Compiling Symbolic Execution with Staging and Algebraic Effects

Guannan Wei, Oliver Bračevac, Shangyin Tan, Tiark Rompf

Department of Computer Science, Purdue University

SPLASH/OOPSLA 2020

PURDUE UNIVERSITY

Symbolic Execution

Translating programs to symbolic expressions and path conditions.

```
x = symbolic_input()
y = symbolic_input()
if (x ⩽ y) { z := x }
else {
  if (y > 1) { z := y }
  else { z := x + y }
}
```

```
path 1: z = x, { x ⩽ y }
path 2: z = y, { x > y ∧ y > 1 }
path 3: z = x+y, { x > y ∧ y ⩽ 1 }
```

Symbolic Execution

Translating programs to symbolic expressions and path conditions.

```
x = symbolic_input()
y = symbolic_input()
if (x ⩽ y) { z := x }
else {
  if (y > 1) { z := y }
  else { z := x + y }
}
```

```
path 1: z = x, { x ⩽ y }
path 2: z = y, { x > y ∧ y > 1 }
path 3: z = x+y, { x > y ∧ y ⩽ 1 }
```

Symbolic Execution

Translating programs to symbolic expressions and path conditions.

```
x = symbolic_input()
y = symbolic_input()
if (x ⩽ y) { z := x }
else {
  if (y > 1) { z := y }
  else { z := x + y }
}
```

```
path 1: z = x, { x ⩽ y }
path 2: z = y, { x > y ∧ y > 1 }
path 3: z = x+y, { x > y ∧ y ⩽ 1 }
```

Manticore

KLEE

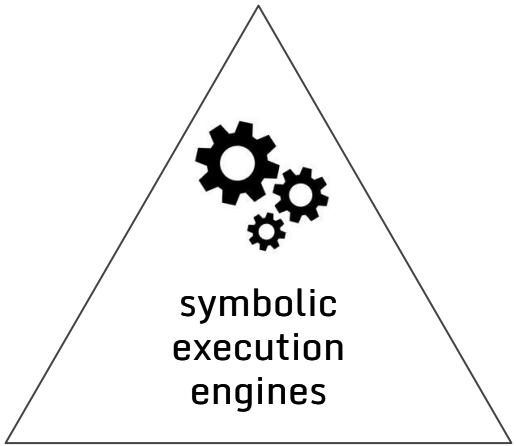
Angr

Symbolic JPF

S2E, Dart, Rosette, CUTE, PyExZ3, EXE, SymCC, Chef, and many others...

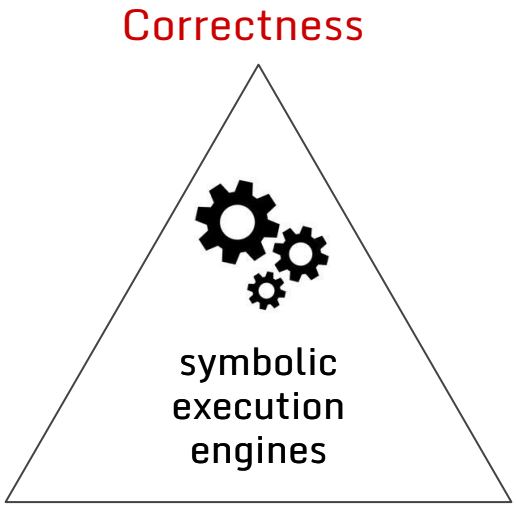
“Building a symbolic compiler is often the most difficult aspect of creating solver-aided tools, especially for general-purpose languages.”

-- Torlak and Bodik [2014]



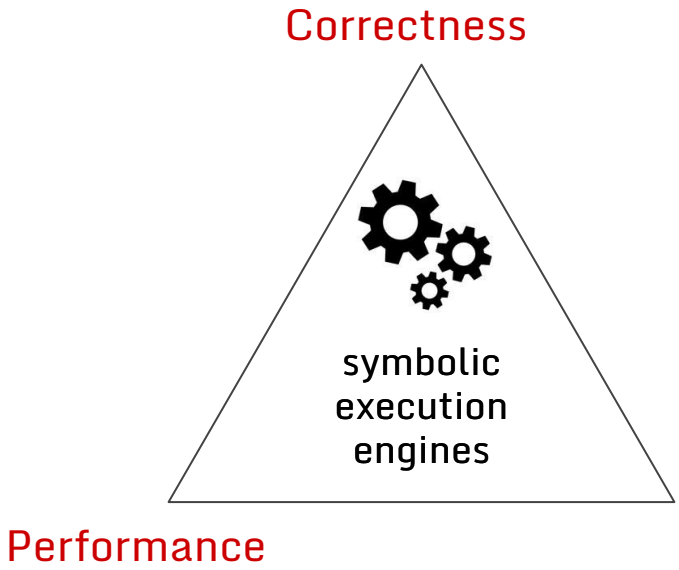
“Building a symbolic compiler is often the most difficult aspect of creating solver-aided tools, especially for general-purpose languages.”

-- Torlak and Bodik [2014]



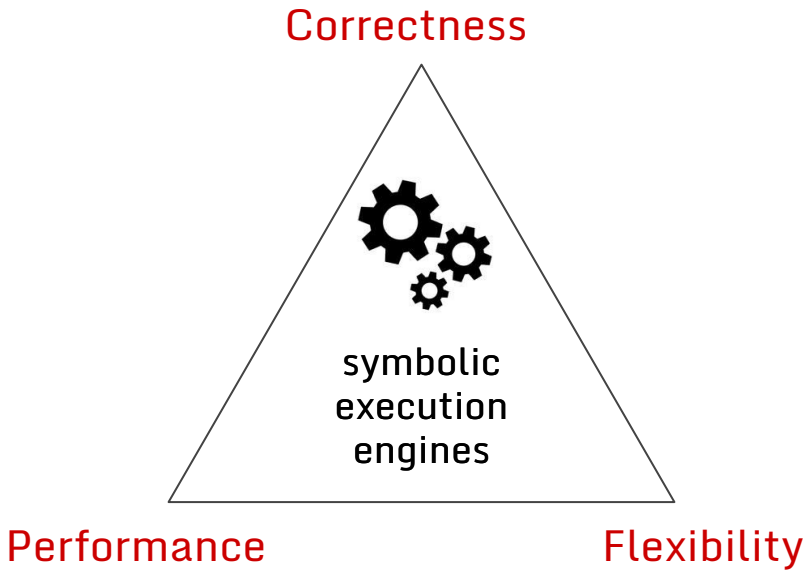
“Building a symbolic compiler is often the most difficult aspect of creating solver-aided tools, especially for general-purpose languages.”

-- Torlak and Bodik [2014]



“Building a symbolic compiler is often the most difficult aspect of creating solver-aided tools, especially for general-purpose languages.”

-- Torlak and Bodik [2014]

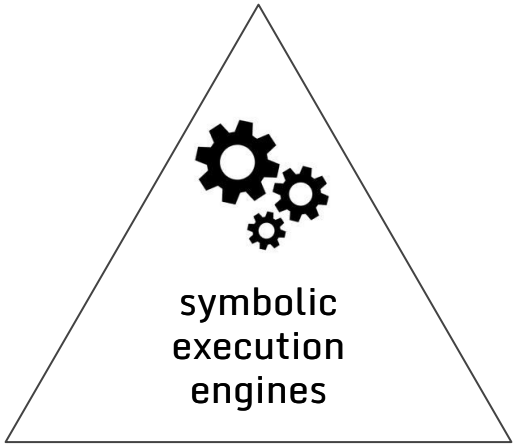


“Building a symbolic compiler is often the most difficult aspect of creating solver-aided tools, especially for general-purpose languages.”

-- Torlak and Bodik [2014]

Definitional interpreters

Correctness



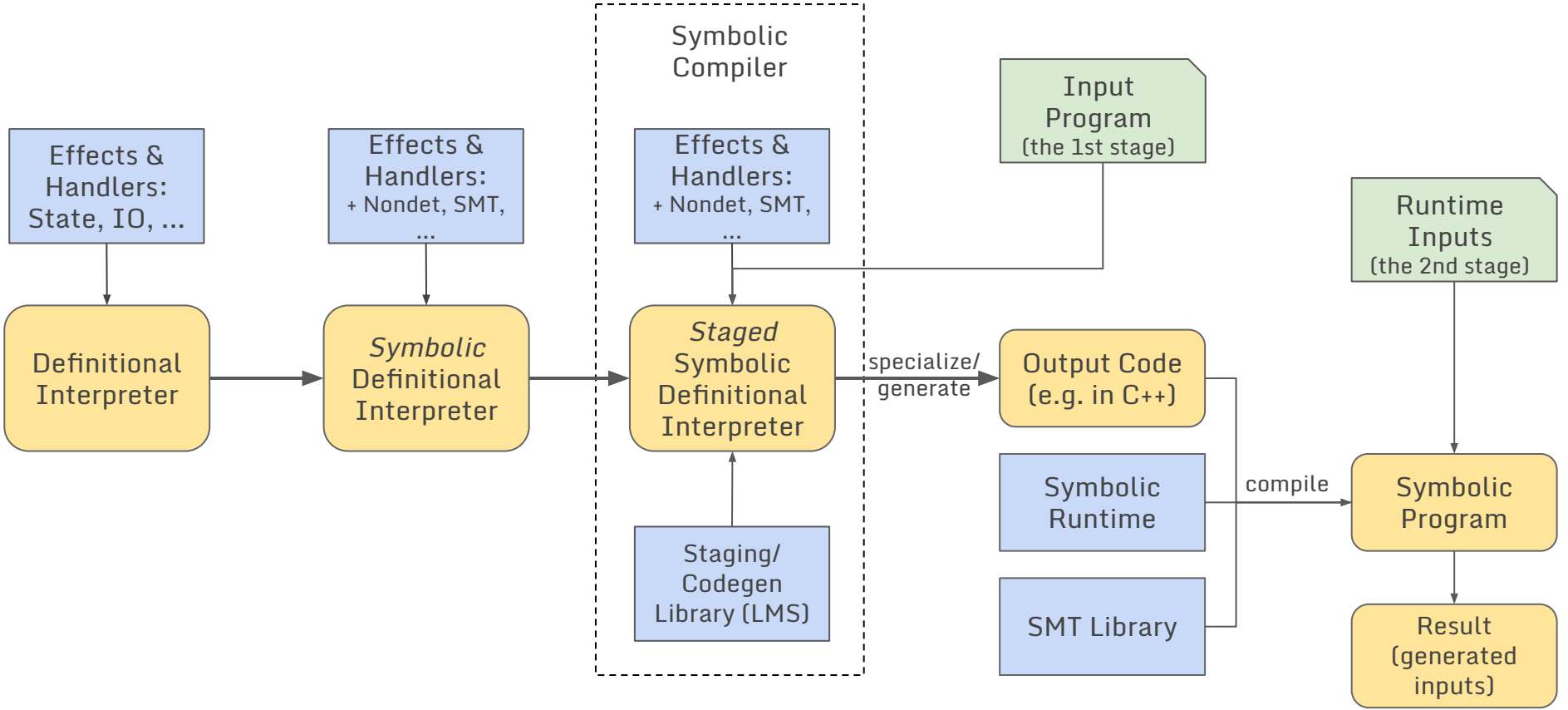
Performance

Multi-stage programming

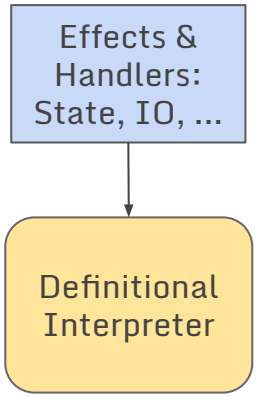
Flexibility

Algebraic effects and handlers

Our Approach – Overview



Our Approach – Definitional Interpreters



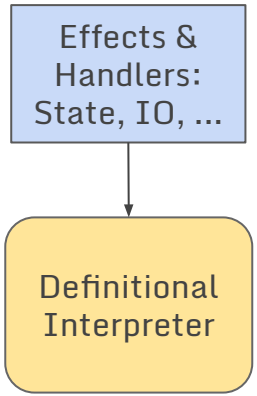
- Consider a simple imperative language Imp/While:

$$v \in \text{Value} ::= \mathbb{Z} \mid \mathbb{B} \qquad op_2 \in \{+, -, \times, =, \neq, \dots\} \qquad op_1 \in \{-, \dots\}$$
$$e \in \text{Expr} ::= v \mid x \mid op_1(e) \mid op_2(e, e) \mid \text{read} \qquad x \in \text{Var}$$
$$s \in \text{Stmt} ::= \text{assign}(x, e) \mid \text{cond}(e, s, s) \mid \text{while}(e, s) \mid \text{seq}(s, s) \mid \text{skip} \mid \text{write}(e)$$

- Define a high-level definitional interpreter with algebraic effects (the State and IO effect):

$$\text{exec} : \text{Stmt} \to \langle \text{State}[\text{Store}], \text{IO} \rangle \text{Unit}$$
$$\text{exec}(\text{assign}(x, e)) = \{\ v \leftarrow \text{eval}(e);\quad \sigma \leftarrow \text{get};\ \textbf{return}\ \text{put}(\sigma[x \mapsto v])\ \}$$
$$\text{exec}(\text{cond}(e, s_1, s_2)) = \{\ b \leftarrow \text{eval}(e);\quad \textbf{return if}\ (b)\ \text{exec}(s_1)\ \textbf{else}\ \text{exec}(s_2)\ \}$$
$$\text{exec}(\text{while}(e, s)) = \{\ b \leftarrow \text{eval}(e);\quad \textbf{return if}\ (b)\ \text{exec}(\text{seq}(s, \text{while}(e, s)))\ \textbf{else}\ \text{exec}(\text{skip})\ \}$$
$$\text{exec}(\text{write}(e)) = \{\ v \leftarrow \text{eval}(e);\quad \textbf{return}\ \text{writeInt}(v)\ \}$$
$$\text{exec}(\text{seq}(s_1, s_2)) = \{\ \_ \leftarrow \text{exec}(s_1);\ \textbf{return}\ \text{exec}(s_2)\ \}$$

Our Approach – Definitional Interpreters



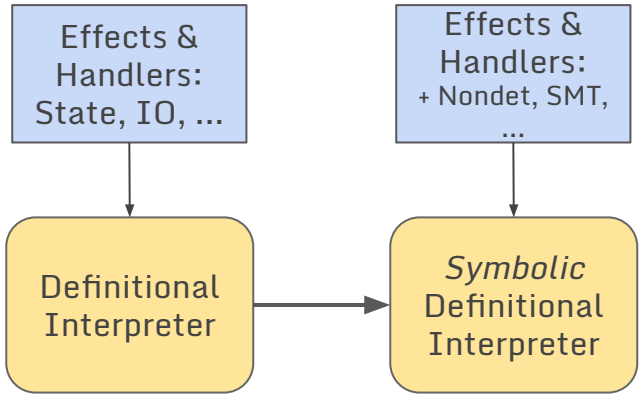
- Consider a simple imperative language Imp/While:

$$v \in \text{Value} ::= \mathbb{Z} \mid \mathbb{B} \qquad op_2 \in \{+, -, \times, =, \neq, \dots\} \qquad op_1 \in \{-, \dots\}$$
$$e \in \text{Expr} ::= v \mid x \mid op_1(e) \mid op_2(e, e) \mid \text{read} \qquad x \in \text{Var}$$
$$s \in \text{Stmt} ::= \text{assign}(x, e) \mid \text{cond}(e, s, s) \mid \text{while}(e, s) \mid \text{seq}(s, s) \mid \text{skip} \mid \text{write}(e)$$

- Define a high-level definitional interpreter with algebraic effects (the State and IO effect):

$$\text{exec} : \text{Stmt} \to \langle \text{State}[\text{Store}], \text{IO} \rangle \text{Unit}$$
$$\text{exec}(\text{assign}(x, e)) = \{\ v \leftarrow \text{eval}(e);\ \ \sigma \leftarrow \text{get};\ \ \textbf{return}\ \text{put}(\sigma[x \mapsto v])\ \}$$
$$\text{exec}(\text{cond}(e, s_1, s_2)) = \{\ b \leftarrow \text{eval}(e);\ \ \textbf{return if}\ (b)\ \text{exec}(s_1)\ \textbf{else}\ \text{exec}(s_2)\ \}$$
$$\text{exec}(\text{while}(e, s)) = \{\ b \leftarrow \text{eval}(e);\ \ \textbf{return if}\ (b)\ \text{exec}(\text{seq}(s, \text{while}(e, s)))\ \textbf{else}\ \text{exec}(\text{skip})\ \}$$
$$\text{exec}(\text{write}(e)) = \{\ v \leftarrow \text{eval}(e);\ \ \textbf{return}\ \text{writeInt}(v)\ \}$$
$$\text{exec}(\text{seq}(s_1, s_2)) = \{\ \_ \leftarrow \text{exec}(s_1);\ \textbf{return}\ \text{exec}(s_2)\ \}$$

Our Approach – Symbolic Definitional Interpreter



- Refactor to a *symbolic* definitional interpreter:
  - Symbolic expressions can be values
  - Collecting path conditions via the State effect

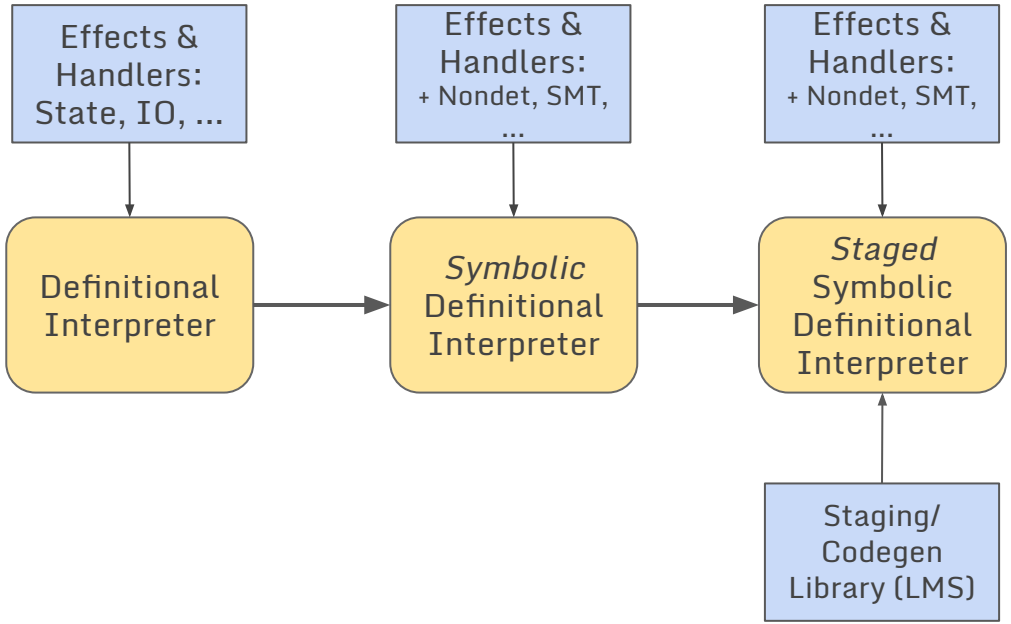
$$\mathbb{S} = \text{Store} \times \text{PC}$$

  - Introducing a nondeterminism Nondet effect
  - Modeling solver invocations with an SMT effect

$$\text{exec} : \text{Stmt} \to \langle \text{State}[\mathbb{S}], \text{IO}^{\#}, \text{SMT}, \text{Nondet} \rangle \text{Unit}$$

$$\begin{aligned}
&\text{exec}(\text{cond}(e, s_1, s_2)) = \\
&\quad \{\ b \leftarrow \text{eval}(e); \\
&\qquad \textbf{return if } (b \equiv \text{true})\ \text{exec}'(s_1, e) \\
&\qquad\qquad \textbf{else if } (b \equiv \text{false})\ \text{exec}'(s_2, \neg e) \\
&\qquad\qquad \textbf{else } (s', e') \leftarrow \text{choice}((s_1, e), (s_2, \neg e));\ \text{exec}(s', e')\ \}
\end{aligned}$$

Our Approach – Staged Symbolic Definitional Interpreter

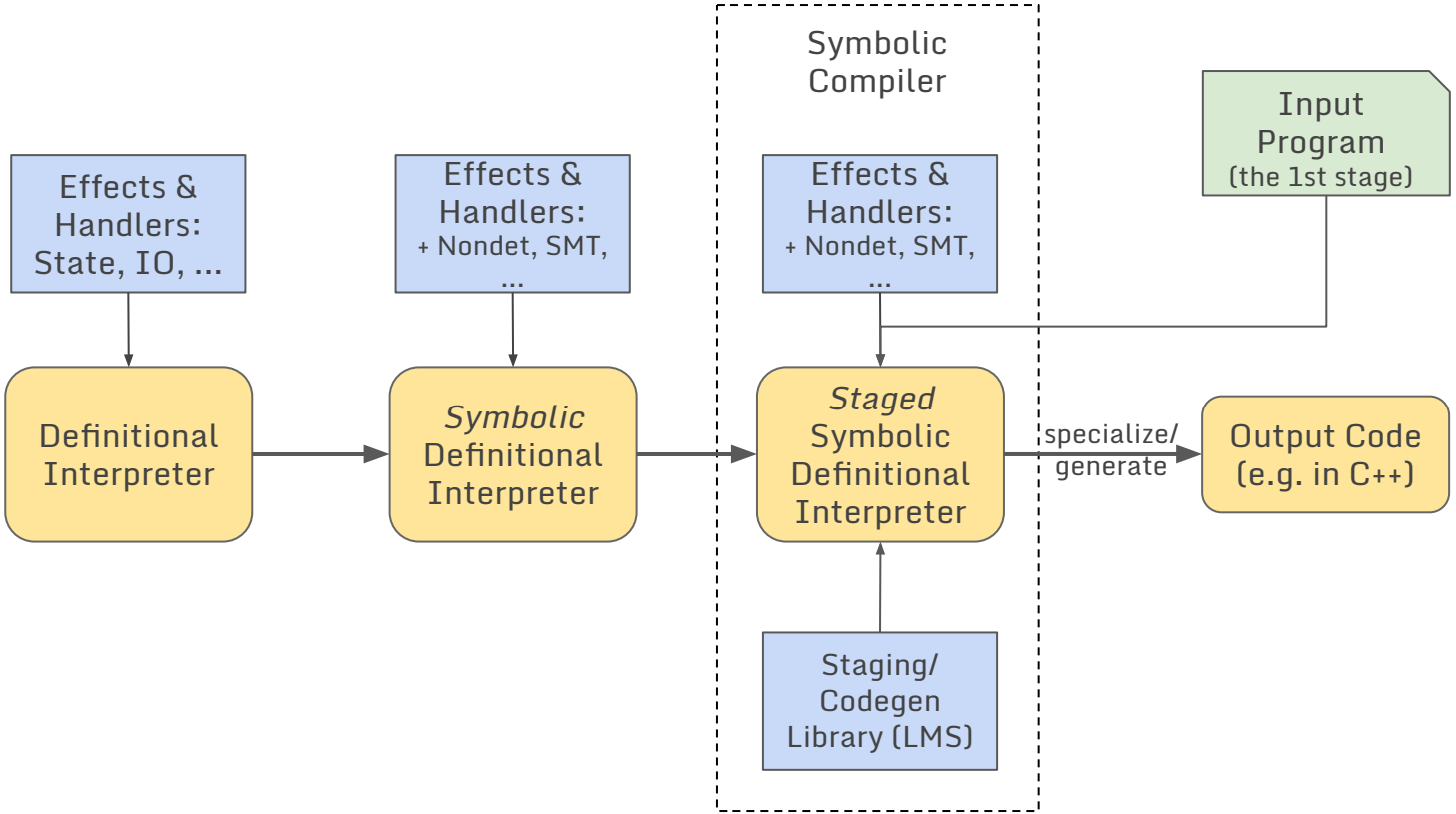


- Perform a manual binding-time analysis:
  - Annotate which parts of the computation happen statically (1st stage) and dynamically (2nd stage)
  - Ensure that effects are resolved strictly in the 1st stage

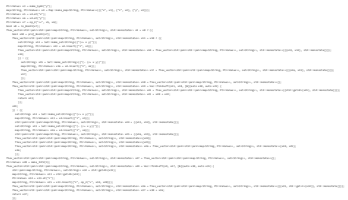
$$\text{exec} : \text{Stmt} \to \langle \text{State}[\underline{\mathbb{S}}], \text{IO}^{\#}, \text{SMT}, \text{Nondet} \rangle \underline{\text{Unit}}$$

$$\begin{aligned} \text{exec}(\text{assign}(x, e)) = \{\ & \underline{v} \leftarrow \text{eval}(e); \\ & \underline{\sigma} \leftarrow \text{getStore}; \\ & \textbf{return}\ \text{putStore}(\underline{\sigma[x \mapsto v]})\ \} \end{aligned}$$

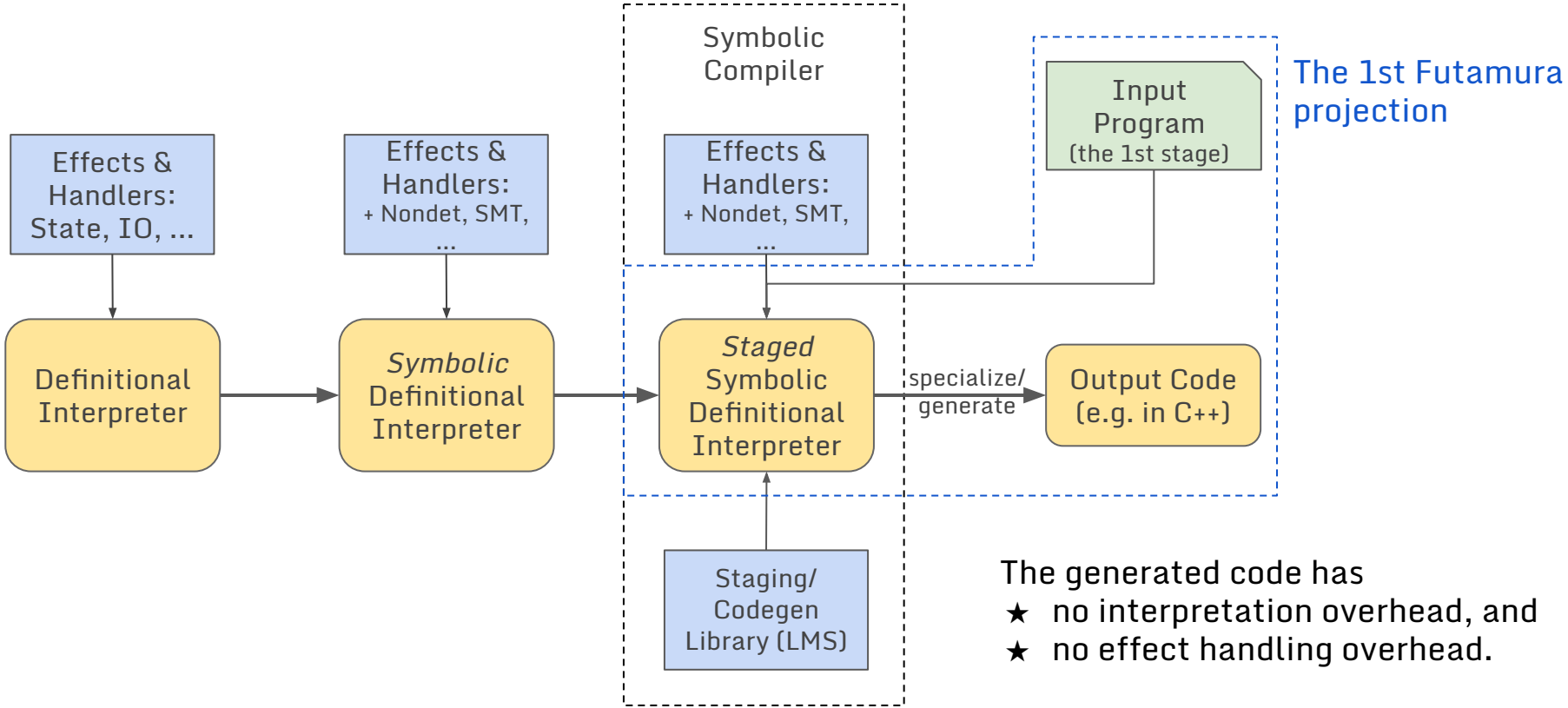
Our Approach – Staged Symbolic Definitional Interpreter




```
if (x ⩽ y#) { z := x }
else { z := y# }
z := z + 1
```



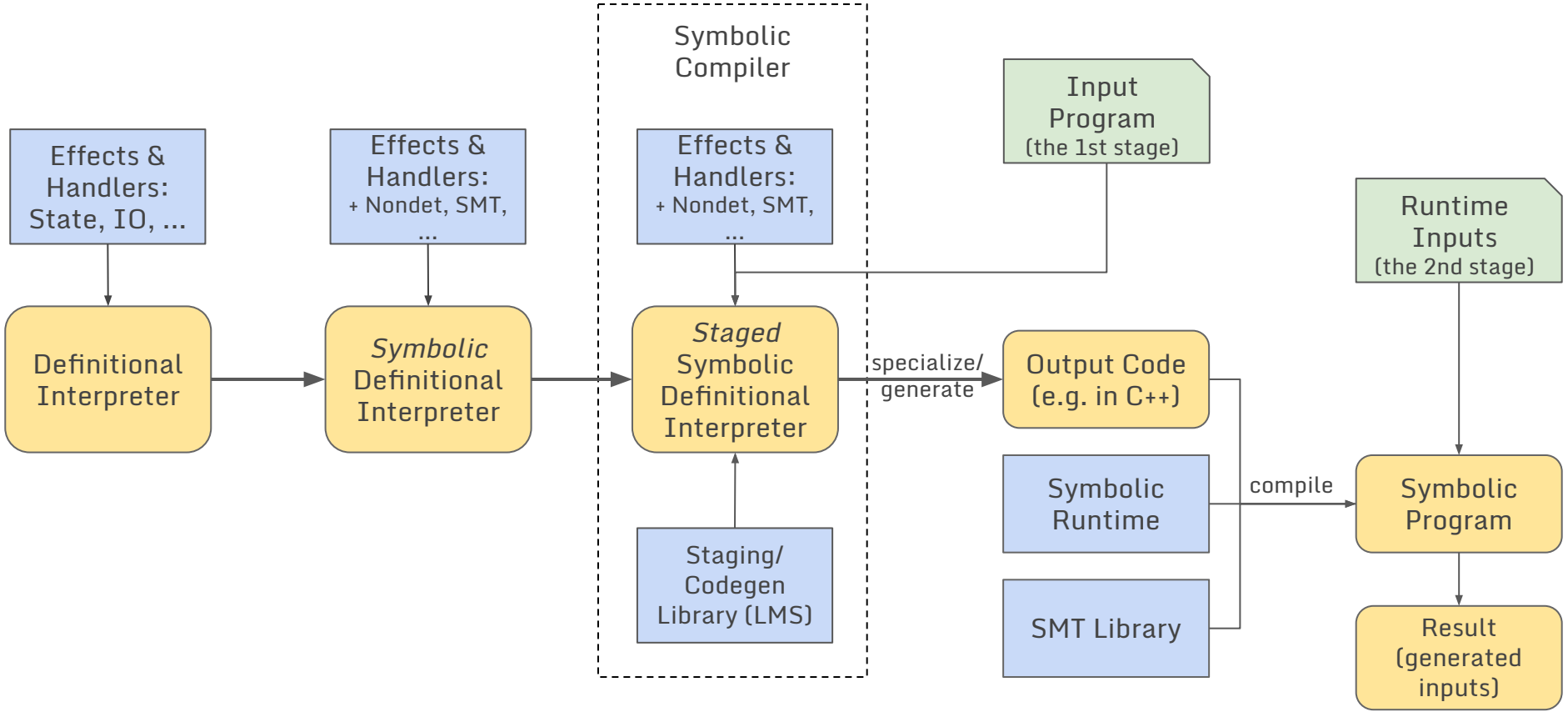
Our Approach – Staged Symbolic Definitional Interpreter



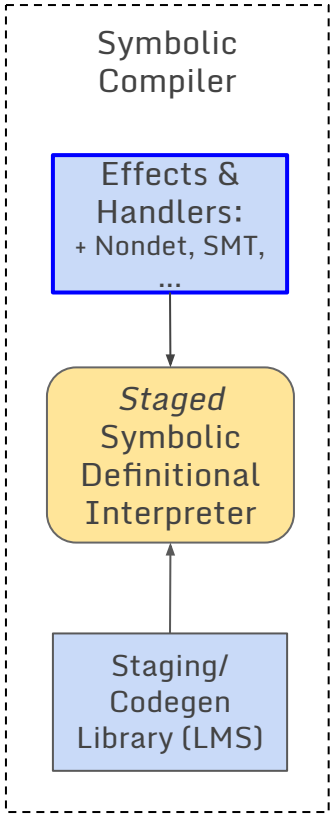
The generated code has

- ★ no interpretation overhead, and
- ★ no effect handling overhead.

Our Approach – Staged Symbolic Definitional Interpreter



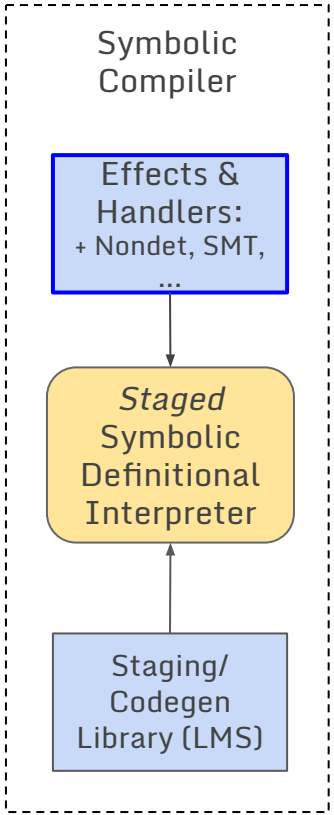
Execution Strategies à la Effect Handlers



- The effect operations used in the interpreter are *abstract*
- Effect handlers provide interpretations of the effect
- Composing effect handlers and exec:

$$(\text{NondetHandler} \circ \text{SMTHandler} \circ \text{IO}^{\#}\text{Handler} \circ \text{StateHandler}(\underline{s_0}) \circ \text{exec})(p) : \underline{\text{List}[\mathbb{S} \times \text{Unit}]}$$

Execution Strategies à la Effect Handlers



- The effect operations used in the interpreter are *abstract*
- Effect handlers provide interpretations of the effect
- Composing effect handlers and exec:

$$(\text{NondetHandler} \circ \text{SMTHandler} \circ \text{IO}^{\#}\text{Handler} \circ \text{StateHandler}(\underline{s_0}) \circ \text{exec})(p) : \underline{\text{List}[\mathbb{S} \times \text{Unit}]}$$

★ Different handlers of the Nondet effect give rise to different path exploration strategies:
  - exhaustive exploration via backtracking (DFS or BFS)
  - random sampling, generating a flip coin into the second stage
  - fair random sampling
  - more to explore...

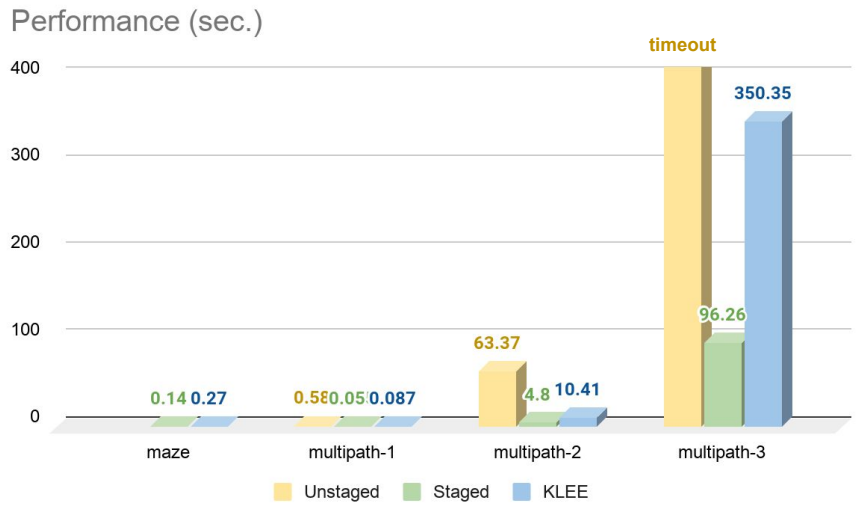
Implementation & Scale-up

- Using the Lightweight Modular Staging framework (LMS) in Scala
- Embedding algebraic effects using freer monads
- Building a prototype for a subset of LLVM IR, supporting ˜20 instructions
- Generating C++ code using immutable data structures (Immer) and STP API
- A simple symbolic runtime also using Immer

Performance

- Path traversal time outperforms over unstaged version and KLEE on microbenchmarks

  10-30x vs unstaged
  2-3x vs KLEE

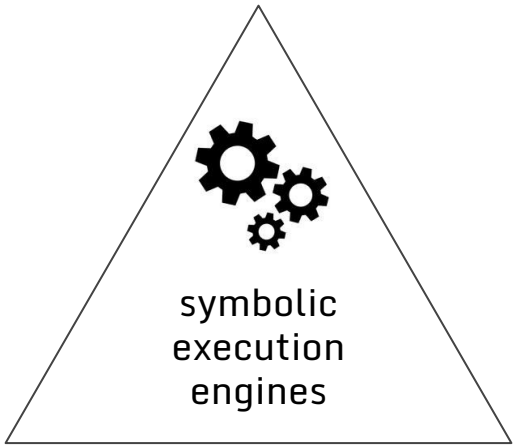


Benchmark	#path / #exec inst	Unstaged SE	Staged SE	KLEE
maze (w. solver)	309 / 65210	-	0.14s	0.27s
multipath-1	2^{10} / 30k	0.58 s	0.055 s	0.087 s
multipath-2	2^{16} / 2.6M	63.37 s	4.80 s	10.41 s
multipath-3	2^{20} / 51M	timeout (1h+)	96.26 s	350.35 s

Staged symbolic definitional interpreter with algebraic effects

Definitional interpreters

Correctness



Performance

Multi-stage programming

Flexibility

Algebraic effects and handlers

Thank you for your attention!

Code available at:

https://github.com/Kraks/sai