Component-based Program Synthesis in OCaml

Zhanpeng Liang
University of Southern California
zhanpenl@usc.edu

Kanae Tsushima
National Institute of Informatics, Japan
k_tsushima@nii.ac.jp

1. Introduction

Program synthesis has been a powerful technology that improves our productivity by doing mundane tasks automatically. In this talk, we present a program synthesis tool OCPet for OCaml, targeted at beginners learning the language. It is transplanted from SyPet, a component-based[1] and test-based Java-program synthesizer (Feng et al. 2017) that employs a type-directed approach. Our synthesizer takes in a function signature (function name, input and output types), a set of library functions and a set of test cases and, hopefully, generates a function implementation that passes all specified test cases.

2. Basic Functionality

Based on the method in the original paper, OCPet is able to synthesize straight-line programs, that is, those without any flow control and recursive structure. The synthesized program will make sequential calls to functions in the libraries provided. Thus functionalities combining library functions as building blocks, like string manipulation, are possible to be generated automatically.

3. Example

Given the following information:

```
Signature:
        make_string : int -> char -> string
libraries:
        Char, String, Pervasives
tests:
        make_string 1 'k' = "kk"
        make_string 5 'm' = "mmmmmmmmmm"
```

OCPet gives output:

```
let rec make_string arg0 arg1 =
        let v0 = String.make arg0 arg1 in
        let v1 = (^) v0 v0 in
        v1
```

4. How it works

Synthesis here is basically a search process, which is composed of two phases: 1) path enumeration and 2) parameters filling. In the first phase, OCPet makes use of type information to enumerate proposal code sketches, which are incomplete codes that parameters are not filled for function calls. The collections of variable types are states of the search space, which are encoded in Petri nets and will be described in the next section; the collections of input types and that of the output type respectively define the start and end state; to find a program is to discover paths from the start to the goal state. In the first phrase the synthesizer, implementation of Petri net is provided by SNAKES (Pommereau 2015), a Python library. In the second phase, Z3 (De Moura et al 2008), a SMT solver is used to find all possible substitutions for all the place holders in the sketches. As numerated, candidate codes are tested against all the test cases until a correct answer is found.

5. Abstractions

The following concepts have their own interpretations in Petri net and are illustrated in Figure 1.

Types are represented as *places* that hold tokens. The number of tokens in a place is the count of the variables having that specific type in the current environment. From the perspective of a place, an outgoing edge with weight c indicates that it can be consumed by c tokens via a transition, and an incoming edge implicates tokens that can be produced here.

Functions are represented as *transitions* that walk the net from one state to another by consuming tokens from incoming places (input types for the function) and producing new ones in outgoing places (output types). In Figure 1, transition *Char.escaped* convert a token from type char to type string.

Variables are represented as *tokens* in places. All tokens of a type are identical, so only the number of tokens of each type is captured in states.

Execution context (or environment) after calling some library functions is simplified as states (or markings in the sense of Petri net). In the example, the initial context is translated into the state with one token in the place of *int* and another in the place of *char*.

Overall, in a specified Petri net framework, one can explore paths from state to state, where each step is a function in the library. For the explanation of the example in Sec-

[1] Components are interchangeable with library functions

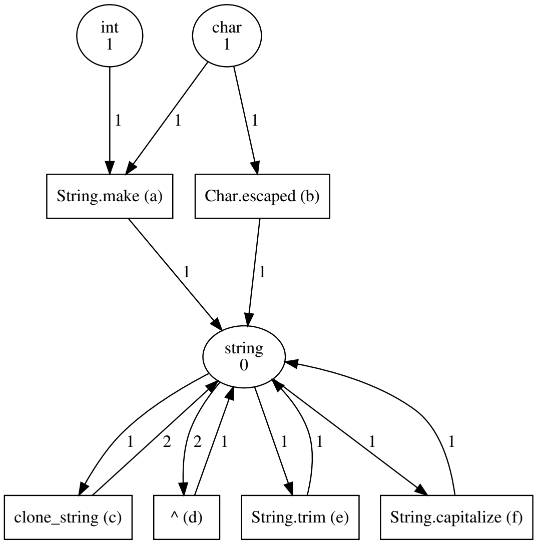

Figure 1: The Petri net with the set of functions used to synthesize the example function in Section 3. Functions (rectangular nodes) connect the three types (elliptical nodes) of char, int and string, in the shown topology. Specifically, the state of the net at this moment is a multi-set having a token of type char and one of type int.

Length	New Discovered States	Paths
0	`{int:1, char:1}`	N/A
1	`{str:1}, {int:1, str:1}`	a
2	`{str:2}, {int:1, str:2}`	`a -> e; a -> f`
3	`None`	`a -> c -> d`

Table 1: An illustration of path enumeration when synthesizing the program in Section 3. Both old (not listed in the table) and new states are explored at each program length and their predecessors are recorded. If the goal state ({str:1}) is met, paths from start state to goal will be listed.

tion 3, we can follow the flow in Table 1, associated with Figure 1. The net starts from a state with an integer and a character; at length 1, goal state {str:1} is discovered so a path containing only one procedure *String.make* is listed; at length 2, the goal is revisited, via two self cycles, so paths *String.make*, *String.trim* and *String.make*, *String.capitalize* are listed; at length 3, the goal is revisited again, and readers can verify that it is reached through the path *String.make*, *clone_string*, ^ *(string concat).*

6. Extended Functionalities

We extend methodology to synthesize codes that has pattern matching as the control flow and also recursive call. For different branches in the synthesizer, different states are specified as starting point in path enumeration (initial state for each branch varies). For recursive calls, a new transition of the target function is simply introduced in the Petri net; and to ensure termination of the synthesized program, constraints are set that only structurally smaller variables are allowed to be passed to recursive function calls. A concrete example is shown below.

```
Signature:
        stutter : int list -> int list
libraries:
        list
tests:
        stutter [] = []
        stutter [1] = [1;1]
        stutter [1;2] = [1;1;2;2]

Output:
let rec stutter arg0 =
match arg0 with
|[] ->
        arg0
|hd::tl ->
        let v0 = stutter tl in
        let v1 = List.cons hd v0 in
        let v2 = List.cons hd v1 in
        v2
```

7. Applications

OCPet itself can be used as a learning tool for beginners: it allows users to program by examples even without (sufficient) knowledge about logic of OCaml or libraries. It can also be used as a successive tool for the type debugger to propose fixes for the code: when a program is detected as having type error, the buggy section of the original code will be found by the debugger and type information in the error can be utilized as input to pass the OCPet, which then suggests code to replace.

8. Conclusion and Future Work

All programs synthesized by OCPet is guaranteed to be type-correct and with less than 6 function calls in reasonable period of time, a program can be synthesized using basic module in OCaml standard libraries.

The synthesizer is limited when it comes to polymorphism since general types in the libraries are only grounded into primitive types (int, bool, char, string); it does not generalize to function types because functions are not, though possible, taken as values to pass to other functions.

In terms of performance, the most significant difficulty the synthesizer faces is the enormous search space because of the volume of library and length of the targeted program. In a library where types are linked by a huge amount of functions, the number of possible programs will explode at a steep rate as the allowed length increases.

Ways to accelerate the synthesizer include providing heuristics like statistics of the use of library functions and intermediate results (divide-and-conquer-like method). Perhaps one of the most effective improvements would be pruning the useless functions to substantially reduce the size of the Petri net.

References

OCPet. `https://github.com/hectorpla/CBS`

Feng, Yu, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps "Component-based synthesis for complex APIs." Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages.

Pommereau, Franck. "SNAKES: A flexible high-level petri nets library (tool paper)." International Conference on Applications and Theory of Petri Nets and Concurrency. Springer International Publishing, 2015.

De Moura, Leonardo, and Nikolaj Bjorner. "Z3: An efficient SMT solver." International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2008.