The Application of Predicate Logic

A Quick Recap

1.1 Propositional Logic

Step 0. Proof.

Step 1. Convert program into mathematical formula.

Step 2. Ask the computer to solve the formula.

1.2 First Order Logic

Step 1. Convert it into first order logic formula.

Step 2. Ask the computer to solve the formula.

1.3 Auto-active Proof

Step 1. Convert it into Hoare logic formula

Step 2. Ask the computer to check the invariants

Outline – This Lecture

- Sudoku
- Test Cases Generation
- Program Verification
- Program Synthesis
- Superoptimization

Sudoku

DEFINITION

Goal: Fill in a 9×9 grid with numbers

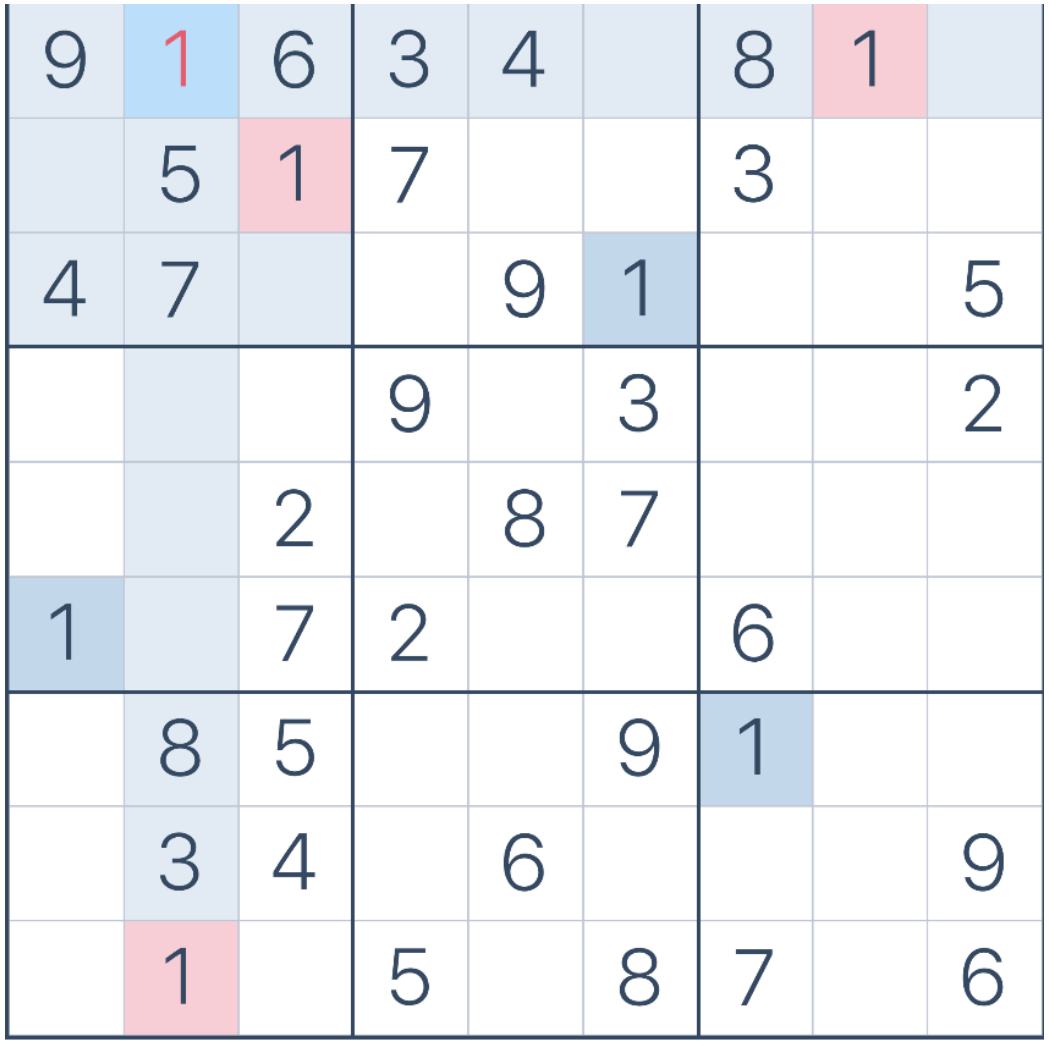
Rule:

- Each cell contains a number with a value in {1, ..., 9}
- Each row contains a number exactly once
- Each column contains a number exactly once
- Each 3x3 square contains a number exactly once
- Some numbers have been filled in

9		6	3	4		8	1	
	5	1	7			3		
4	7			9	1			5
			9		3			2
		2		8	7			
1		7	2			6		
	8	5			9	1		
	3	4		6				9
	1		5		8	7		6

Sudoku Example

9		6	3	4		8	1	
	5	1	7			3		
4	7			9	1			5
			9		3			2
		2		8	7			
1		7	2			6		
	8	5			9	1		
	3	4		6				9
	1		5		8	7		6



Solve Sudoku with Z3

```
from z3 import *

# sudoku instance, we use '0' for empty cells
instance = ((9, 0, 6, 3, 4, 0, 8, 1, 0),
            (0, 5, 1, 7, 0, 0, 3, 0, 0),
            (4, 7, 0, 0, 9, 1, 0, 0, 5),
            (0, 0, 0, 9, 0, 3, 0, 0, 2),
            (0, 0, 2, 0, 8, 7, 0, 0, 0),
            (1, 0, 7, 2, 0, 0, 6, 0, 0),
            (0, 8, 5, 0, 0, 9, 1, 0, 0),
            (0, 3, 4, 0, 6, 0, 0, 0, 9),
            (0, 1, 0, 5, 0, 8, 7, 0, 6))

# 9x9 matrix of integer variables
X = [[Int("x_%s_%s" % (i+1, j+1)) for j in range(9)]
                                  for i in range(9)]
```

9		6	3	4		8	1	
	5	1	7			3		
4	7			9	1			5
			9		3			2
		2		8	7			
1		7	2			6		
	8	5			9	1		
	3	4		6				9
	1		5		8	7		6

Solve Sudoku with Z3

```
# Each cell contains a number with a value in {1, ..., 9}
  cells_c = [And(1 <= X[i][j], X[i][j] <= 9) for i in range(9) for j in range(9)]

# Each row contains a number exactly once
  rows_c = [Distinct(X[i]) for i in range(9)]

# Each column contains a number exactly once
  cols_c = [Distinct([X[i][j] for i in range(9)]) for j in range(9)]

# Each 3x3 square contains a number exactly once
  sq_c = [Distinct([X[3*i0 + i][3*j0 + j] for i in range(3) for j in range(3)])
                                          for i0 in range(3) for j0 in range(3)]

# Set filled numbers in the instance
  instance_c = [If(instance[i][j] == 0, True, X[i][j] == instance[i][j])
                                          for i in range(9) for j in range(9)]
```

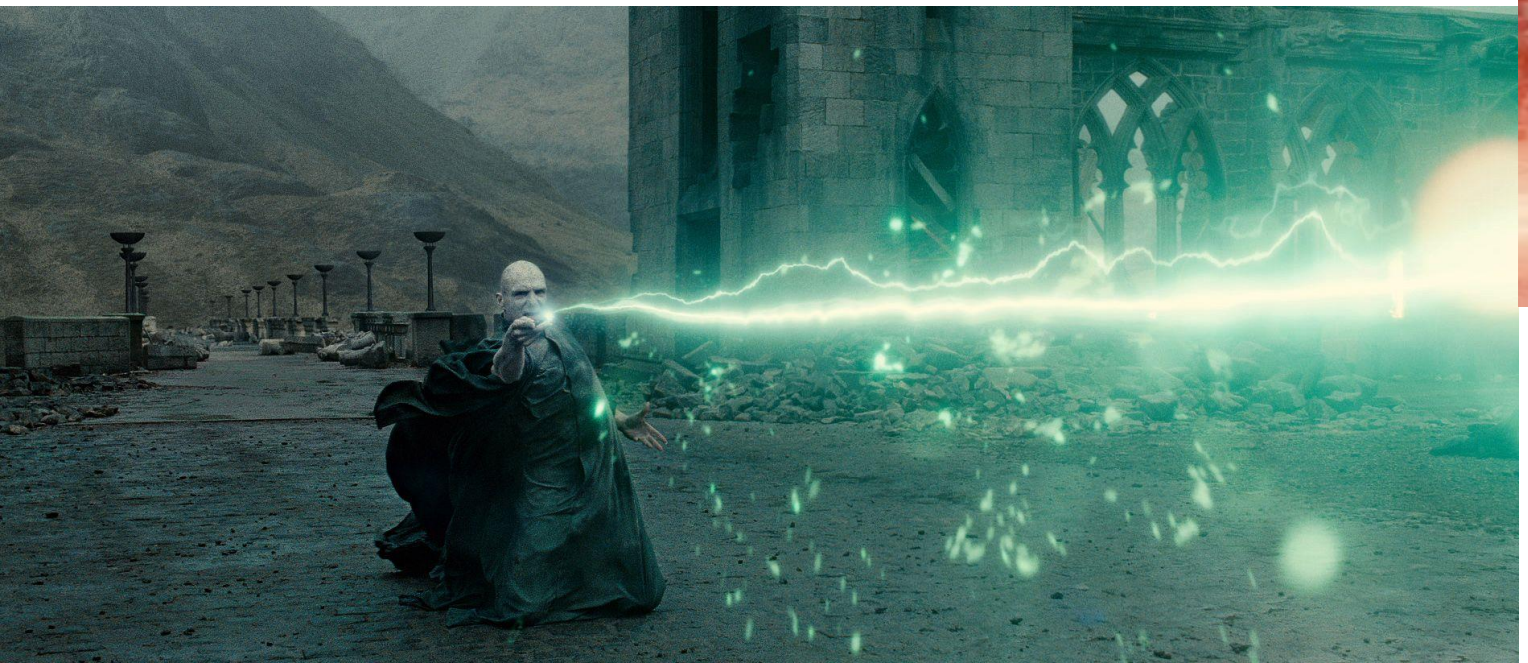
Solve Sudoku with Z3

```
# Add all constraints
sudoku_c = cells_c + rows_c + cols_c + sq_c + instance_c

s = Solver()
s.add(sudoku_c)

# Solve
if s.check() == sat:
   m = s.model()
   r = [[m.evaluate(X[i][j]) for j in range(9)]
        for i in range(9)]
   print_matrix(r)
else:
   print("failed to solve")
```

9		6	3	4		8	1	
	5	1	7			3		
4	7			9	1			5
			9		3			2
		2		8	7			
1		7	2			6		
	8	5			9	1		
	3	4		6				9
	1		5		8	7		6

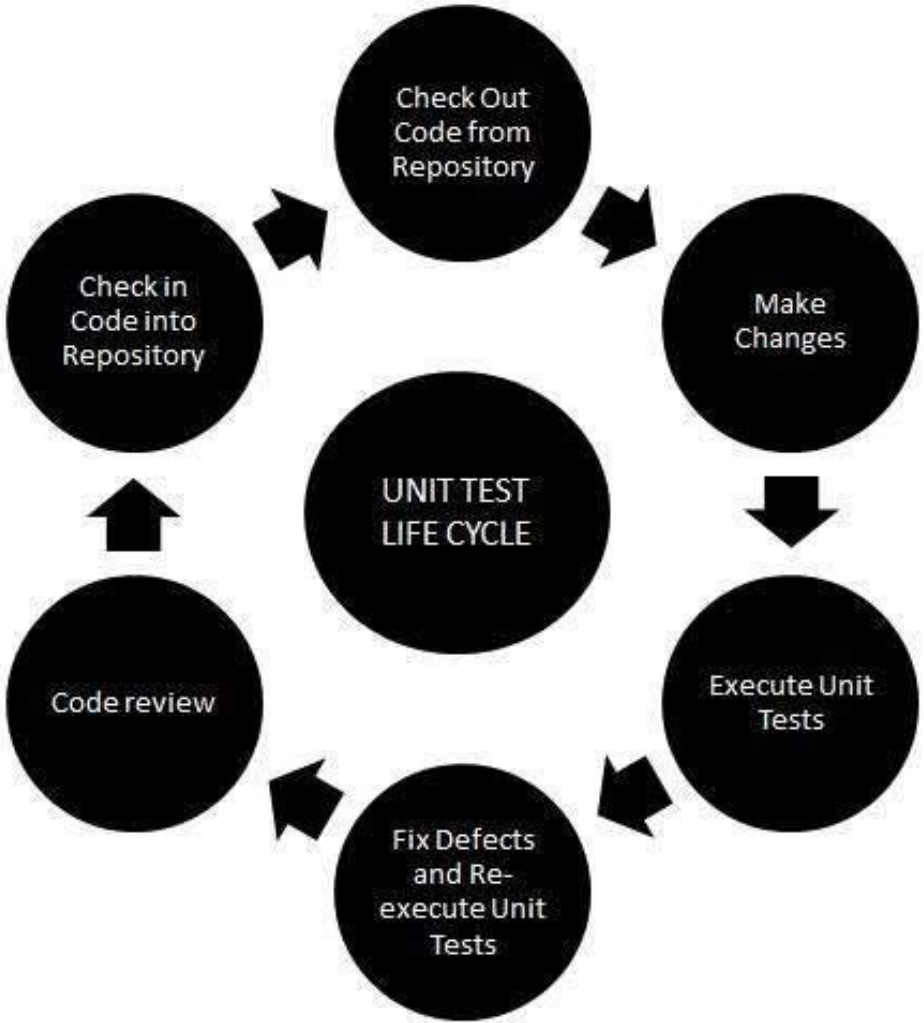
```
> ./sudoku.py
  [[9, 2, 6, 3, 4, 5, 8, 1, 7],
  [8, 5, 1, 7, 2, 6, 3, 9, 4],
  [4, 7, 3, 8, 9, 1, 2, 6, 5],
  [5, 6, 8, 9, 1, 3, 4, 7, 2],
  [3, 4, 2, 6, 8, 7, 9, 5, 1],
  [1, 9, 7, 2, 5, 4, 6, 3, 8],
  [6, 8, 5, 4, 7, 9, 1, 2, 3],
  [7, 3, 4, 1, 6, 2, 5, 8, 9],
  [2, 1, 9, 5, 3, 8, 7, 4, 6]]
```

SMT solver is a magic wand



Test Cases Generation

- **After writing a program, we should manually write test cases**



Test Cases Generation

- **After writing a program, we should manually write test cases**
- **How to estimate the quality of test cases?**

DEFINITION

Code coverage(代码覆盖率) means the ratio of the LoC executed under test cases, which should be as higher as possible.

We can achieve higher code coverage when more execution paths are covered under testing

Test Cases Generation

- **After writing a program, we should manually write test cases**
- **How to estimate the quality of test cases?**

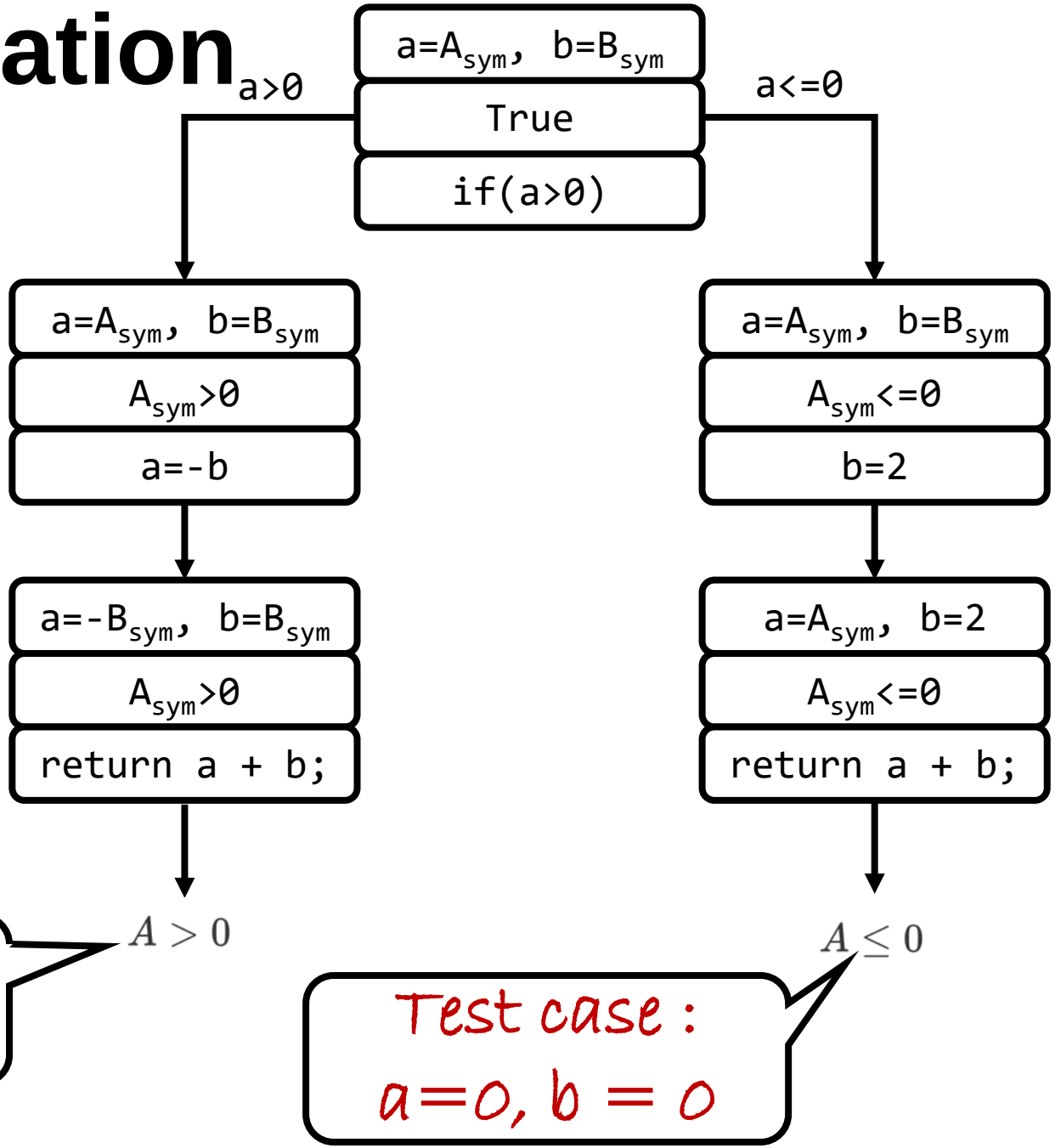
DEFINITION

Code coverage(代码覆盖率) means the ratio of the LoC executed under test cases, which should be as higher as possible.

Could we automatically generate test cases with high code coverage?

Test Cases Generation

```
void func(int a, int b) {
  if(a > 0)
    a = -b;
  else
    b = 2;
  return a + b;
}
```



Code coverage : 100%

Test case :
a=1, b = 0

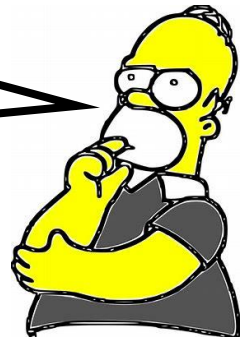
Test case :
a=0, b = 0

Test Cases Generation

```
void func(int a, int b) {
  if(a > 0)
    a = -b;
  else
    b = 2;
  return a + b;
}
```

- **Test case 1: a = 1, b = 0; output: 0**
- **Test case 2: a = 0, b = 0; output: 2**

When code coverage is 100% and the code passes all test cases, can we ensure the absence of bugs?



Test Cases Generation

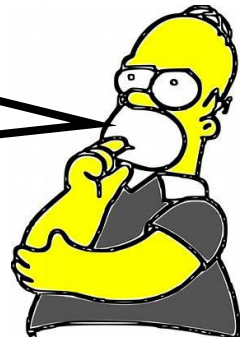
```
void func(int a, int b) {
  if(a > 0)
    a = -b;
  else
    b = 2;
  return a + b;
}
```

- **Test case 1: a = 1, b = 0; output: 0**
- **Test case 2: a = 0, b = 0; output: 2**

When we require that the return value should always be less than 10

- **The program passes the testing**
- **But it is incorrect (a = 0, b = 100)**

Software testing cannot ensure the correctness, which is complete but not sound.



Test Cases Generation

KLEE

Getting Started Documentation Tutorials Publications Projects Getting Involved Releases

KLEE Symbolic Execution Engine

KLEE is a dynamic symbolic execution engine built on top of the LLVM compiler infrastructure, and available under the UIUC open source license. For more information on what KLEE is and what it can do, see the OSDI 2008 paper.

OSDI 2008 Best Paper

http://klee.github.io/

Example

```
#include "klee/klee.h”

int get_sign(int x) {
  if (x == 0) return 0;
  if (x < 0) return -1;
  else return 1;
}
int main() {
  int a;
  klee_make_symbolic(&a, sizeof(a), "a");
  return get_sign(a);
}
```

test cases

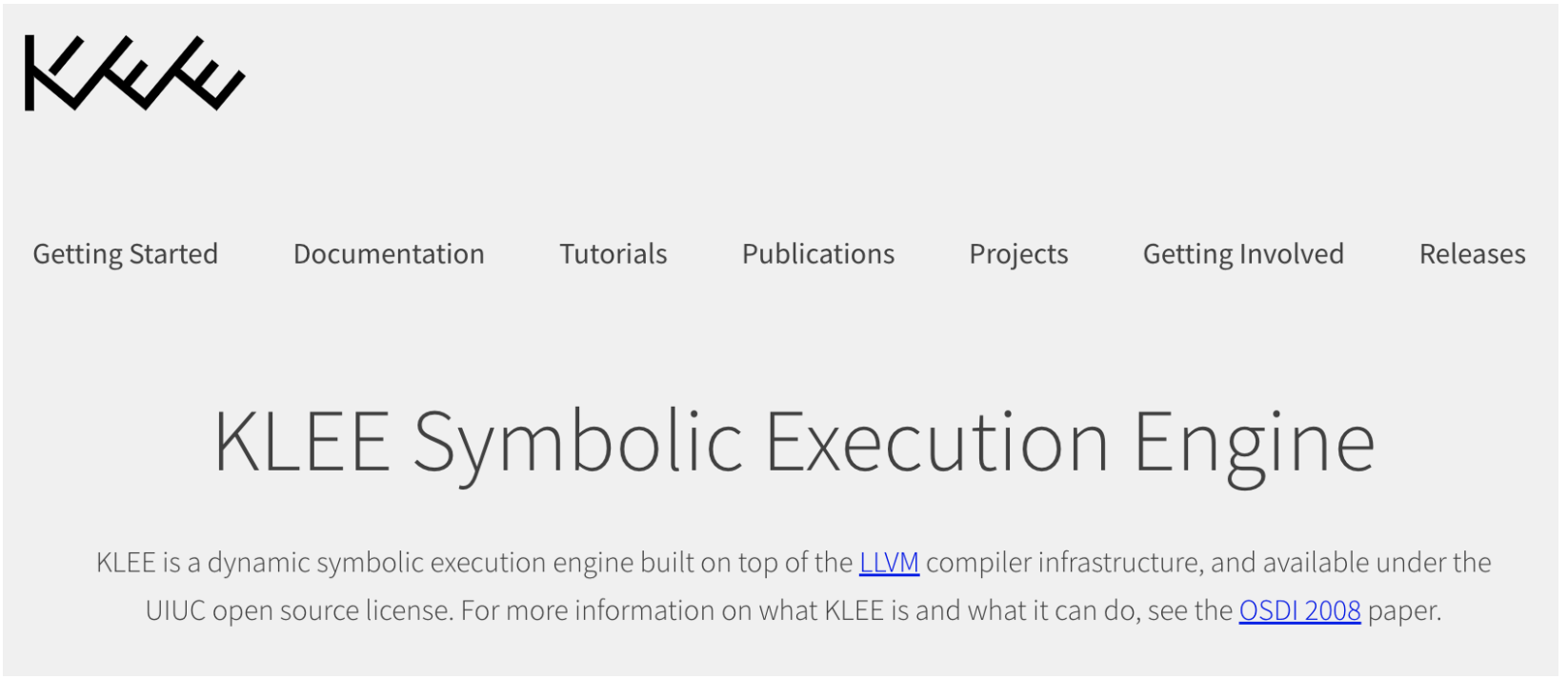
```
> ktest-tool test000001.ktest
  object 0: name: 'a’
  object 0: int : 0
> ktest-tool test000002.ktest
  object 0: name: 'a’
  object 0: int : 16843009
> ktest-tool test000003.ktest
  object 0: name: 'a’
  object 0: int : -2147483648
```

retest

```
> KTEST_FILE=test000001.ktest ./a.out
  0
> KTEST_FILE=test000002.ktest ./a.out
  1
> KTEST_FILE= test000003.ktest ./a.out
  -1
```

Program Verification

- **We still need program verification**
- **KLEE can also be used to perform verification**



Example

```
#include "klee/klee.h"
// Reverse the order of bytes in x
uint64_t myhtobe64(uint64_t x){
  uint64_t addr = 0;
  uint8_t byte = 0;
  for (int i = 0; i < sizeof(x) * 8; i+=8) {
     byte = (x >> i) & 0xFF;
     addr |= byte;
     addr <<= 8;
  }
  assert(addr == htobe64(x));
  return addr;
}

int main(int argc, char** argv) {
   uint64_t a;
   klee_make_symbolic(&a, sizeof(a), "a");
   myhtobe64(a);
   return 0;
}
```

```
> ./run.sh
  KLEE: done: total instructions = 352
  KLEE: done: completed paths = 1
  KLEE: done: partially completed paths =
1
  KLEE: done: generated tests = 2
  ……
  ktest-tool ./klee-last/test000001.ktest
  a.out: main.c:18: myhtobe64: Assertion
`addr == htobe64(x)' failed.
```

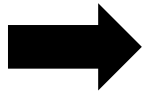
Debug with testcases

```
> ./debug.sh
  ./klee-last/test000001.ktest
  Input:0X0100000000000000
  Myhtobe64:0x100
  Htobe64:0x1
```

Example

```
#include "klee/klee.h"
// Reverse the order of bytes in x
uint64_t myhtobe64(uint64_t x){
  uint64_t addr = 0;
  uint8_t byte = 0;
  for (int i = 0; i < sizeof(x) * 8; i+=8) {
     byte = (x >> i) & 0xFF;
     addr |= byte;
     if(i < (sizeof(x)-1) * 8)
       addr <<= 8;
  }
  assert(addr == htobe64(x));
  return addr;
}

int main(int argc, char** argv) {
   uint64_t a;
   klee_make_symbolic(&a, sizeof(a), "a");
   myhtobe64(a);
   return 0;
}
```

retest



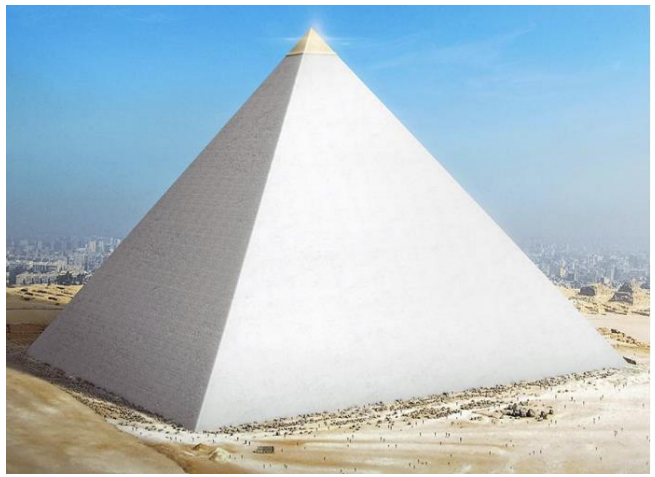
```
> ./run.sh
  KLEE: done: total instructions = 387
  KLEE: done: completed paths = 1
  KLEE: done: partially completed paths
= 0
  KLEE: done: generated tests = 1
```

An Interesting Problem

Problem

Can we discard software testing with program verification?

Program Testing



Program Verification

Pharaoh Sneferu’s Pyramid Project

Try 1:
Meidum(52° angle)

Try 2:
Dashur/Bent(52° to 43.5° angle)

Try 3:
Red pyramid(43° angle)

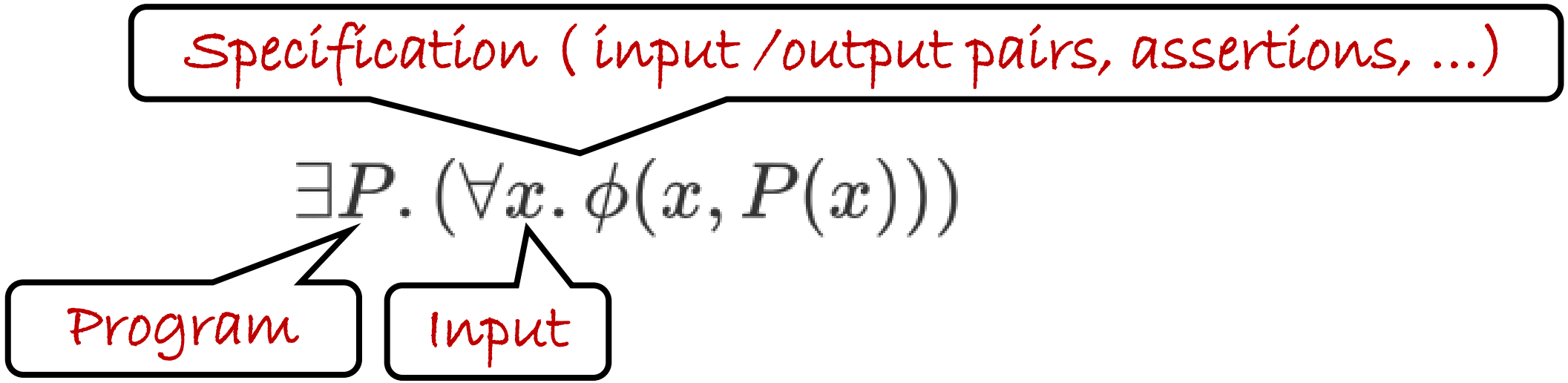
Program Synthesis

- Program verification tells if the program is incorrect
- It does not tell why the code is incorrect
- Programmers need to manually
  - locate the codes introducing bugs
  - think how to fix the bug

Program Synthesis

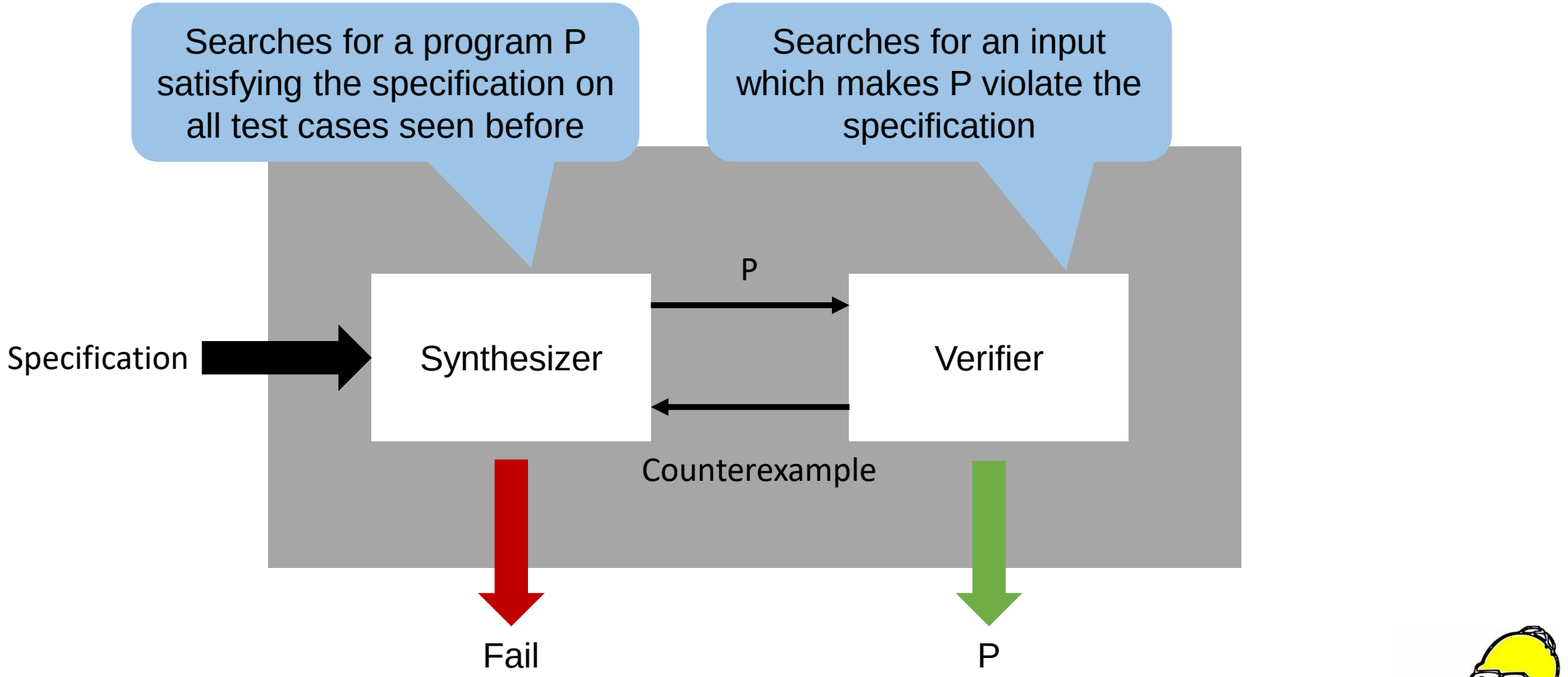
DEFINITION

Program synthesis(程序合成) is automatically finding a program that satisfies the user intent expressed in the form of some specification.



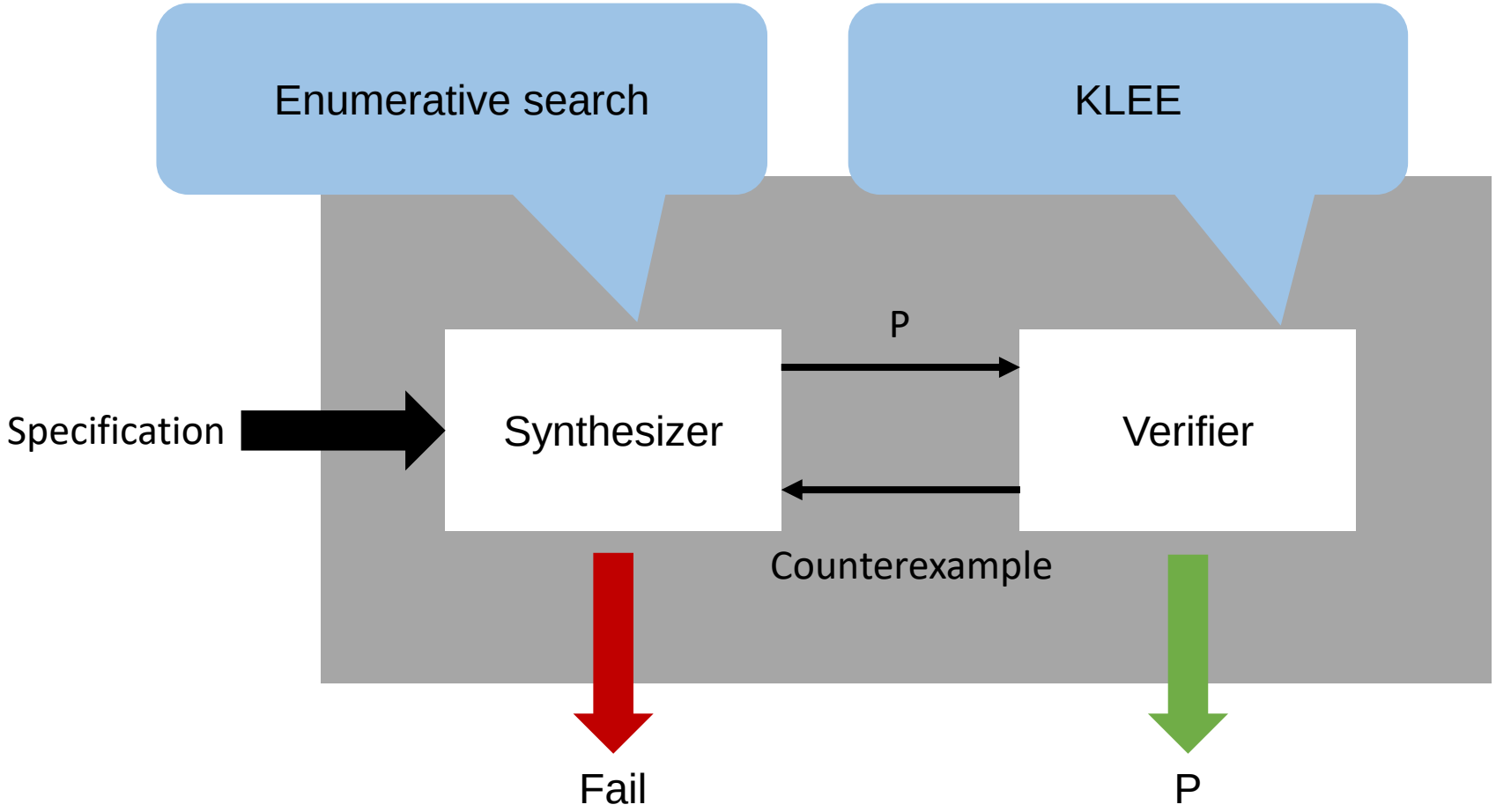
https://www.microsoft.com/en-us/research/publication/program-synthesis/

Program Synthesis



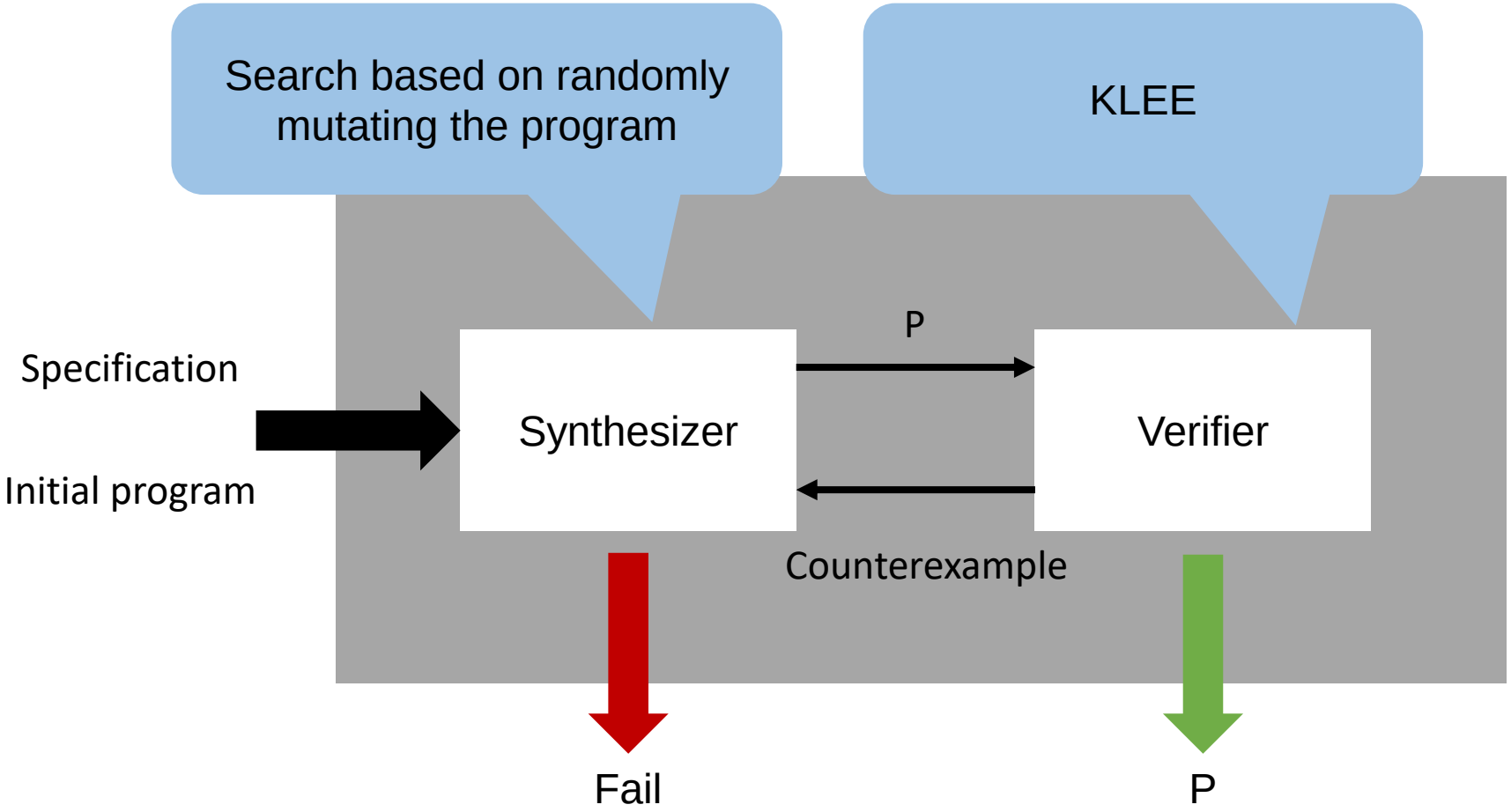
How to implement the synthesizer and the verifier?

Program Synthesis



It may be hard to find the program based on enumerative search

Program Synthesis



Symbolic execution cannot handle unbounded loops

Program Synthesis

Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis

Sergey Mechtaev Jooyong Yi Abhik Roychoudhury
School of Computing, National University of Singapore, Singapore
{mechtaev,jooyong,abhik}@comp.nus.edu.sg

ICSE 2016

https://github.com/jyi/fangelix

Print Number from n-1 to 0

```
#define ANGELIX_OUTPUT(type, expr, id) expr

#include <stdio.h>
#ifndef ANGELIX_OUTPUT
#define ANGELIX_OUTPUT(type, expr, id) expr
#endif

int main(int argc, char *argv[]) {
  int n;
  n = atoi(argv[1]);
  // some complex logic...

  while (n > 1) {
    n--;
    printf("%d\n", ANGELIX_OUTPUT(int, n, "n"));
  }
  return 0;
}
```

oracle

```
Test cases:
"input": 2 "n": [1, 0]
"input": 3 "n": [2, 1, 0]
"input": 4 "n": [3, 2, 1, 0]
```

Print Number from n-1 to 0

```
#define ANGELIX_OUTPUT(type, expr, id) expr

#include <stdio.h>
#ifndef ANGELIX_OUTPUT
#define ANGELIX_OUTPUT(type, expr, id) expr
#endif

int main(int argc, char *argv[]) {
  int n;
  n = atoi(argv[1]);
  // some complex logic...

  while (n > 1) {
    n--;
    printf("%d\n", ANGELIX_OUTPUT(int, n, "n"));
  }
  return 0;
}
```

oracle

```
Test cases:
"input": 2 "n": [1, 0]
"input": 3 "n": [2, 1, 0]
"input": 4 "n": [3, 2, 1, 0]
```

patch

```
--- a/test.c
+++ b/test.c
@@ -27,7 +27,7 @@
     i--;
   }

-  while (n > 1) {
+  while (n >= 1) {
     n--;
}
```

Superoptimization

- Correctness is not the only goal of programming
- Performance is also very important
- However, program synthesis does not guarantee the performance

Superoptimization

"Given an instruction set, the superoptimizer finds the shortest program to compute a function"

Superoptimizer -- A Look at the Smallest Program

Henry Massalin

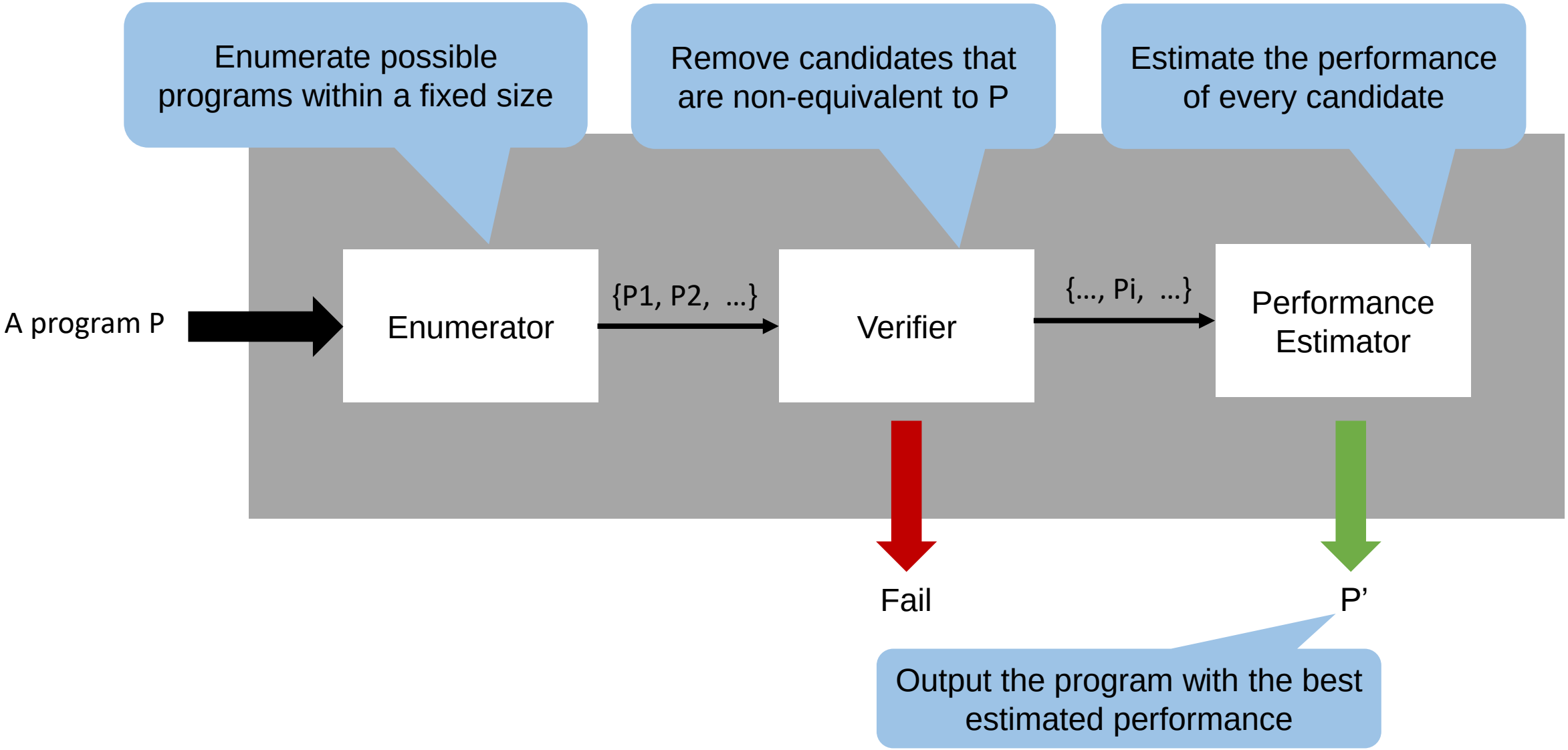
Department of Computer Science
Columbia University
New York, NY 10027

Abstract

Given an instruction set, the superoptimizer finds the shortest program to compute a function. Startling programs have been generated, many of them engaging in convoluted bit-fiddling bearing little resemblance to the source programs which defined the functions. The key idea in the superoptimizer is a probabilistic test that makes exhaustive searches practical for programs of useful size. The search space is defined by the processor's instruction set, which may include the whole set, but it is typically restricted to a subset. By constraining the instructions and observing the effect on the output program, one can gain insight into the design of instruction sets. In addition, superoptimized programs may be used by peephole optimizers to improve the quality of generated code, or by assembly language programmers to improve manually written code.

https://courses.cs.washington.edu/courses/cse501/15sp/papers/massalin.pdf

Superoptimization

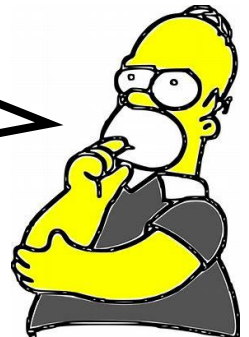


Superoptimization

```
int func1(int a) {
  if(a > 0)
    return 0;
  else
    return 1;
}
```

```
int func2(int a) {
  a = a + a;
  if(a > 0)
    return 0;
  else
    return 1;
}
```

How to verify the equivalence between two programs?



Superoptimization

```
int func1(int a) {
  if(a > 0)
    return 0;
  else
    return 1;
}
```

```
int func2(int a) {
  a = a + a;
  if(a > 0)
    return 0;
  else
    return 1;
}
```

```
void func(int a) {
  int val1 = func1(a);
  int val2 = func2(a);
  assert(val1 == val2);
}
```

Given the same input, func1 and func2 always return the same value.

We can still use symbolic execution engines

Superoptimization

```
int func1(int a) {
  if(a > 0)
    return 0;
  else
    return 1;
}
```

```
int func2(int a) {
  a = a + a;
  if(a > 0)
    return 0;
  else
    return 1;
}
```

A simple method of estimating the performance

- **Estimate the execution time of every instruction**
- **Compute the sum of the execution time of all instructions**

The performance of func1 is better than func2

How to estimate the performance?

Superoptimization

Stochastic Superoptimization

Eric Schkufza
Stanford University
eschkufz@cs.stanford.edu

Rahul Sharma
Stanford University
sharmar@cs.stanford.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

ASPLOS 2013

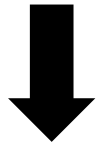
https://github.com/StanfordPL/stoke

Example

```
/* The number of 1s in the binary
   representation */
size_t popcnt(uint64_t x) {
  int res = 0;
  for (; x > 0; x >>= 1) {
    res += x & 0x1;
  }
  return res;
}

int main(int argc, char** argv) {
  const auto itr = atoi(argv[1]);
  auto ret = 0; uint64_t j = 1;
  for (auto i = 0; i < itr; ++i) {
    j = (j*19 + 7 + (rand() % 5));
    ret += popcnt(j);
  }
  std::cout << ret << std::endl;
  return 0;
}
```

bins/_Z6popcntm.s

```
# Text                  #  Line  RIP       Bytes  Opcode
._Z6popcntm:            #        0x4009a0  0      OPC=<label>
  testq %rdi, %rdi      #  1     0x4009a0  3      OPC=testq_r64_r64
  je .L_4009bf          #  2     0x4009a3  2      OPC=je_label
  xorl %eax, %eax       #  3     0x4009a5  2      OPC=xorl_r32_r32
.L_4009b0:              #        0x4009b0  0      OPC=<label>
  movl %edi, %edx       #  13    0x4009b0  2      OPC=movl_r32_r32
  andl $0x1, %edx       #  14    0x4009b2  3      OPC=andl_r32_imm8
  addl %edx, %eax       #  15    0x4009b5  2      OPC=addl_r32_r32
  shrq $0x1, %rdi       #  16    0x4009b7  3      OPC=shrq_r64_one
  jne .L_4009b0         #  17    0x4009ba  2      OPC=jne_label
  cltq                  #  18    0x4009bc  2      OPC=cltq
  retq                  #  19    0x4009be  1      OPC=retq
.L_4009bf:              #        0x4009bf  0      OPC=<label>
  xorl %eax, %eax       #  20    0x4009bf  2      OPC=xorl_r32_r32
  retq                  #  21    0x4009c1  1      OPC=retq
```



result.s

```
# Text                  #  Line  RIP       Bytes  Opcode
._Z6popcntm:            #        0x4009a0  0      OPC=<label>
  popcntq %rdi, %rax    #  1     0x4009a0  5      OPC=popcntq_r64_r64
  setl %ch              #  2     0x4009a5  3      OPC=setl_rh
  retq                  #  3     0x4009a8  1      OPC=retq
```

Large Language Model

- Large language model (LLM) is more and more popular AI tool now
- People can use LLM to do many useful things
  - Improve paper writing
  - Check typos in documents
  - Meeting assistant
  - …

Large Language Model

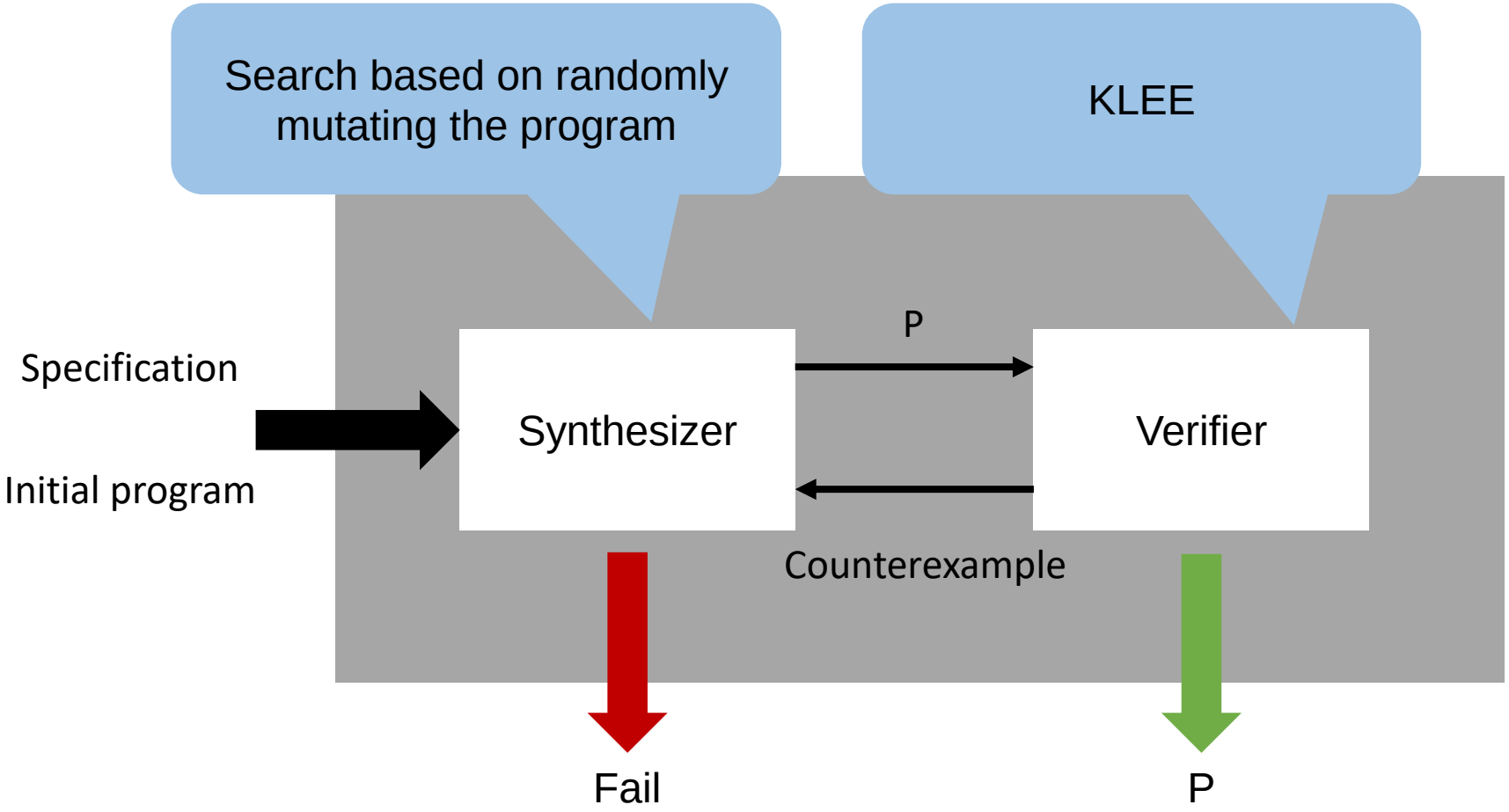
- Large language model (LLM) is more and more popular AI tool now
- People can use LLM to do many useful things
  - Improve paper writing
  - Check typos in documents
  - Meeting assistant
  - …

What can we do by combining LLM with verification?

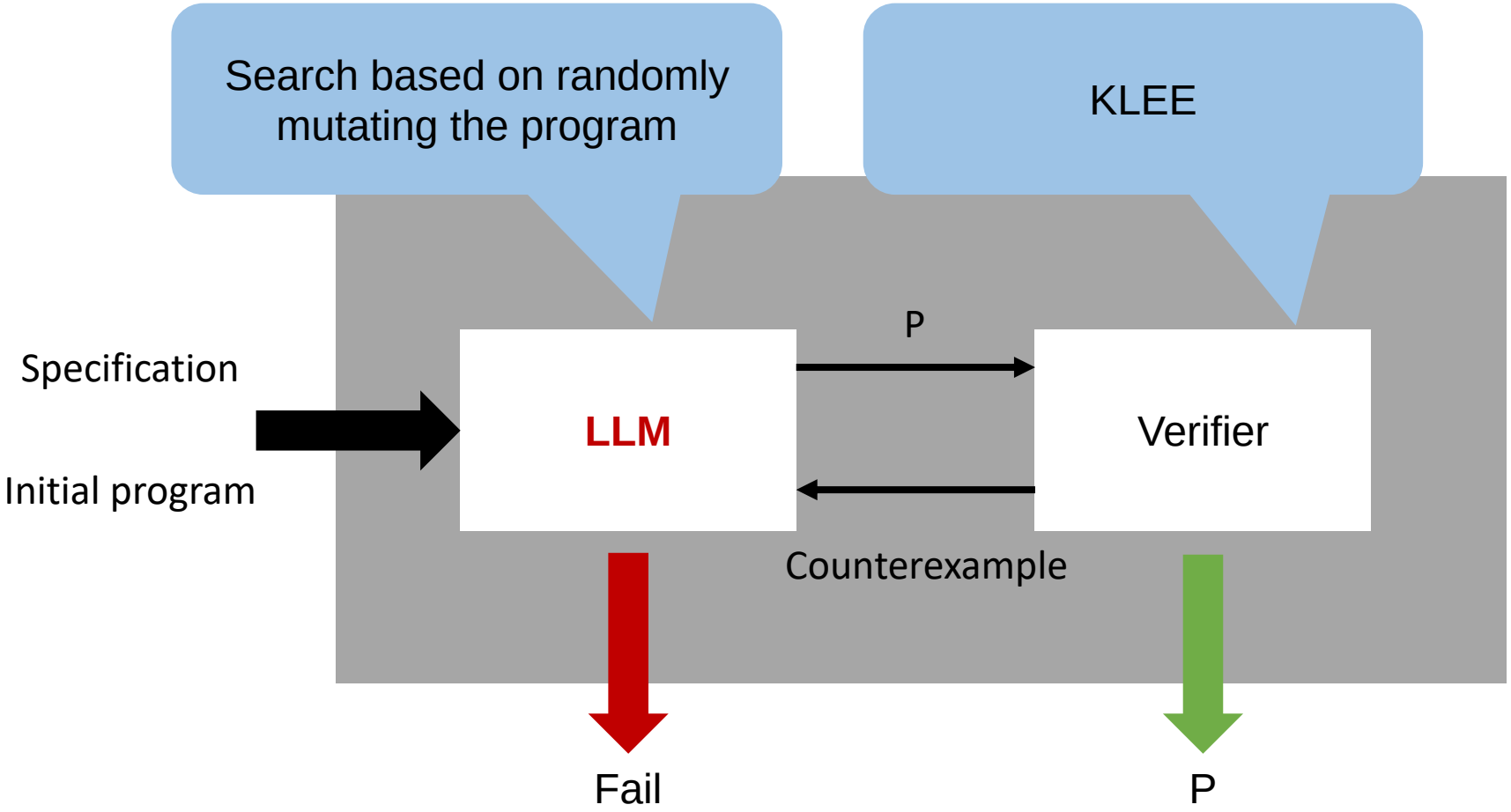
What can LLM do?

- Program Verification 
- Program Synthesis 
- Superoptimization 

Program Synthesis



LLM + Program Synthesis



Example

Please fix the following incorrect function such that it can print 2, 1, 0 when the input is 2:

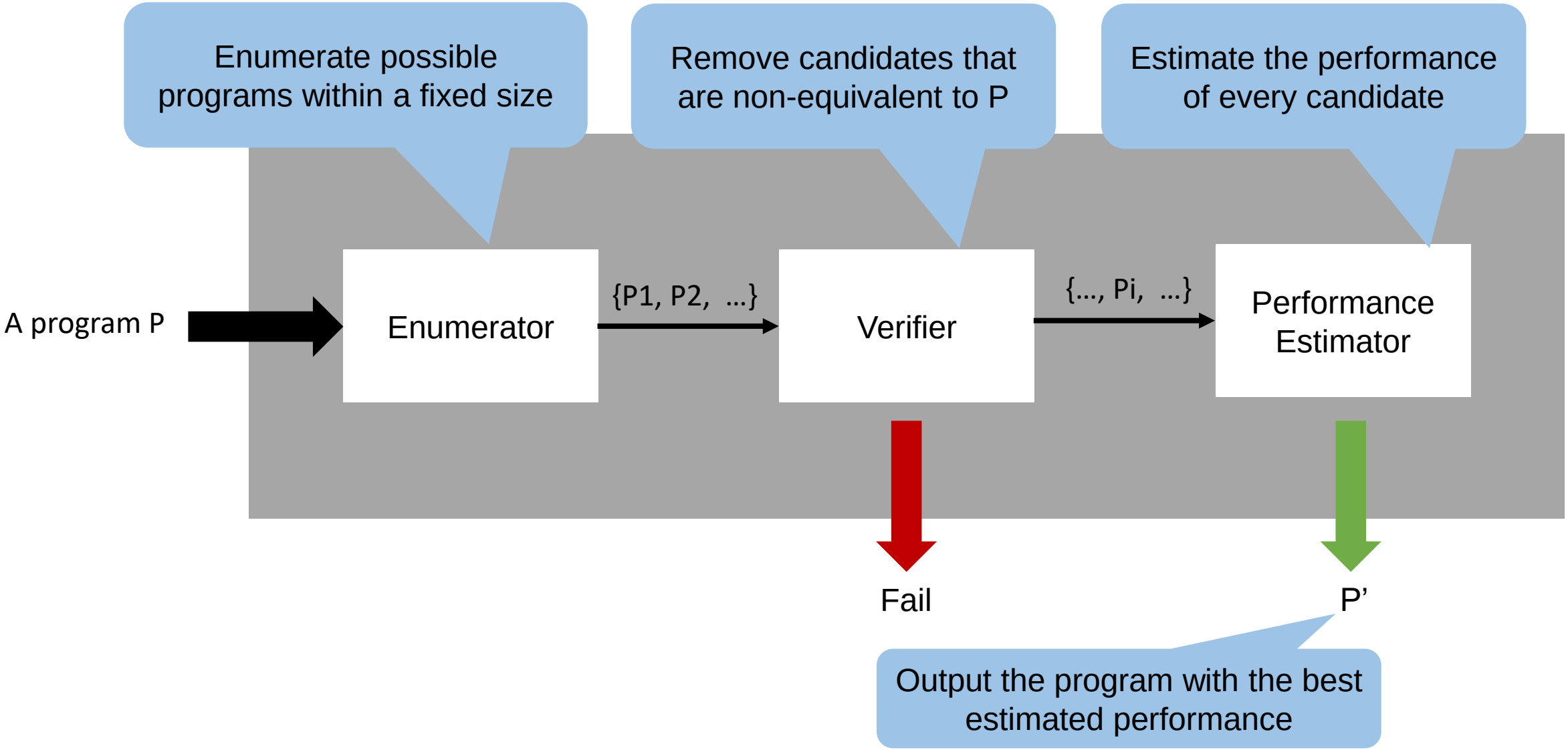
```
void func(int input) {
  while (input > 1) {
    printf("%d ", input);
    input --;
  }
}
```

To make the function print 2, 1, 0 when the input is 2, you can modify the function as follows:

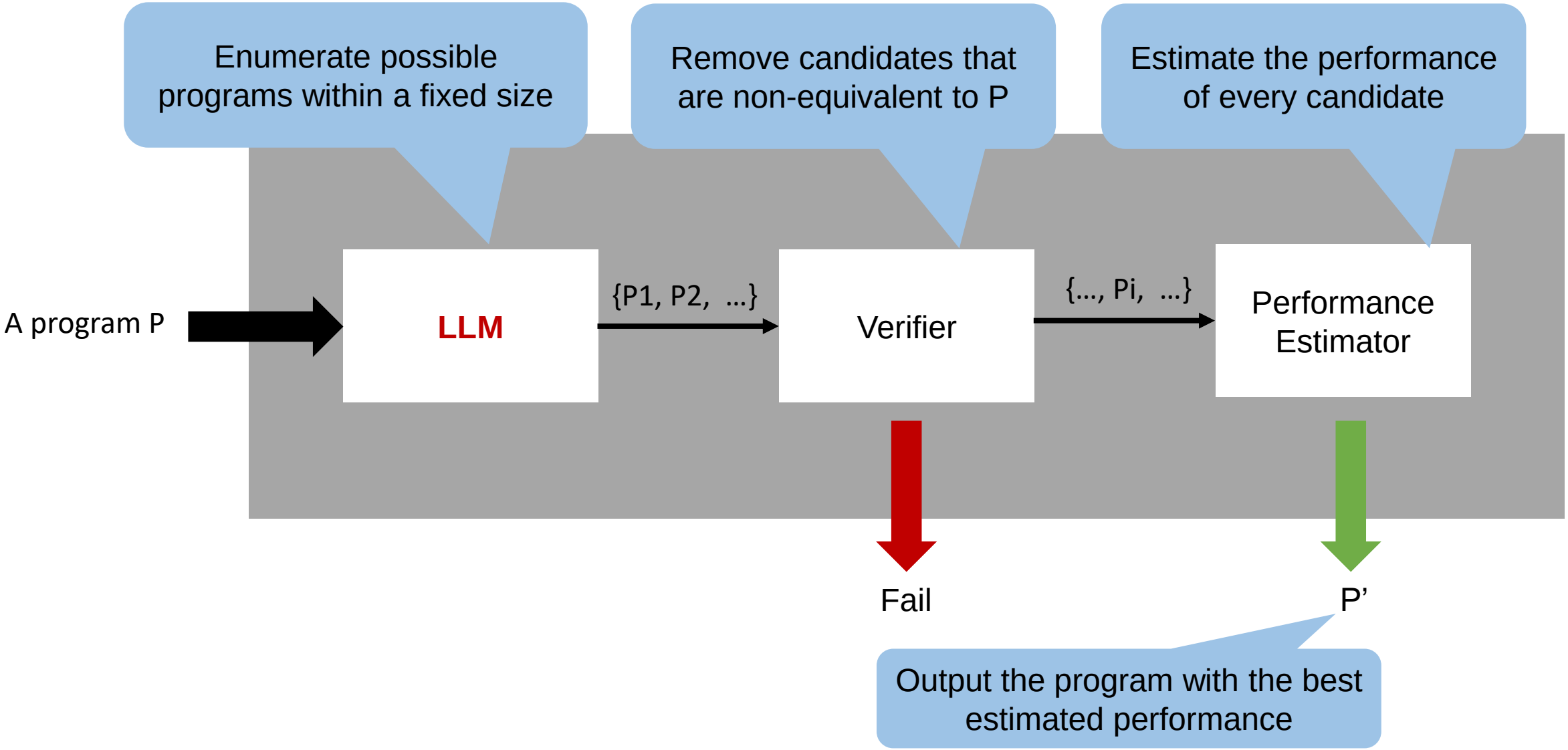
```
void func(int input) {
  while (input >= 0) {
    printf("%d ", input);
    input --;
  }
}
```

With this modification, the function will print 2, 1, 0 when the input is 2.

Superoptimization



LLM + Superoptimization



Example

Please rewrite the following function to improve its performance but not change its computation result:

```
int func(int input) {
  int i = 0;
  for ( ; i < input; ++ i);
  return i + i;
}
```

To improve the performance of your function while maintaining its computation result, we can eliminate the loop entirely. The original function effectively counts up to input, and then returns i + i, which is equivalent to 2 * input. Here's a more efficient version:

```
int func(int input) {
  return 2 * input;
}
```

Program Analysis

```
int select(int a, int b, int c)
{
  if(c >= 0)
    return a;
  else
    return b;
}
```

Prove that the return value is greater than zero if both a and b are greater than zero.

Program Analysis

```
int select(int a, int b, int c)
{
  if(c >= 0)
    return a;
  else
    return b;
}
```

$$(a > 0) \wedge$$

$$(b > 0) \wedge$$

$$((c \geq 0) \rightarrow (result = a)) \wedge$$

$$(\neg(c \geq 0) \rightarrow (result = b))$$

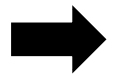
$$\rightarrow (result > 0)$$

Prove it is universally valid.

Program Analysis

$$(a > 0) \wedge$$
$$(b > 0) \wedge$$
$$((c \geq 0) \rightarrow (result = a)) \wedge$$
$$(\neg(c \geq 0) \rightarrow (result = b))$$
$$\rightarrow (result > 0)$$

Prove it is universally valid.



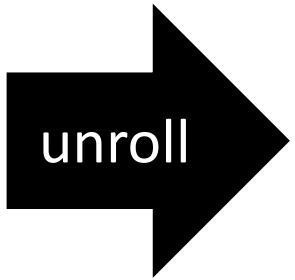
$$(a > 0) \wedge$$
$$(b > 0) \wedge$$
$$((c \geq 0) \rightarrow (result = a)) \wedge$$
$$(\neg(c \geq 0) \rightarrow (result = b)) \wedge$$
$$\neg(result > 0)$$

Prove it is unsatisfiable.

Loops

```
void f(unsigned a)
{
    unsigned i = 3;
    a = 0;
    while(i > 0)
    {
        a = a + 1;
        i = i - 1;
    }
    assert(a == 3);
}
```

Loops

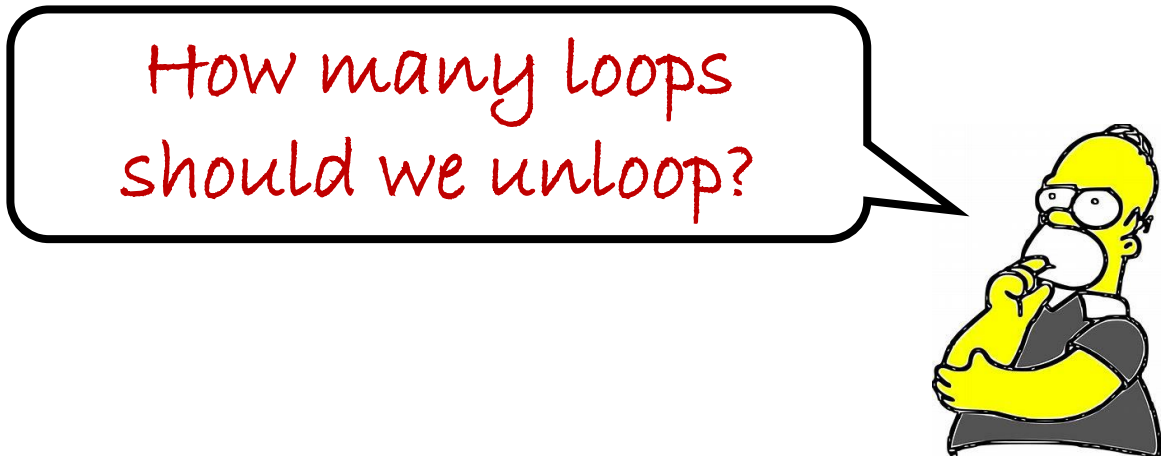



```
void f(unsigned a)
{
    unsigned i = 3;
    a = 0;
    a = a + 1;
    i = i - 1;
    a = a + 1;
    i = i - 1;
    a = a + 1;
    i = i - 1;
    assert(a == 3);
}
```

Straight-Line Code

Loops

```
void f(unsigned a)
{
    unsigned i = 0;
    while(i < a)
    {
        i = i + 1;
    }
    assert(i == a);
}
```



Loops

```
void f(unsigned a)
{
    unsigned i = 0;
    while(i < a)
    {
        i = i + 1;
    }
    assert(i == a);
}
```

Dafny : A More Powerful Program Verifier

- Write a program and its proof with the same programming language
- Still convert a program into a predicate logic formula
- https://dafny.org/

Dafny: An Automatic Program Verifier for Functional Correctness

K. Rustan M. Leino

Microsoft Research, Redmond, WA, USA
leino@microsoft.com

Abstract. Traditionally, the full verification of a program's functional correctness has been obtained with pen and paper or with interactive proof assistants, whereas only reduced verification tasks, such as extended static checking, have enjoyed the automation offered by satisfiability-modulo-theories (SMT) solvers. More recently, powerful SMT solvers and well-designed program verifiers are starting to break that tradition, thus reducing the effort involved in doing full verification.

This paper gives a tour of the language and verifier Dafny, which has been used to verify the functional correctness of a number of challenging pointer-based programs. The paper describes the features incorporated in Dafny, illustrating their use by small examples and giving a taste of how they are coded for an SMT solver. As a larger case study, the paper shows the full functional specification of the Schorr-Waite algorithm in Dafny.

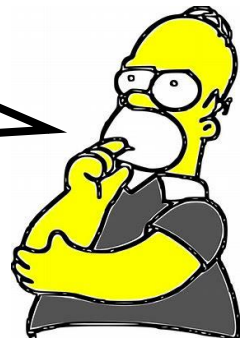
How to define correctness

```
method MultipleReturns(x: int, y: int) returns (more: int, less: int)
  ensures less < x && x < more
{
  more := x + y;
  less := x - y;
}
```

How to reason about loops

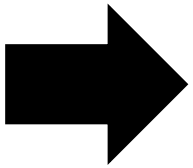
```
void f(unsigned a)
{
    unsigned i = 0;
    while(i < a)
    {
        i = i + 1;
    }
    assert(i == a);
}
```

Let's come back to he original example



How to reason about loops

```
void f(unsigned a)
{
    unsigned i = 0;
    while(i < a)
    {
        i = i + 1;
    }
    assert(i == a);
}
```



```
method f(a: nat)
{
 var i := 0;
 while i < a
 {
  i := i + 1;
 }
 assert i == a;
}
```

How to reason about loops

```
method f(a: nat)
{
  var i := 0;
  while i < a
    invariant 0 <= i <= n
  {
    i := i + 1;
  }
  assert i == a;
}
```

Thanks & Questions