Synthesizing Implementations for Native Operations

Souradeep Dutta - Nicholas Lewchenko - William Temple

Assembly-level Code is Tricky to Write

- Efficient operations require us to write x86 code
- Concepts we understand become difficult to express

$$\begin{tikzcd} \texttt{x ** y} \arrow[r] & x^y \arrow[r] & {} \end{tikzcd}$$

```
movl x, %ebx

movl y, %ecx

movl $1, %eax

loop_begin:

cmpl $0, %ebx

je loop_end

imull %ebx, %eax

decl %ecx

jmp loop_begin

loop_end:
```

- Can we define operations in our language at a higher level?

How Can Our Computer Help Us?

- What do we already know a computer can do?
- Execute! Inputs -> Outputs
    - We can compute $\mathbf{x^y}$ using an SMT solver
    - Even easier, we can compute **`x**y`** using a reference python interpreter
    - And we can run **x86** assembly programs!
    - These together give us **verification** that an x86 program does what we want

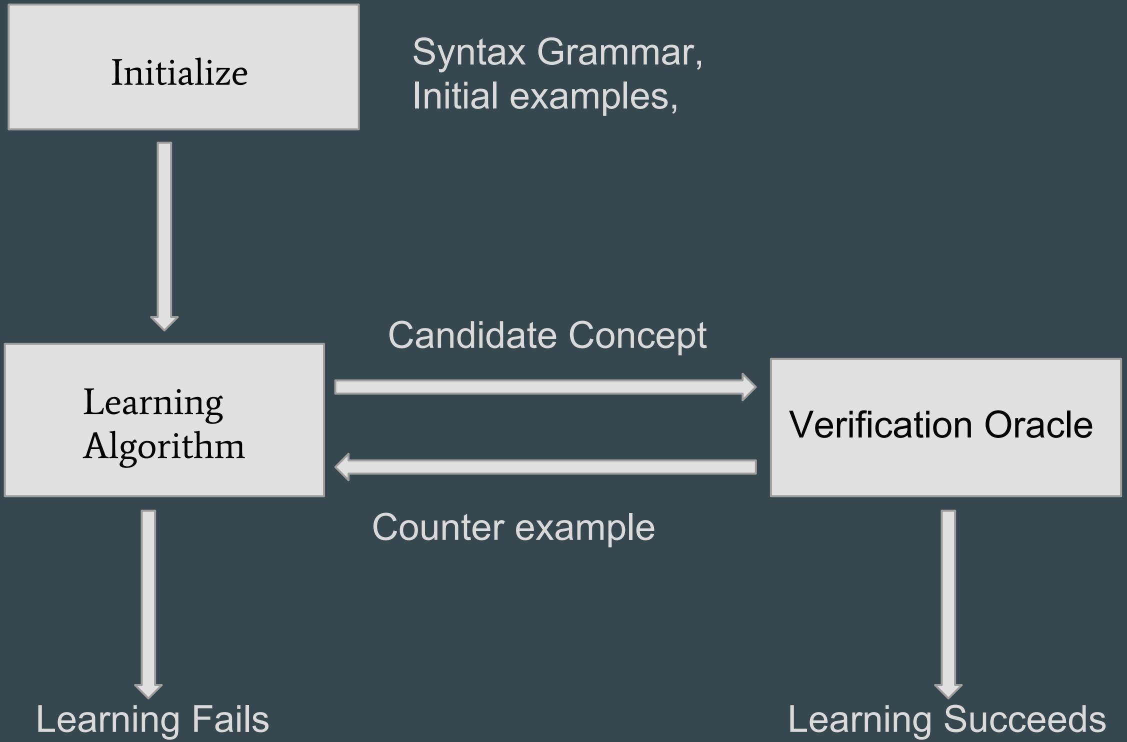
- So how do we find the x86 programs to check?
- Automated **search** followed by automated **verification** are together the essence of **program synthesis**

Syntax Guided Synthesis (SyGuS)

From a top level the function synthesis problem is basically trying to find a function ‘ f ’ such that some logical formula Q representing the correctness of ‘f’, is valid. For syntax guided synthesis problem it is constrained in three ways :

1. The interpretation of the logical symbols and connectives, restricted to the background theory
2. The specification Q , which is limited to a first order formula in the background theory
3. The possible functions is only restricted to the syntactic expressions described by the grammar.

The approach that is followed here is that of a *Counter Example Guided Inductive Synthesis* (CEGIS): that is refining the search by learning from the counter-examples provided by the verification oracle.



SyGuS - using Enumerative Learning

SKETCH Model

- The idea here is that, the programmer comes with a partial implementation of the desired program, with ‘*holes*’ , called a sketch.
- The programmer comes with a high level description of the program , and relying on the synthesizer to come up with the low-level details of the implementation. Along with this the programmer is expected to come up with assertions to specify what the synthesizer is expected to do.

‘Hello World’ of the Sketch Model

```
void function (int x)

{

        int  y  =   x   *   ??   ;

        assert ( y == x + x) ;

}
```

HOLES

ASSERTIONS

Our Model - vx86 Enumeration

- Fully-enumerative synthesis over vx86 (Virtual x86)
  - Program ::= Instr Program | ε

  -
    ```
    Instr ::=      `mov` R R
              |    `add` R R
              |    `mul` R R
              | ...
    ```

  -
    ```
    R ::= `r`[0-9]+
    ```
- Easy to write an interpreter
- Restricted Subset of x86 instructions
  - Runtime concerns
  - Consistency

Definitions

- **Program :** a pair (C, V) where C is a sequence of instructions and V is a set of virtual registers
- **Specification** : a set of test-cases, representable as a map from inputs to expected outputs
- **Output State :** the assignment of values to registers that results from the execution of a program ($\mathbb{N}$ -> Integer)
- **Signature** : the ordered set of Output States for a Program executed on each test case in the Specification

Synthesis Algorithm

To synthesize a k-ary operator from a set of test-cases Q:

1. **Initialize -** let C be a set of Candidate Programs, initialized to a set containing a program with no instructions and **k** virtual registers. (C = { ([ ], {r1, r2 ... rk]) }). Let V be the empty set (V = {}).
2. **Verify** - for every program P = (P_C, P_V) in C, run C with the first *k* virtual registers initialized to the inputs in each test in Q to generate its **Signature** S. If S is in V, then remove P from the set of candidate programs. Else, add S to V and if any register contains the expected output defined by Q in all Output States in S, then P satisfies Q, and we are done.

Algorithm (continued)

3. **Derivation** - Extend C by deriving new programs for each program P in C by performing both of the following:
   a. **Aliasing** - consider every P’ with an additional virtual register `rn` and an instruction `mov ri, rn` for every `ri` previously in the virtual registers set
   b. **Operation** - consider every P’ with an additional instruction (that is NOT a `mov`) for every combination of virtual registers that is possible
4. Go back to **Verification** to see if we’ve got one this time.

Simple Example

Consider sqr(x), defined by the following tests

- T({1}) = 1
- T({2}) = 4
- T({0}) = 0
- T({-1}) = 1
- T({-2 }) = 4

These tests will produce a resulting program very quickly. $\longrightarrow$

1. Initialize C = { ([ ], {vr1}) }, V = {}
2. Verify
   a. Execute ([ ], {vr1}) -> [[1],[2].[0],[-1],[-2]]
   b. V = { [[1],[2].[0],[-1],[-2]] }
   c. Incorrect program
3. Derive C = {
   a. ([mov r1, r2], {r1, r2})
   b. ([add r1, r1], {r1})
   c. ([mul r1, r1], {r1}) }
4. Back to verify

Taking it Further

- Just tests (great for TDD fans)
- We can interrogate python to generate tests, and then we can synthesize native operations from an existing interpreter
- But it’s very slow (with only add, mov, and mul)
    - <5 instructions instantly on a MacBook
    - 5 instructions in about 13 seconds
    - Generic Exponentiation? The universe will implode long before it’s done.
    - Practical for short sequences, which makes it great for native operations
    - Not guaranteed to terminate
- Questions for us?