhead.

Instrumentation-based



- ▶ **Efficient:** no interpretation overhead.
- ▶ **No snapshot reuse:** starts from the beginning every time.

Compiling with Continuations
Efficiency Snapshot reuse

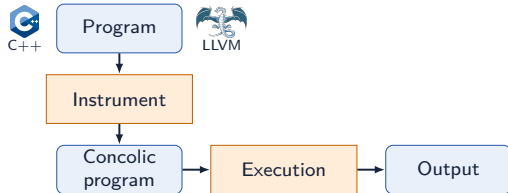
Resolving the Tension Between Two Families

Interpretation-based



- ▶ **Snapshot reuse:** the interpreter owns the state and control.
- ▶ **Slow:** interpretation overhead.

Instrumentation-based



- ▶ **Efficient:** no interpretation overhead.
- ▶ **No snapshot reuse:** starts from the beginning every time.

Compiling with **Continuations**
Efficiency Snapshot reuse
via **Staging**
Simplicity

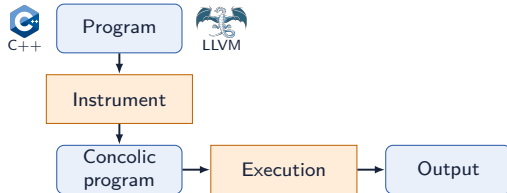
Resolving the Tension Between Two Families

Interpretation-based



- ▶ **Snapshot reuse:** the interpreter owns the state and control.
- ▶ **Slow:** interpretation overhead.

Instrumentation-based



- ▶ **Efficient:** no interpretation overhead.
- ▶ **No snapshot reuse:** starts from the beginning every time.

Compiling with Continuations
Efficiency Snapshot reuse
via Staging for WebAssembly
Simplicity

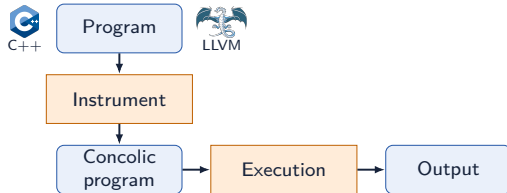
Resolving the Tension Between Two Families

Interpretation-based



- ▶ **Snapshot reuse:** the interpreter owns the state and control.
- ▶ **Slow:** interpretation overhead.

Instrumentation-based



- ▶ **Efficient:** no interpretation overhead.
- ▶ **No snapshot reuse:** starts from the beginning every time.

Compiling with Continuations

Efficiency

Snapshot reuse

via Staging for WebAssembly

Simplicity



Contributions

1. **A compilation-based architecture for efficient concolic execution with snapshot reuse**

From CPS semantics to staged concolic semantics with snapshot reuse

2. **GenWasm**: a concolic testing tool for WebAssembly

44.9× speedup over the state of the art

Key Idea 1: CPS-Based Concolic Semantics

Capturing Control

What must we save to resume here without repeating the prefix?

```
int main() {  
    int x = input();  
    int y = input();  
    if (x <= 0)  
        return 0;  
    else if (y <= 0)  
        return 0;
```

Prefix

Already executed

→

```
    else if (x + y == 5)  
        return assert(false);  
    else return 0;  
}
```

Key Idea 1: CPS-Based Concolic Semantics

Capturing Control

What must we save to resume here without repeating the prefix?

```
int main() {  
    int x = input();  
    int y = input();  
    if (x <= 0)  
        return 0;  
    else if (y <= 0)  
        return 0;
```

Prefix

Already executed

State: what values we have

Concrete: $x = 1, y = 1$

Symbolic: $x \mapsto X, y \mapsto Y$

```
→ else if (x + y == 5)  
    return assert(false);  
else return 0;  
}
```

Key Idea 1: CPS-Based Concolic Semantics

Capturing Control

What must we save to resume here without repeating the prefix?

```
int main() {  
    int x = input();  
    int y = input();  
    if (x <= 0)  
        return 0;  
    else if (y <= 0)  
        return 0;
```

Prefix

Already executed

```
→ else if (x + y == 5)  
    return assert(false);  
else return 0;  
}
```

State: what values we have

Concrete: $x = 1, y = 1$

Symbolic: $x \mapsto X, y \mapsto Y$

Control: how to resume
execution from here

?

Key Idea 1: CPS-Based Concolic Semantics

Capturing Control

What must we save to resume here without repeating the prefix?

```
int main() {  
    int x = input();  
    int y = input();  
    if (x <= 0)  
        return 0;  
    else if (y <= 0)  
        return 0;
```

Prefix

Already executed

→

```
    else if (x + y == 5)  
        return assert(false);  
    else return 0;  
}
```

State: what values we have

Concrete: $x = 1, y = 1$

Symbolic: $x \mapsto X, y \mapsto Y$

Control: how to resume
execution from here

Continuation

Key Idea 1: CPS-Based Concolic Semantics

Our CPS Definitional Interpreter

```
type Cont = (stack : List[(Val, Sym)], pc : List[Sym]) => Ans
```

Key Idea 1: CPS-Based Concolic Semantics

Our CPS Definitional Interpreter

type *Cont* = (*stack* : List[(*Val*, *Sym*)], *pc* : List[*Sym*]) => Ans

$\mathbb{E}[\text{es} : \text{List}[\text{Inst}]](\text{stack} : \text{List}[(\text{Val}, \text{Sym})], \text{pc} : \text{List}[\text{Sym}], k : \text{Cont}) : \text{Ans}$

Key Idea 1: CPS-Based Concolic Semantics

Our CPS Definitional Interpreter

type **Cont** = (*stack* : List[(**Val**, **Sym**)], *pc* : List[**Sym**]) => Ans

$\mathbb{E}[\text{es} : \text{List}[\text{Inst}]](\text{stack} : \text{List}[(\text{Val}, \text{Sym})], \text{pc} : \text{List}[\text{Sym}], k : \text{Cont}) : \text{Ans}$

$\mathbb{E}[\text{if } \text{es}_1 \text{ es}_2 :: \text{rest}]((v, s) :: \text{stack}_0, \text{pc}_0, k_0) =$

Key Idea 1: CPS-Based Concolic Semantics

Our CPS Definitional Interpreter

type **Cont** = (*stack* : List[(**Val**, **Sym**)], *pc* : List[**Sym**]) => Ans

$\mathbb{E}[\text{es} : \text{List}[\text{Inst}]](\text{stack} : \text{List}[(\text{Val}, \text{Sym})], \text{pc} : \text{List}[\text{Sym}], k : \text{Cont}) : \text{Ans}$

$\mathbb{E}[\text{if } \text{es}_1 \text{ es}_2 :: \text{rest}]]((v, s) :: \text{stack}_0, \text{pc}_0, k_0) =$

if ($v \neq 0$)

val $\text{pc}_b = s :: \text{pc}_0$

Key Idea 1: CPS-Based Concolic Semantics

Our CPS Definitional Interpreter

type Cont = (stack : List[(Val, Sym)], pc : List[Sym]) => Ans

$\mathbb{E}[\text{es} : \text{List}[\text{Inst}]](\text{stack} : \text{List}[(\text{Val}, \text{Sym})], \text{pc} : \text{List}[\text{Sym}], k : \text{Cont}) : \text{Ans}$

$\mathbb{E}[\text{if } \text{es}_1 \text{ es}_2 :: \text{rest}]]((v, s) :: \text{stack}_0, \text{pc}_0, k_0) =$

if (v != 0)

val pc_b = s :: pc₀

val k₁ = (stack₁, pc₁) => $\mathbb{E}[\text{rest}]](\text{stack}_0, \text{pc}_1, k_0)$

Key Idea 1: CPS-Based Concolic Semantics

Our CPS Definitional Interpreter

type **Cont** = (*stack* : List[(**Val**, **Sym**)], *pc* : List[**Sym**]) => Ans

$\mathbb{E}[\text{es} : \text{List}[\text{Inst}]](\text{stack} : \text{List}[(\text{Val}, \text{Sym})], \text{pc} : \text{List}[\text{Sym}], k : \text{Cont}) : \text{Ans}$

$\mathbb{E}[\text{if } \text{es}_1 \text{ es}_2 :: \text{rest}]]((v, s) :: \text{stack}_0, \text{pc}_0, k_0) =$

if ($v \neq 0$)

val $\text{pc}_b = s :: \text{pc}_0$

val $k_1 = (\text{stack}_1, \text{pc}_1) \Rightarrow \mathbb{E}[\text{rest}]](\text{stack}_0, \text{pc}_1, k_0)$

$\mathbb{E}[\text{es}_1]]([], \text{pc}_b, k_1)$

Key Idea 1: CPS-Based Concolic Semantics

Our CPS Definitional Interpreter

type **Cont** = (*stack* : List[(**Val**, **Sym**)], *pc* : List[**Sym**]) => Ans

$\mathbb{E}[\![es : List[Inst]]\!](stack : List[(\text{Val}, \text{Sym})], pc : List[\text{Sym}], k : \text{Cont}) : \text{Ans}$

$\mathbb{E}[\![\text{if } es_1 \text{ } es_2 :: rest]\!](v, s) :: stack_0, pc_0, k_0) =$

if (*v* != 0)

val *pc_b* = *s* :: *pc₀*

val *k₁* = (*stack₁*, *pc₁*) => $\mathbb{E}[\![rest]\!](stack_0, pc_1, k_0)$

$\mathbb{E}[\![es_1]\!]([], pc_b, k_1)$

else ...

Key Idea 1: CPS-Based Concolic Semantics

Our CPS Definitional Interpreter

type `Cont` = (`stack` : List[(`Val`, `Sym`)], `pc` : List[`Sym`]) => `Ans`

$\mathbb{E}[\text{es} : \text{List}[\text{Inst}]](\text{stack} : \text{List}[(\text{Val}, \text{Sym})], \text{pc} : \text{List}[\text{Sym}], k : \text{Cont}) : \text{Ans}$

$\mathbb{E}[\text{if } \text{es}_1 \text{ es}_2 :: \text{rest}]]((v, s) :: \text{stack}_0, \text{pc}_0, k_0) =$

if (`v` != 0)

val `pcb` = `s` :: `pc0`

val `k1` = (`stack1`, `pc1`) => $\mathbb{E}[\text{rest}]](\text{stack}_0, \text{pc}_1, k_0)$

$\mathbb{E}[\text{es}_1]]([], \text{pc}_b, k_1)$

else ...

Key Idea 2: Compiling via Staging

Generate Code via Staging

Unstaged power

```
def power(x: Int, n: Int): Int =  
  if (n == 0) 1  
  else x * power(x, n - 1)
```

Staged power

```
def genPower(x: Expr[Int], n: Int): Expr[Int] =  
  if (n == 0) '{ 1 }  
  else '{ $x * ${ genPower(x, n - 1) } }
```

Key Idea 2: Compiling via Staging

Generate Code via Staging

Unstaged power

```
def power(x: Int, n: Int): Int =  
  if (n == 0) 1  
  else x * power(x, n - 1)
```

Staged power

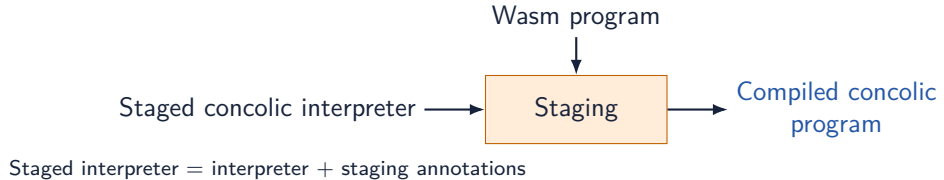
```
def genPower(x: Expr[Int], n: Int): Expr[Int] =  
  if (n == 0) '{ 1 }  
  else '{ $x * ${ genPower(x, n - 1) } }
```

Run

```
val power3 = run('{ (x: Int) => ${ genPower('x, 3) } }')  
             = run('{ (x: Int) => x * x * x }')
```

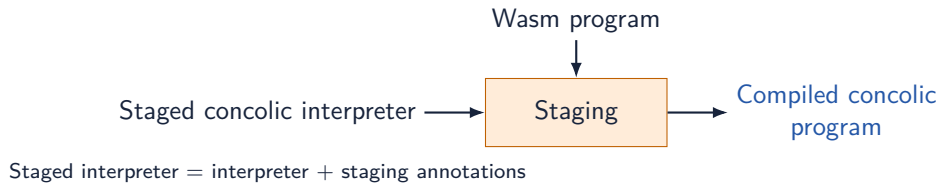
Key Idea 2: Compiling via Staging

First Futamura Projection



Key Idea 2: Compiling via Staging

First Futamura Projection



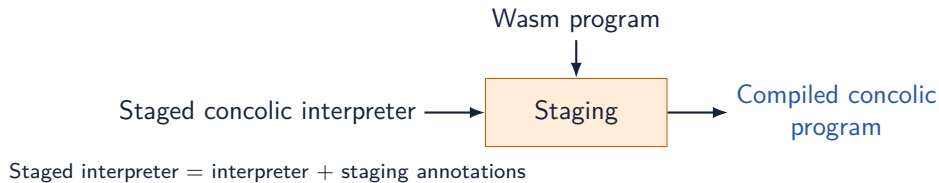
Staged semantics

$\text{Expr}[T]$: code producing a T

type $\text{Cont} = (\text{stack} : \text{List}[(\text{Val}, \text{Sym})], \text{pc} : \text{List}[\text{Sym}]) \Rightarrow \text{Ans}$

Key Idea 2: Compiling via Staging

First Futamura Projection



Staged semantics

$\text{Expr}[T]$: code producing a T

type $\text{Cont} = (\text{stack} : \text{List}[(\text{Val}, \text{Sym})], \text{pc} : \text{List}[\text{Sym}]) \Rightarrow \text{Ans}$

$\mathbb{S}[\text{es} : \text{List}[\text{Inst}]](\text{stack} : \text{Expr}[\text{List}[(\text{Val}, \text{Sym})]], \text{pc} : \text{Expr}[\text{List}[\text{Sym}]], k : \text{Expr}[\text{Cont}]) : \text{Expr}[\text{Ans}]$

Derived from concolic semantics by adding staging annotations.

Key Idea 2: Compiling via Staging

Staged Concolic Semantics

type Cont = (stack : List[(Val, Sym)], pc : List[Sym]) => Ans

Expr[T]: code producing a T

$\mathbb{S}[\text{es} : \text{List}[\text{Inst}]](\text{stack} : \text{Expr}[\text{List}[(\text{Val}, \text{Sym})]], \text{pc} : \text{Expr}[\text{List}[\text{Sym}]], k : \text{Expr}[\text{Cont}]) : \text{Expr}[\text{Ans}]$

Current-stage values Expr types

```
 $\mathbb{S}[\text{if } \text{es}_1 \text{ es}_2 :: \text{rest}]](\text{stack}_0, \text{pc}_0, k_0) = \text{'}\{$   
  val (v, s) = $stack0.head  
  if (v != 0)  
    val pcb = s :: $pc0  
    val k1 : Cont = (stack1, pc1) => ${  $\mathbb{S}[\text{rest}]](\text{'}\{ \$\text{stack}_0.\text{tail} \}, \text{'pc}_1, k_0) \}$   
    ${  $\mathbb{S}[\text{es}_1]](\text{'}\{ [] \}, \text{'pc}_b, \text{'k}_1) \}$   
  else ...  
}
```

Key Idea 2: Compiling via Staging

Staged Concolic Semantics

```
type Cont = (stack : List[(Val, Sym)], pc : List[Sym]) => Ans
```

$\text{Expr}[T]$: code producing a T

$$\mathbb{S}[\text{es} : \text{List}[\text{Inst}]](stack : \text{Expr}[\text{List}[(\text{Val}, \text{Sym})]], pc : \text{Expr}[\text{List}[\text{Sym}]], k : \text{Expr}[\text{Cont}]) : \text{Expr}[\text{Ans}]$$

Current-stage values Expr types

```

S[[if es1 es2 :: rest]](stack0, pc0, k0) = '{
  val (v, s) = $stack0.head
  if (v != 0)
    val pcb = s :: $pc0
    val k1 : Cont = (stack1, pc1) => ${ S[[rest]]('{$stack0.tail }, 'pc1, k0) }
    ${ S[[es1]]('{} }, 'pcb, 'k1) }
  else ...
}
```

Key Idea 2: Compiling via Staging

Staged Concolic Semantics

type Cont = (stack : List[(Val, Sym)], pc : List[Sym]) => Ans

Expr[T]: code producing a T

$\mathbb{S}[\text{es} : \text{List}[\text{Inst}]](\text{stack} : \text{Expr}[\text{List}[(\text{Val}, \text{Sym})]], \text{pc} : \text{Expr}[\text{List}[\text{Sym}]], \boxed{k} : \boxed{\text{Expr}[\text{Cont}]}) : \text{Expr}[\text{Ans}]$

Current-stage values Expr types

$\mathbb{S}[\text{if } \text{es}_1 \text{ es}_2 :: \text{rest}]](\text{stack}_0, \text{pc}_0, \boxed{k_0}) = \text{'}\{$

val (v, s) = $\text{\$stack}_0.\text{head}$

if (v != 0)

val $\text{pc}_b = s :: \text{\$pc}_0$

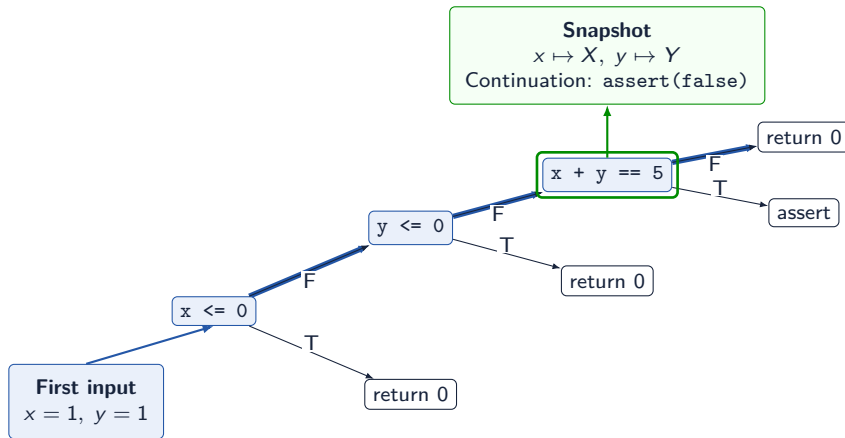
val $\boxed{k_1} : \text{Cont} = (\text{stack}_1, \text{pc}_1) \Rightarrow \text{\$}\{\mathbb{S}[\text{rest}]](\text{'}\{\text{\$stack}_0.\text{tail}\}, \text{'pc}_1, \boxed{k_0})\}$

$\text{\$}\{\mathbb{S}[\text{es}_1]](\text{'}\{\text{\$}\{\text{\$}\}\}, \text{'pc}_b, \boxed{k_1})\}$

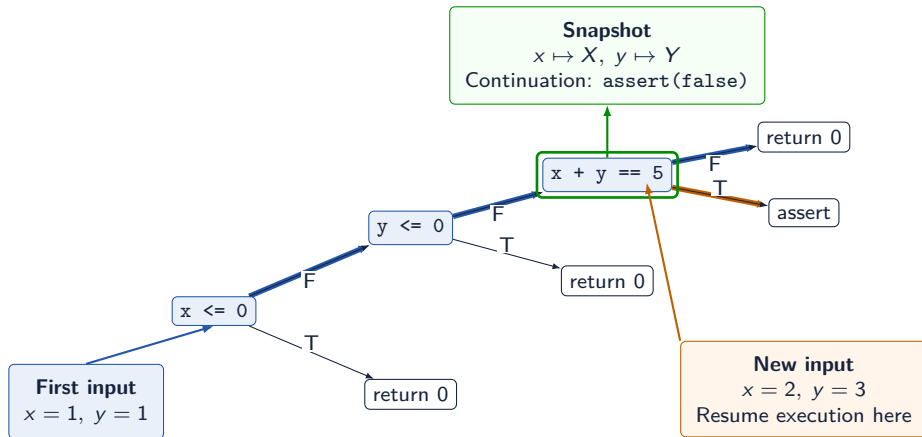
else ...

$\text{'}\}$

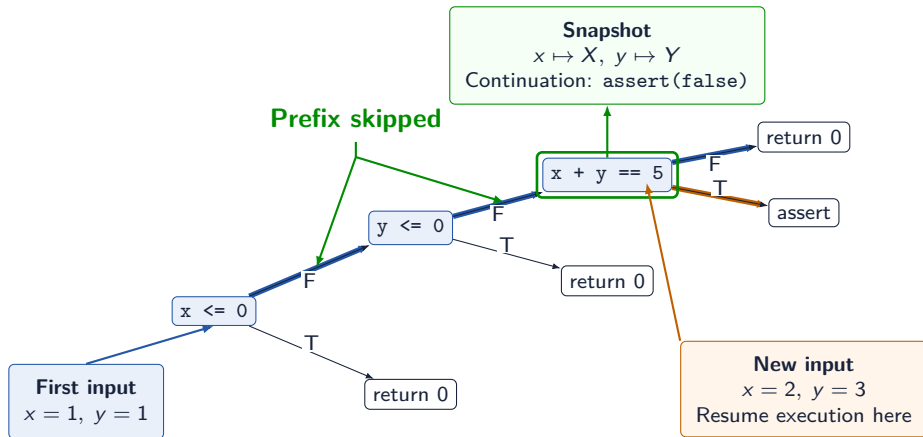
Key Idea 3: Heuristic Snapshot Reuse



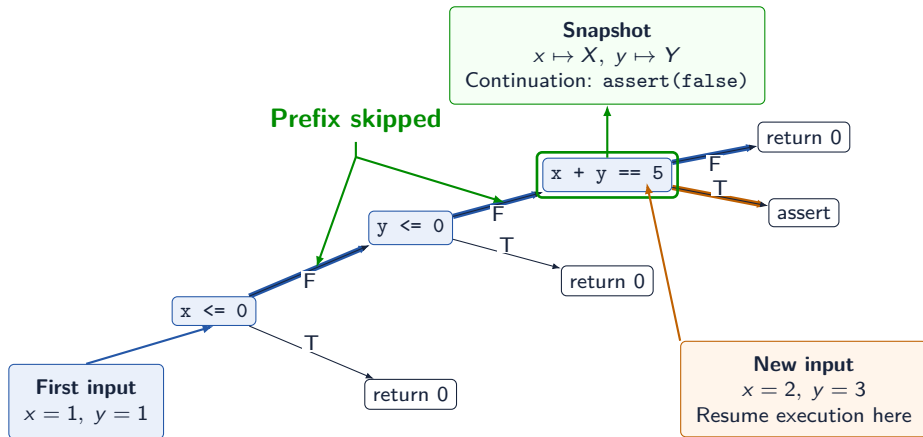
Key Idea 3: Heuristic Snapshot Reuse



Key Idea 3: Heuristic Snapshot Reuse

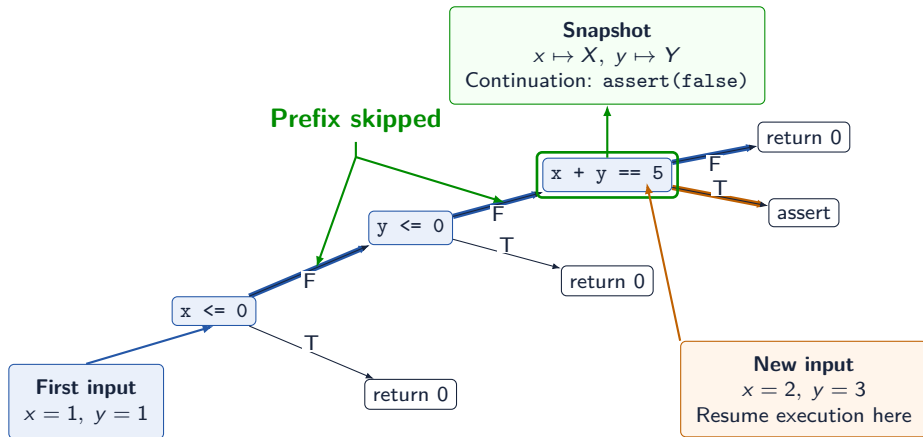


Key Idea 3: Heuristic Snapshot Reuse



**With a complex symbolic state,
restoring execution can be slower than restarting.**

Key Idea 3: Heuristic Snapshot Reuse



Reuse a snapshot when

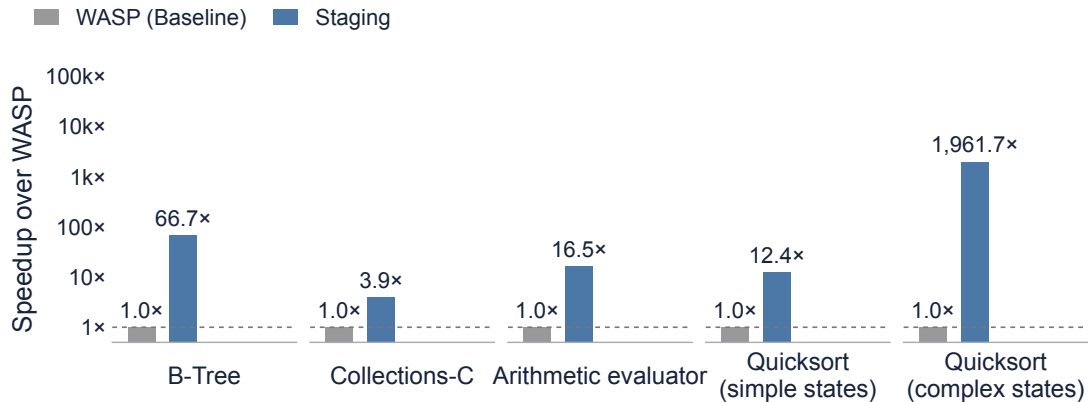
Prefix re-execution time



Estimated snapshot resumption time

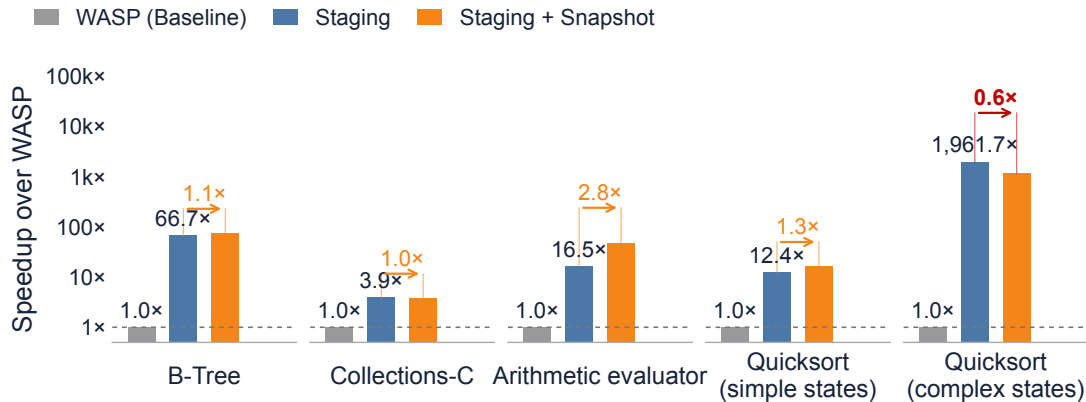
Speedup Across Benchmark Groups

Average speedup (geometric mean)



Speedup Across Benchmark Groups

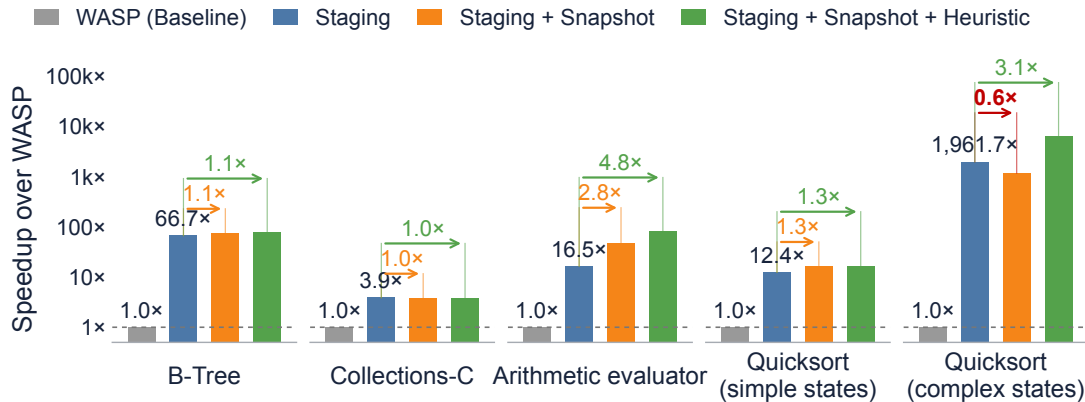
Average speedup (geometric mean)



Arrows: gain over staging $< 1\times$ = **slowdown**

Speedup Across Benchmark Groups

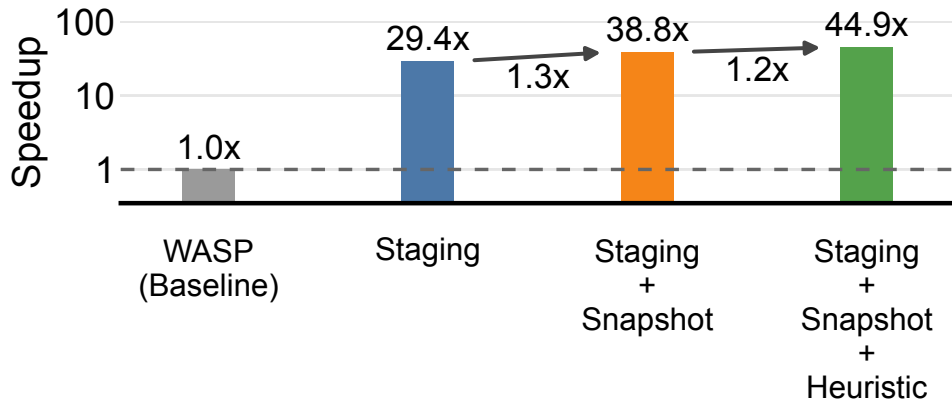
Average speedup (geometric mean)



Arrows: gain over staging $< 1\times$ = slowdown

Evaluation

184 test cases across 5 benchmark groups



Takeaways

In a nutshell:

Compiling with Continuations
Efficiency Snapshot reuse
via Staging for WebAssembly
Simplicity

Key ideas

1. CPS-based concolic semantics → Capturable control flow & snapshot reuse
2. Compiling via staging → Efficiency
3. Heuristic snapshot reuse → Avoid costly snapshots

Thank you! Questions?

<https://github.com/Generative-Program-Analysis/GenSym/>

