# Symbolic Computation via Program Transformation

**Henrich Lauko**, Petr Ročkai and Jiří Barnat

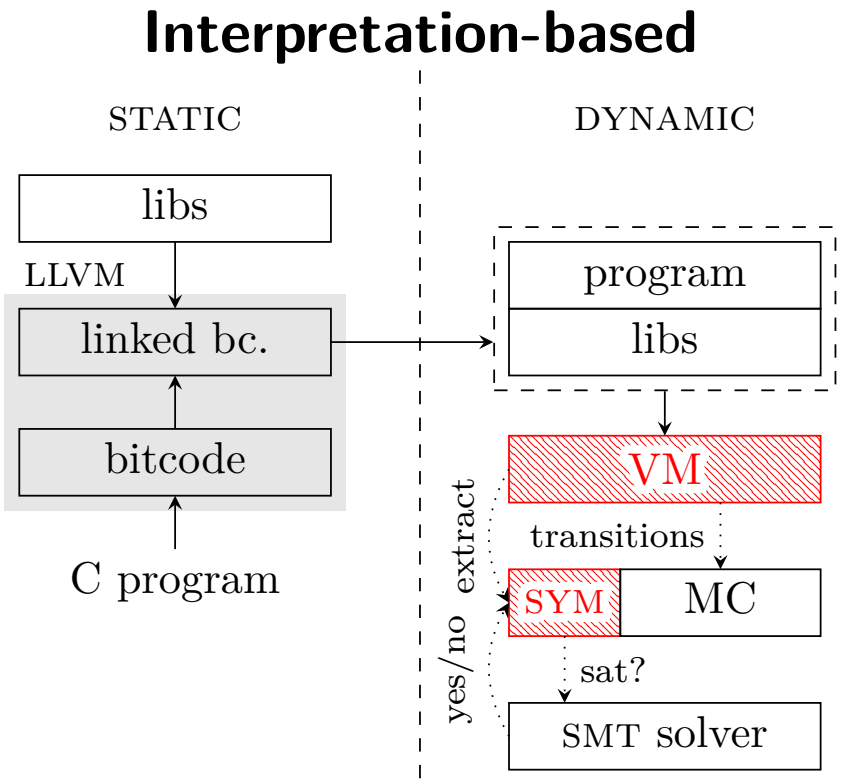

Masaryk University

Brno, Czech Republic

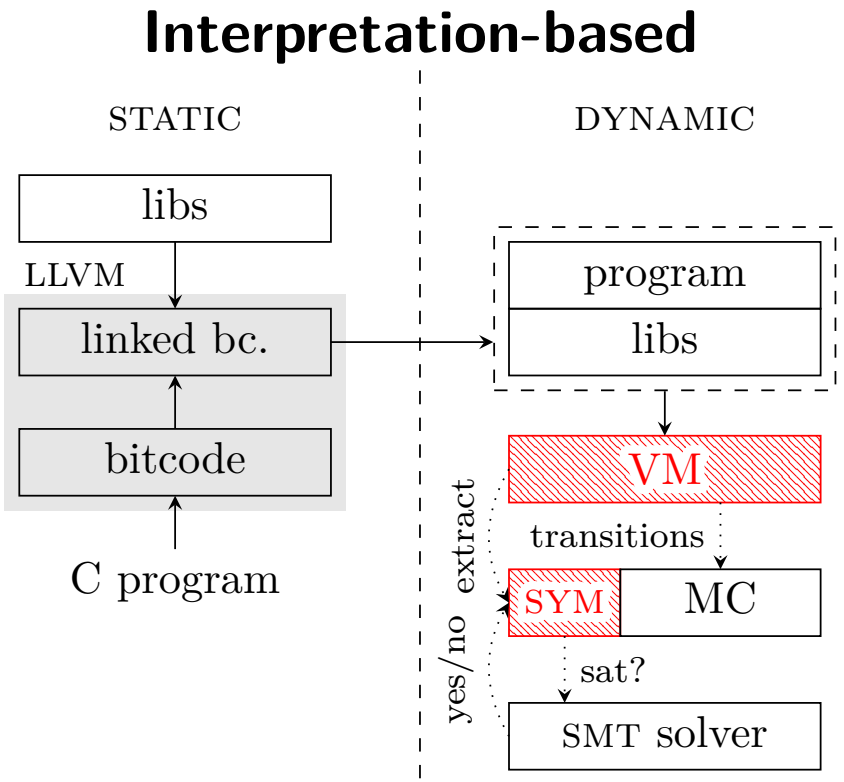
# Symbolic Computation Motivation

- verify programs with inputs from the enviroment
- symbolic execution, concolic testing, test generation etc.

# Symbolic Computation

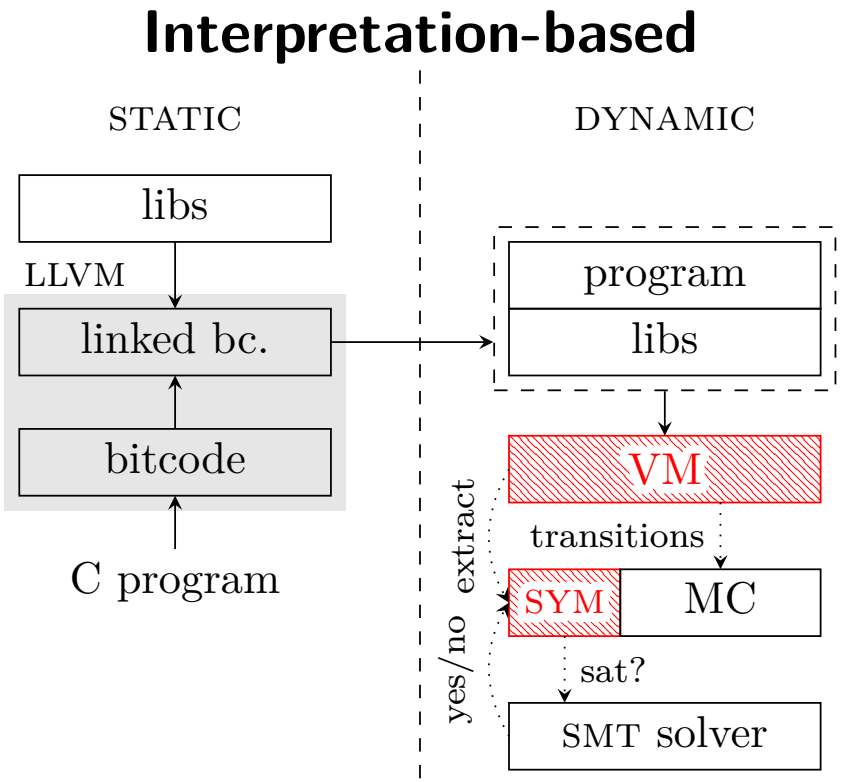


# Symbolic Computation



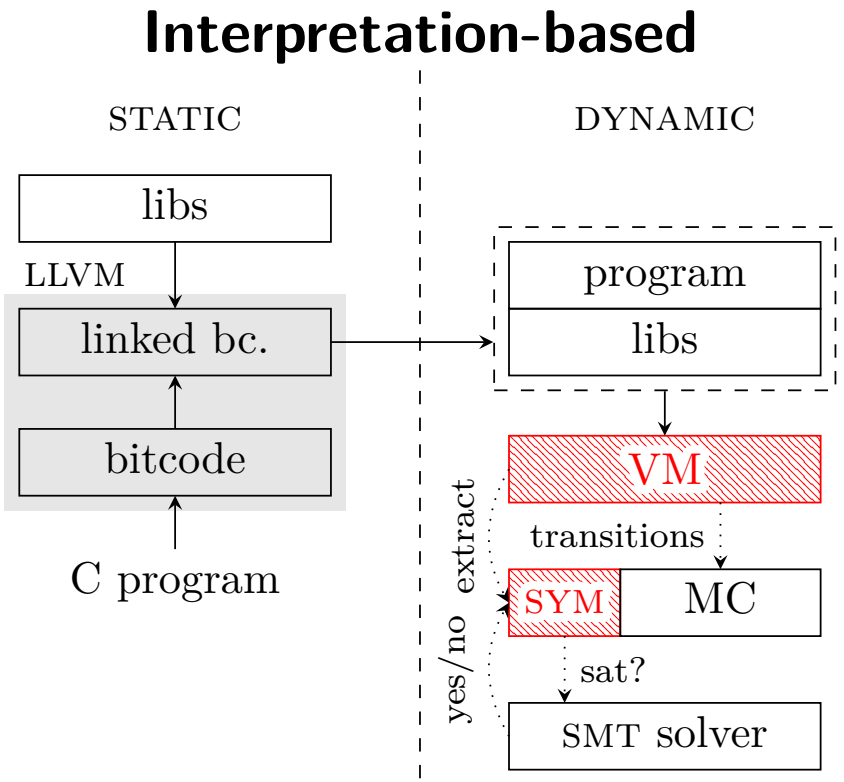

```
1 a ← input()
2 if (a > 0)
3    b ← a + 1
4 else
5    b ← a - 1
```

# Symbolic Computation



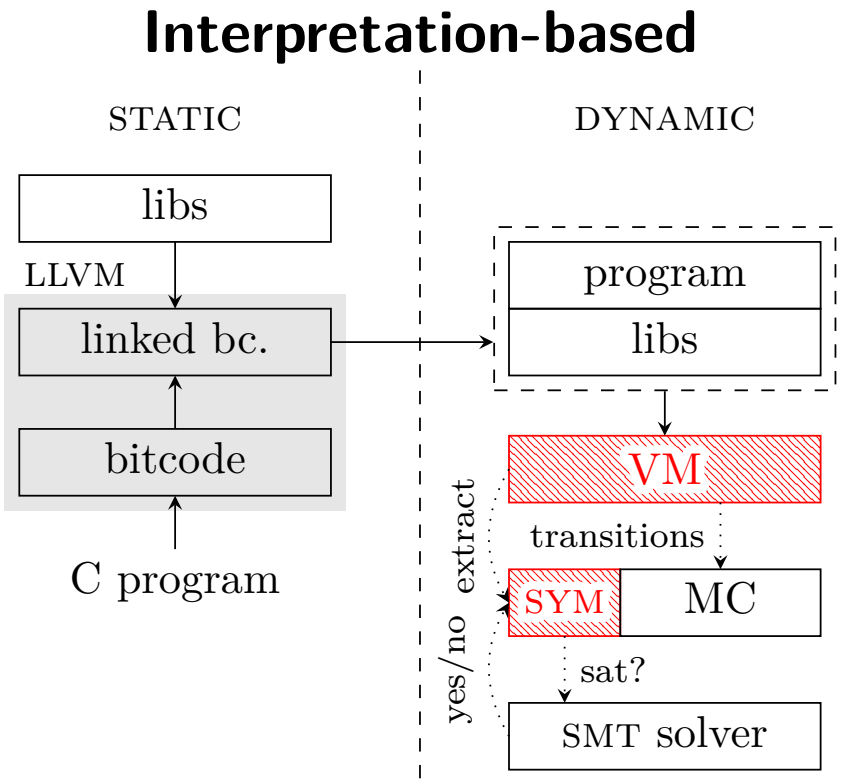

```
1 a ← input()   [true]
2 if (a > 0)
3    b ← a + 1
4 else
5    b ← a - 1
```

# Symbolic Computation



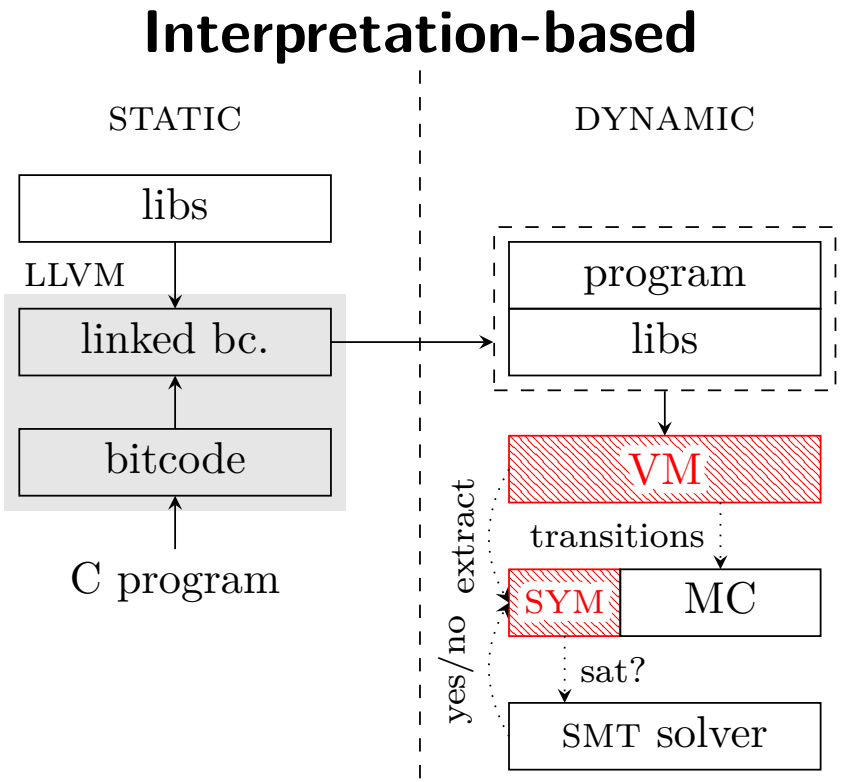

```
1 a ← input()   [true]
2 if (a > 0)    [a > 0]
3   b ← a + 1
4 else
5   b ← a - 1
```

# Symbolic Computation



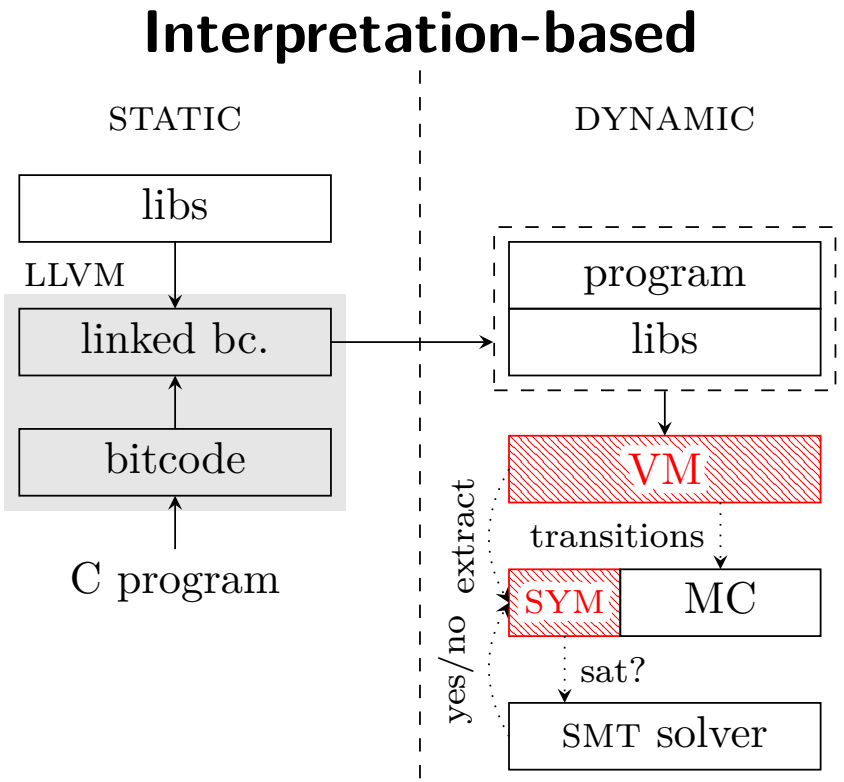

```
1 a ← input()    [true]
2 if (a > 0)     [a > 0]
3    b ← a + 1
        [a > 0 ∧ b = a + 1]
4 else
5    b ← a - 1
```

# Symbolic Computation




```
1 a ← input()    [true]
2 if (a > 0)     [a > 0]
3    b ← a + 1
       [a > 0 ∧ b = a + 1]
4 else
5    b ← a - 1
       [a ≤ 0 ∧ b = a − 1]
```

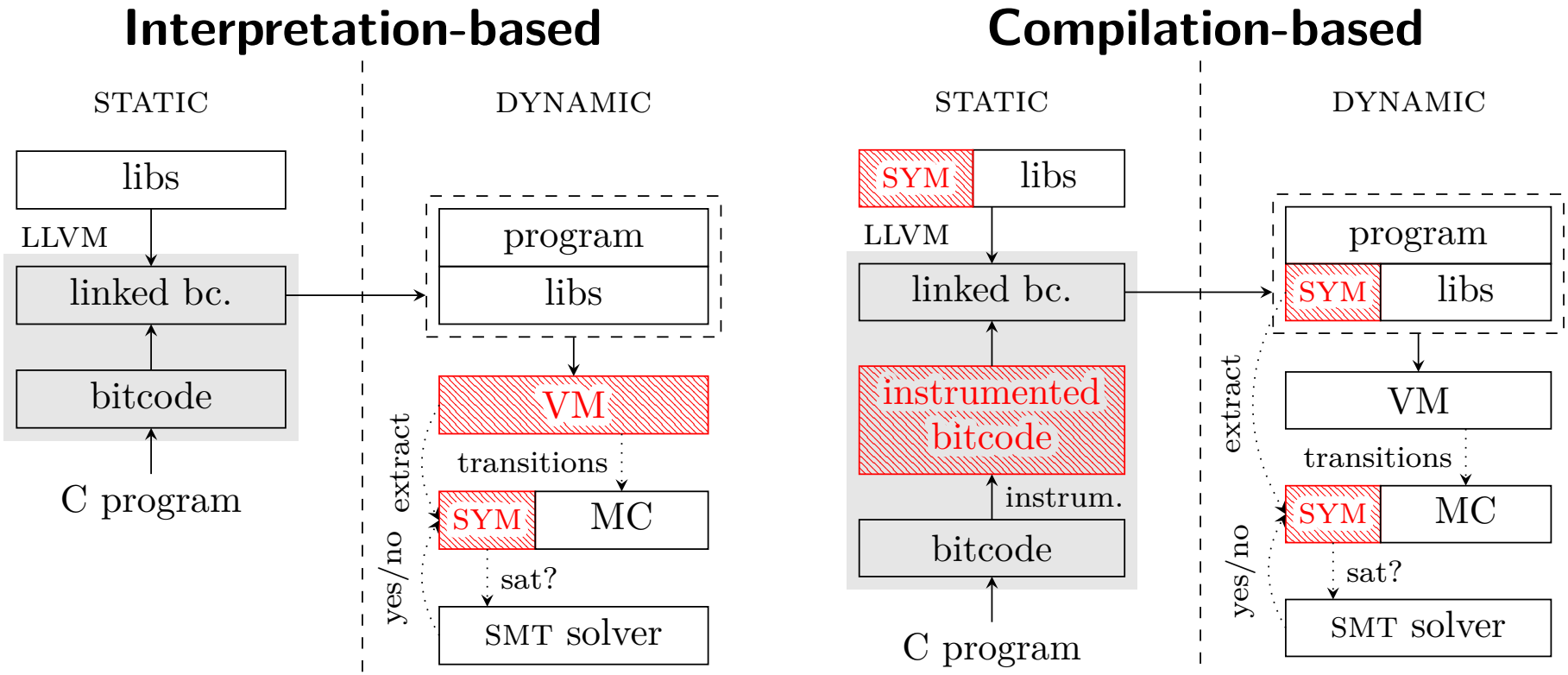
# Symbolic Computation



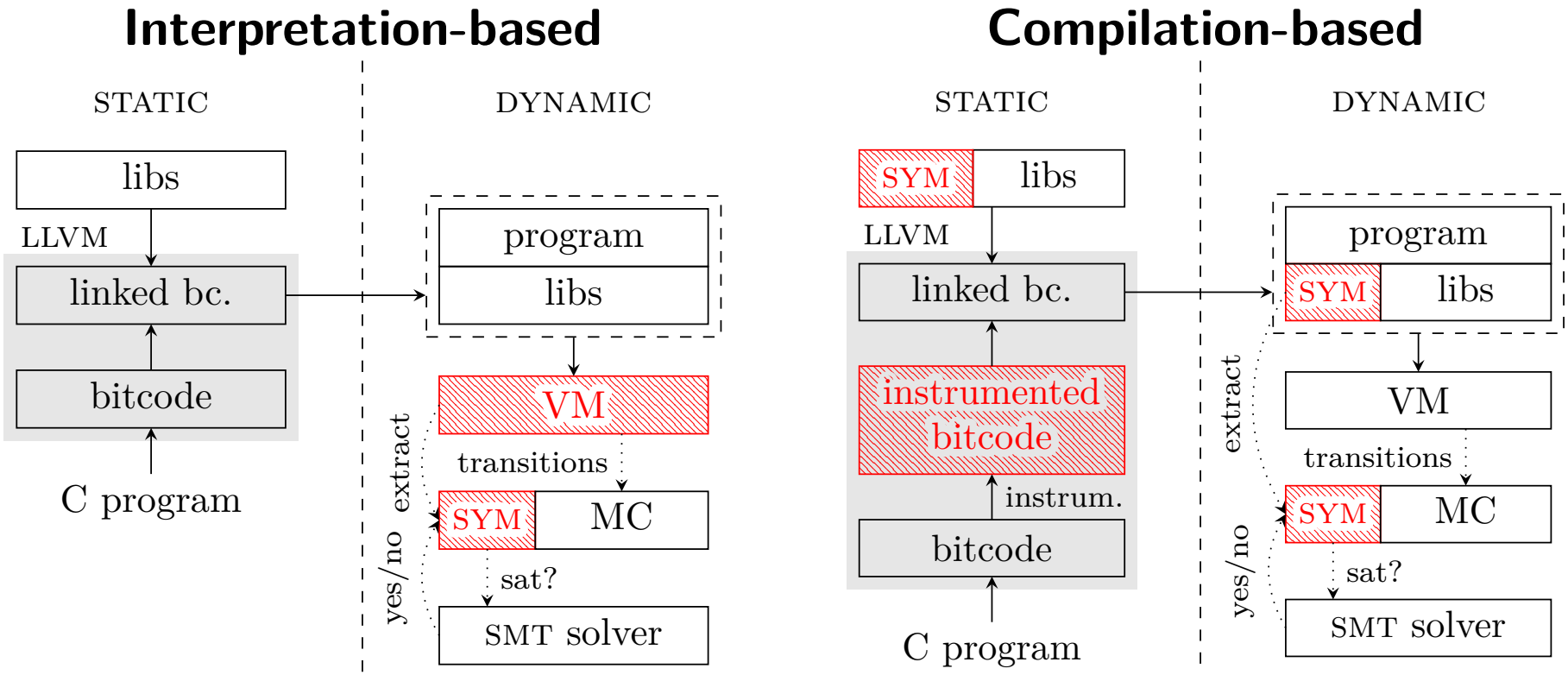

```
1 a ← input()    [true]
2 if (a > 0)     [a > 0]
3    b ← a + 1
        [a > 0 ∧ b = a + 1]
4 else
5    b ← a - 1
        [a ≤ 0 ∧ b = a − 1]
```

- multiple possible paths
- maintained in interpreter
- program does not know about symbolic values

# Proposed Symbolic Computation



# Proposed Symbolic Computation



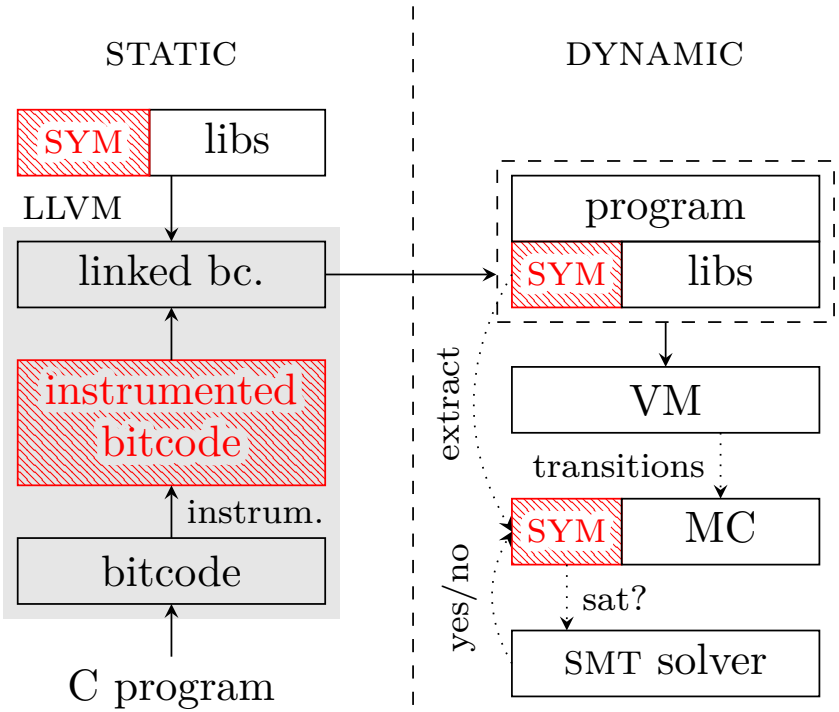

**Motivation:** minimize complexity of the verification tool

# Proposed Symbolic Computation

```
1 a ← input()
2 if (a > 0)
3   b ← a + 1
4 else
5   b ← a - 1
```

```
1 a ← sym_input()
2 if (sym_gt(a, 0))
3   b ← sym_add(a, 1)
4 else
5   b ← sym_sub(a, 1)
```

**Compilation-based**




**Motivation:** minimize complexity of the verification tool

# Goals

1. mixing of explicit and symbolic computation

# Goals

1. mixing of explicit and symbolic computation
2. expose a small interface to the rest of the system

# Goals

1. mixing of explicit and symbolic computation
2. expose a small interface to the rest of the system
3. impose minimal run-time overhead

# Transformation

# Transformation of Program

1. **syntactically abstract the input program**

```
x:int  ← input()          x: a_int  ← lift(*)
y:int  ← factorial(7)     y: int    ← factorial(7)
z:int  ← x + y            z: a_int  ← a_add(x, lift(y))
b:bool ← z < 0            b: a_bool ← a_lt(z, lift(0))
```

# Transformation of Program

1. **syntactically abstract the input program**

```
x:int  ← input()          x: a_int  ← lift(*)
y:int  ← factorial(7)     y: int    ← factorial(7)
z:int  ← x + y            z: a_int  ← a_add(x, lift(y))
b:bool ← z < 0            b: a_bool ← a_lt(z, lift(0))
```

- transform instructions, types, functions
- preserve concrete computation
- lift concrete values

# Transformation of Program

## 1 syntactically abstract the input program

```
x:int  ← input()          x: a_int  ← lift(*)
y:int  ← factorial(7)     y: int    ← factorial(7)
z:int  ← x + y            z: a_int  ← a_add(x, lift(y))
b:bool ← z < 0            b: a_bool ← a_lt(z, lift(0))
```

- transform instructions, types, functions
- preserve concrete computation
- lift concrete values

## 2 concretely realize abstraction

```
x: sym_int  ← lift(*)
y: int      ← factorial(7)
z: sym_int  ← sym_add(x, lift(y))
b: sym_bool ← sym_lt(z, lift(0))
```

- replace abstract calls with provided implementation

# Control Flow of Symbolic Program

**Problem:** constrained values by control flow

```
x: int ← input()
cond: bool ← x < 0
if (cond)
  y: int ← x + 1
else
  ...
```

- both paths can happen
- x is not constrained

# Control Flow of Symbolic Program

**Problem:** constrained values by control flow

```
x: int ← input()
cond: bool ← x < 0
if (cond)
  y: int ← x + 1
else
  ...
```

- both paths can happen
- x is not constrained

**Solution:** instrument constraint propagation

```
x: sym_int ← lift(*)
cond: sym_bool ← sym_lt(x, 0)
if (*) // nondeterministic
  x': sym_int ← assume(cond)
  y : sym_int ← sym_add(x', 1)
else
  x': sym_int ← assume(!cond)
  ...
```

- assumes extend a path condition

# Types in the Symbolic Program

**Problem:** how to deal with aggregate types?

```
arr: int[]  ← [1, 2, 3]
arr[1]: int ← input()
```

- we want to minimize the number of symbolic values

# Types in the Symbolic Program

**Problem:** how to deal with aggregate types?

```
arr: int[] ← [1, 2, 3]
arr[1]: int ← input()
```

- we want to minimize the number of symbolic values

**Solution:** use discriminated union type

- realize abstract value as union of *concrete* and *symbolic* value

```
arr: union[] ← [1, 2, 3] // either int or sym_int
arr[1]: union ← lift(*)
```

- similarly deal with recursive structures

# Function Calls

**Problem:** how to transform functions with symbolic arguments?

```
int foo(a: int, b: int, c: int)
```

# Function Calls

**Problem:** how to transform functions with symbolic arguments?

```
int foo(a: int, b: int, c: int)
```

- may produce exponentially many duplicates:

```
int foo(a: sym_int, b: int, c: int)
int foo(a: int, b: sym_int, c: int)
int foo(a: int, b: int, c: sym_int)
int foo(a: sym_int, b: sym_int, c: int)
...
```

- resolve return type

# Function Calls

**Problem:** how to transform functions with symbolic arguments?

```
int foo(a: int, b: int, c: int)
```

- may produce exponentially many duplicates:

```
int foo(a: sym_int, b: int, c: int)
int foo(a: int, b: sym_int, c: int)
int foo(a: int, b: int, c: sym_int)
int foo(a: sym_int, b: sym_int, c: int)
...
```

- resolve return type

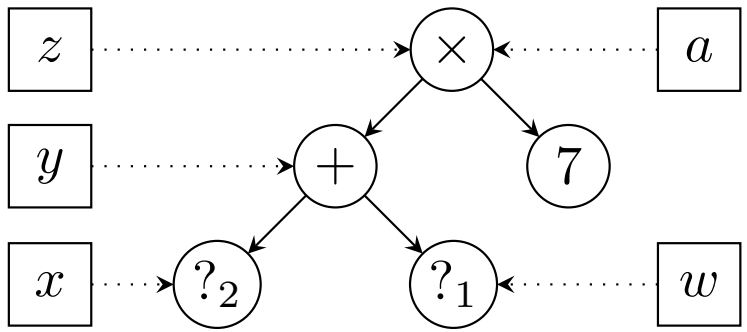
**Solution**: static analysis + use discriminated union

```
union foo(a: union, b: union, c: int)
```

# Symbolic Runtime

**Symbolic execution:**

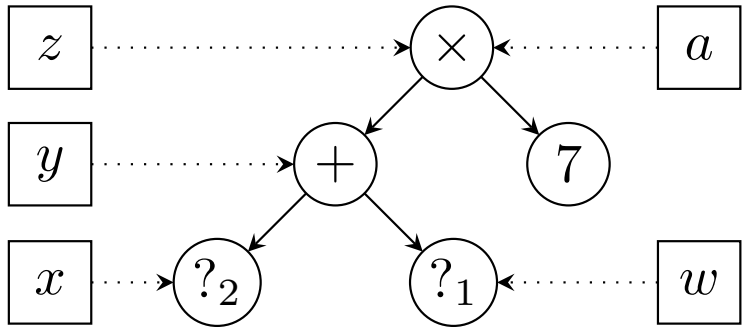
```
a: pointer ← malloc()
w: sym_int ← lift(*)
x: sym_int ← lift(*)
y: sym_int ← sym_add(w, x)
z: sym_int ← sym_mul(y, 7)
store z → a
```



# Data Representation

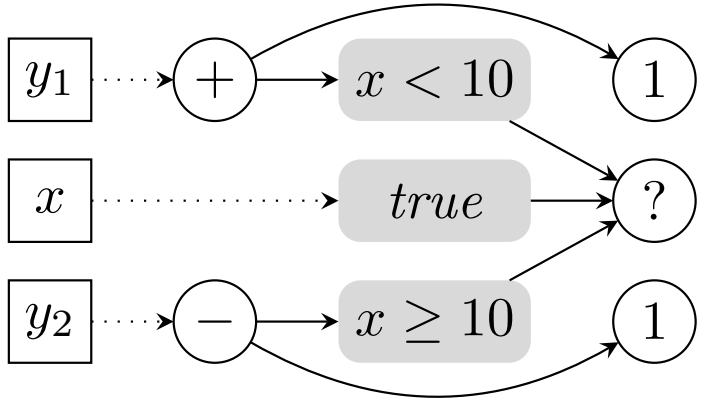
## Symbolic execution:

```
a: pointer ← malloc()
w: sym_int ← lift(*)
x: sym_int ← lift(*)
y: sym_int ← sym_add(w, x)
z: sym_int ← sym_mul(y, 7)
store z → a
```




## Branching example:

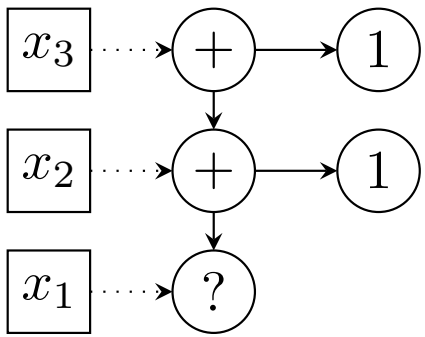
```
x : sym_int ← lift(*)
if (*) // nondeterministic
  x': sym_int ← assume(x < 10)
  y : sym_int ← sym_add(x', 1)
else
  x': sym_int ← assume(x >= 10)
  y : sym_int ← sym_sub(x', 1)
```



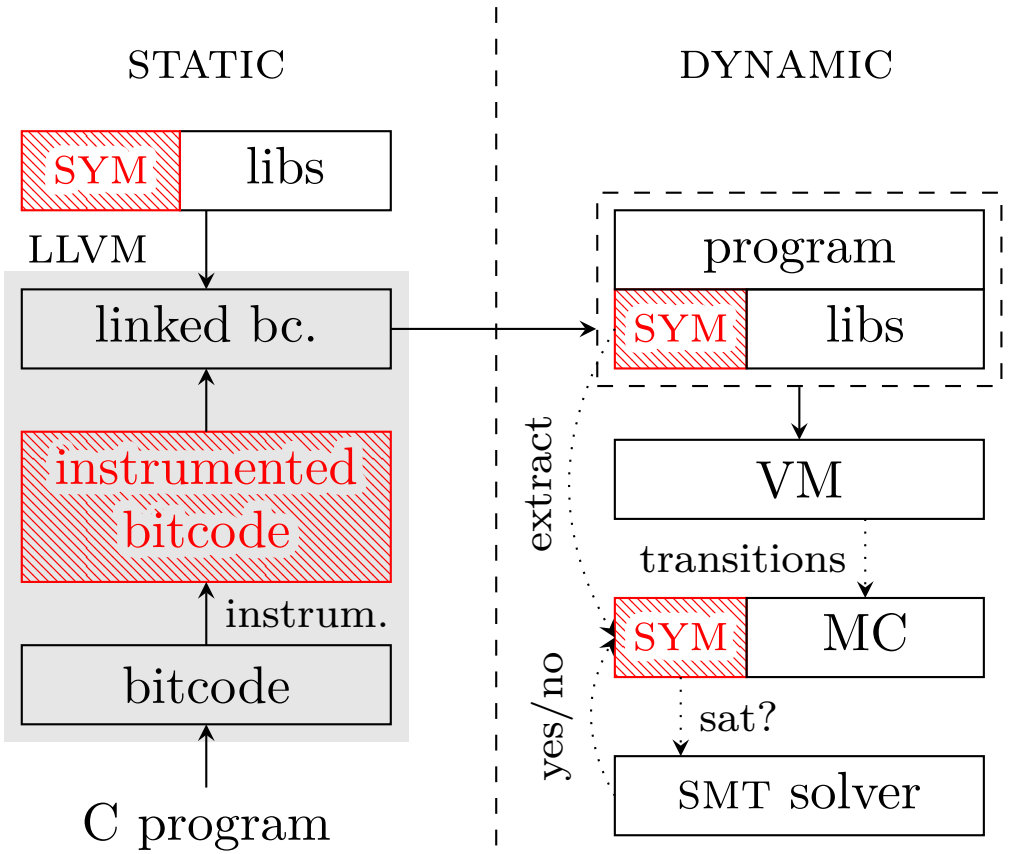
# Data Representation II.

**Cycle example:**

```
x : sym_int ← lift(*)
for i: int ← 1 .. 2
  x: sym_int ← sym_add(x, 1)
```



# Symbolic Verification Algorithm




**Required support in a tool:**

- nondeterminism
- feasibility check
- equality check
- values metadata

Simpler domains do not even need *SMT* support (sign domain).

# Results

Integrated with DIVINE model checker:

- LLVM-to-LLVM transformation
- STP SMT solver

# Results

Integrated with DIVINE model checker:

- LLVM-to-LLVM transformation
- STP SMT solver

**Component sizes:** (lines of code)

| | DIVINE* | KLEE | SymDIVINE | CBMC |
|---|---|---|---|---|
| symbolic support | 5.4 | 24.2 | 7 | 39.8 |
| shared code | 136.5 | 125 | 423 | 27.5 |

- reduced complexity of verification tool

# Results

**SV-COMP Benchmarks:**

| tag | total | DIVINE* | SymDIVINE | CBMC |
|---|---|---|---|---|
| array | 190 | **96** | 68 | 93 |
| bitvector | 32 | **17** | 9 | 2 |
| loops | 178 | **72** | 67 | 9 |
| product-lines | 575 | 336 | **411** | 234 |
| pthread | 45 | **9** | 0 | 1 |
| recursion | 81 | **47** | 43 | 22 |
| systemc | 59 | 14 | **27** | 0 |
| **total** | 1160 | 591 | **625** | 361 |

# Conclusion

**Goals**

1. mixing of explicit and symbolic computation ✓
2. expose a small interface to the rest of the system ✓
3. impose minimal run-time overhead ✓

# Conclusion

## Goals

1. mixing of explicit and symbolic computation ✓
2. expose a small interface to the rest of the system ✓
3. impose minimal run-time overhead ✓

## Summary

- introduced compilation-based symbolic verification
- generalized approach to the abstraction of programs