Enumerative Synthesis of Native Compiler Transformations

Souradeep Dutta Nicholas V. Lewchenko William Temple

December 17, 2017

1 Introduction

Writing a compiler is difficult, in part because it requires a programmer to implement the core functionality of their source language in one or more assembly languages.

An assembly language is optimized for simple evaluation by a computer (simple enough that *hardware* can do it).

The consequence of this design requirement, illustrated in Figure 1, is that non-trivial assembly programs grow large, and writing or reasoning about them is tedious and error-prone for a human.

The difference in complexity between the fully-specifying formal definition of an operation and its assembly implementation is dramatic, suggesting that computer assistance in this implementation process would significantly reduce the necessary programming effort of a compiler programmer.

2 Background

One of the promising ideas in Program synthesis is that of Sketching, explained in [2]. The idea in the Sketching program model is to allow the programmer a robust scheme to tailor in the basic implementation outline while leaving some of the details to the synthesizer. The programmer is expected to come with the assertions in order to specify the correct behavior of the program while giving the outline. The SKETCH Language is basically a layer on top of a simple procedural language, with the only difference being the introduction of the integer hole **??**. Where the programmer expects the integer hole to be replaced with a suitable integer.
The "Hello World" in a SKETCH programming model can be the following program,

```
void main (int x) {
int y = x * ?? ;
```

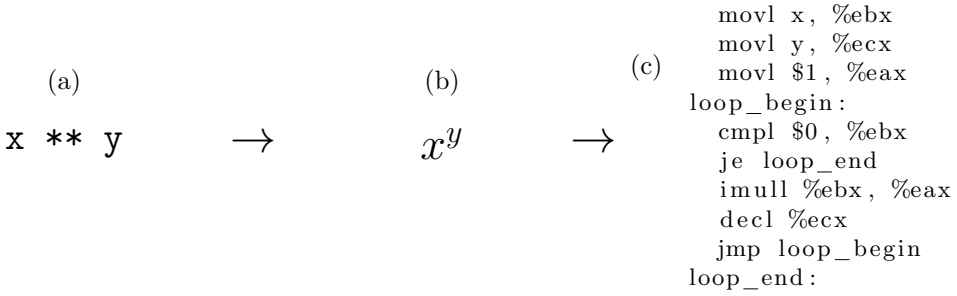

Figure 1: An exponentiation node in Python (a) translated into the arithmetic theory definition (b) and then x86 assembly implementation (c).

assert y == x + x;
}

The three basic components of a SKETCH model are the following: i) the main procedure , ii) holes, and iii) assertions. The synthesizer is only allowed to the fill up the holes with appropriate integer and not introduce any new lines.

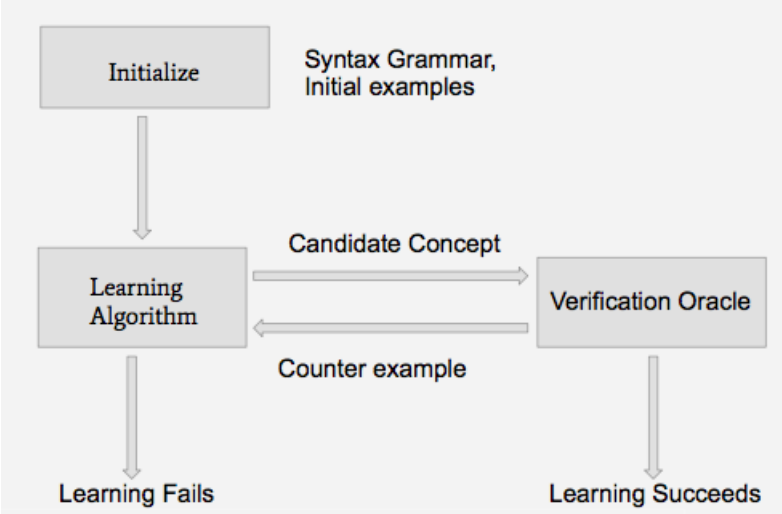

Figure 2: SyGuS procedure [1]

Another interesting idea in program synthesis is that of using enumerative learning to come up with the expression. This idea is known by the name of Syntax Guided Synthesis also known as **SyGuS** [1]. Now, from a top level the function synthesis problem is basically trying to find a function F such that some specification Φ is valid. The specification is expected to capture the correctness of the desired program. For the syntax guided synthesis problem it is constrained in the following ways:

- The interpretation of the logical symbols and connectives are restricted to the background theory only.
- The logical formula Φ is only limited to be first order formulae in the back ground theoy.
- The possible functions are only restricted to be syntactic expressions generated by the grammar.

The main procedure in syntax guided synthesis is shown in Fig 2. It essentially involves generating expressions and asking a verification oracle to check the correctness of the generated expressions. If correct then , we have the right function. Else, the oracle comes up with a counter example, and the learning uses the Counter Example to refine it's search procedure. This process is typically known in literature as a Counter Example Guided Inductive Synthesis procedure or $CEGIS$.

This sort of enumerative synthesis algorithm though very simple to understand and implement, can quickly become quite computationally expensive and infeasible in practice when we try to generate even decent sized programs.

3 Our Approach

We have defined and implemented an enumerative synthesis procedure called SYNTHEPYZ, so named for our reference implementation that utilizes the Python interpreter, for operation implementations in a compiler for which a reference interpreter is available. Our technique is source-language agnostic, using the preexisting reference interpreter to produce a fixed set of test cases against which x86 programs are sought. This reference-guided partial verification reduces the complexity of the technique.

3.1 Virtual x86

We define *Virtual x86* (or vx86) to be a subset of the x86 language where hardware registers have been replaced with infinitely many virtual, indexed storage locations that behave exactly like hardware registers. The vx86 machine has the following properties:

- Any instruction in vx86 has the same operating semantics as in regular x86.
- Only virtual registers are valid locations (there is no address space, nor any alternative addressing modes)
- No instructions which may alter the behavior of a future instruction are permitted

By limiting x86 in this way, we define a language which allows us to recursively enumerate all possible programs and consider their behaviors incrementally. Formally, a vx86 program has the following grammar:

⟨*program*⟩ ::= ⟨*instruction*⟩ ⟨*program*⟩
  | ⟨*empty*⟩

⟨*instruction*⟩ ::= '`mov`' ⟨*register*⟩ ⟨*register*⟩
  | '`add`' ⟨*register*⟩ ⟨*register*⟩
  | '`mul`' ⟨*register*⟩ ⟨*register*⟩
  | '`neg`' ⟨*register*⟩

⟨*register*⟩ ::= '`r`' ⟨*natural-number*⟩

3.2 Generating Tests

As our synthesis algorithm describes a general method for discovering programs that satisfy a set of constraints, it is therefore also desirable to automate test generation. In our example scenario, we do this by interrogating a reference interpreter using a random distribution of input values to produce a suite of tests which are *probably* comprehensive.

Given a k-ary operator, we will choose tests based on

- all combinations of the numbers one and zero (based the fact that these are the multiplicative and additive identities, respectively), and
- a sample of sequences of k values chosen by the result $f(x) = x^9$ for random values of x, up to a certain threshold

Because the synthesis engine is slow, it is desirable to generate a suite of test cases that covers both the boundary cases around zero and one as well as a probabilistic subset of the larger space of integers in as few tests as possible.

3.3 Synthesis

Our synthesis algorithm is fully enumerative. That is, it begins with an empty program and considers every program that exists by enumerating all valid derivations of new programs from existing programs.

We will say a program is a pair (C, R), where C is a sequence of instructions following the grammar given above, and R is a set of virtual registers in use within C. For synthesis of an arbitrary operator with k operands (i.e. a k-ary operator), we will initialize our synthesis engine with a set of candidate programs $P = (\epsilon, \{r_0, r_1, r_2 \dots r_{k-1}\})$, containing only the null program with zero instructions and as many virtual registers as our operator has inputs. We further initialize a set $V = \{\}$, which is to store the set of program outputs we have seen, to be empty.

3.3.1 Validation

During the validation step, we *execute* each program $c \in P$ by assigning the inputs of each test case to the first k virtual registers and simulating the result of executing each instruction in c, producing a **signature**. The signature of a program is the set of all register assignments resulting from the execution of the program on each test. If the signature S is in V, the set of visited program outputs, then we remove c from P, as it is functionally equivalent to a program we have already considered. If S is not in V, then c will either be validated or extended.

If the signature under consideration, S, contains any register which is *always* assigned to the result of its test case, then we say that c satisfies the specification, and we accept c as the correct solution and the result of the synthesis problem. If c does not satisfy the specification, then we add S to V.

Once we have considered each program in P, and if no program was admitted as the correct result, replace P with P' and continue to the next step, program derivation.

3.3.2 Derivation

We begin derivation by constructing a set of next-generation programs $P' = \{\}$ to be empty. We then consider again, each program c in P, and we produce derivations of c by considering modifications of c according to the following rules:

- **Aliasing** one register to another, by adding a new virtual register r_j to c, and inserting the instruction `mov` r_i r_j to copies of c for each virtual register r_i already in c.
- **Operating** over registers already in c, by considering all instructions other than `mov` and all combinations of registers in c as operands. In other words, try every combination of registers with every possible instruction.

Every candidate derivation produced by the rules above will be added to P'. Then, once we have considered all derivations of all programs and added them to P', we replace P with P' and return to the Validation step, repeating this process until a correct program is found. We refer to each cohort of programs generated by one pass of this derivation algorithm is a **generation**.

4 Implementation & Evaluation

4.1 Implementation

We implemented a proof-of-concept of SYNTHEPYZ in Rust [1]. The implementation follows the algorithm given above, but contains some optimizations. Most notably (from a performance standpoint), we compute the results of each program incrementally (each Program contains a reference to its parent program, its set of registers, and the instruction that makes it unique from its parent), where each program is simulated only as an incremental change from the results of the program that generated it.

4.2 Example

Consider the expression $x^2 + y$. A reasonable set of test cases of the form $(x, y) \to result$ (generated by all combinations of one and zero plus an equal number of random test cases) for this expression might be:

- $(0, 0) \to 0$
- $(0, 1) \to 1$
- $(1, 0) \to 1$
- $(1, 1) \to 2$
- $(-512, 1) \to 262145$

[1] `https://github.com/csci4555-f17/project-synthepyz`

- $(512, -262144) \rightarrow 0$
- $(0, 1953125) \rightarrow 1953125$
- $(-1, 512) \rightarrow 513$

The synthesis engine will first consider the null program with two instructions, as the expression has two operands: $P = (\epsilon, \{r_0, r_1\})$. From this program, it will derive the program $(\texttt{(mul r0 r0)}, \{r0, r1\})$, and from that program it will derive the first correct program: $(\texttt{((mul r0 r0) add r1 r0)}, \{r_0, r_1\})$. The simulation of this correct program results in a signature where the correct result is always in r_0

4.3 Evaluation

For simple examples such as the previous, the correct program is computed virtually instantaneously. The result for the previous example was computed in 0.012 seconds of total time.

We constructed a simple test of the engine's performance in which we attempt to run the algorithm on a set of test cases with no simple solution. Below are the runtimes and overall CPU utilization (where 100% represents the full use of a single thread) for fully considering each generation of programs. We ran each test on an Intel i7 with six cores (twelve logical threads) at 3.8 GHz.

- **Generation 1**: $0.005s$ total time, 228% CPU
- **Generation 2**: $0.006s$ total time, 295% CPU
- **Generation 3**: $0.013s$ total time, 341% CPU
- **Generation 4**: $0.057s$ total time, 294% CPU
- **Generation 5**: $0.652s$ total time, 201% CPU
- **Generation 6**: $10.227s$ total time, 168% CPU

Beyond Generation 6, the run time becomes difficult to measure (as the process is killed due to memory consumption), and the problem becomes intractable.

5 Conclusion

We believe that this technique presents a novel method for synthesizing transformations from a high-level language to a machine language by leveraging an existing interpreter (or any model by which expressions in the high-level language may be automatically evaluated). Our goal is not to eliminate the work of writing a compiler, only to show that the most mundane aspects of compiler design may be automated in some cases. We are interested in how this approach may be integrated with more conventional synthesis/verification approaches involving, for example, SMT or SAT solvers, as our method is fast for small operations though large programs remain intractable. We believe that this method may be useful for constructing the basic building blocks of a compiled language, from which more advanced synthesis techniques may build more comprehensive programs, utilizing the artifacts of our synthesis engine as basic components.

Acknowledgements

We thank Steven Malis (Software Engineer, Microsoft, Inc.) for his inspiration and assistance with the incremental simulation optimization in our proof-of-concept code, and for his permission to use this idea in our implementation.

References

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis, 10 2013.

[2] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.