Simpler Implementation of Sketches Through Enhanced Expressiveness

Miguel Velez[1] and Armando Solar-Lezama[2]

1 University of St. Thomas, St. Paul, MN, 2 Massachusetts Institute of Technology, Cambridge, MA

Massachusetts Institute of Technology

MSRP MIT Summer Research Program

CSAIL Computer-Aided Programming Group

Introduction

The Sketch [6] engine is a tool that allows programmers to write a *sketch*, a partial program with some missing implementation, which the synthesizer uses to discover the missing code. The programmer uses expressions and place holders to indicate what variables and functions should be used to determine the missing code. When *sketches* increase in size, it can be very difficult to track and specify all the variables and functions that the engine should use. We implemented some features that made the language more expressive and the engine more powerful.

Local Variables Construct

Grammar

local_variable : **$(*data_type*);**
data_type : **bit | int | float | double | *user defined object*;**

- Pick local variables within scope and of the specific type

```
void swap(ref bit[W] x, ref bit[W] y){
    minrepeat{
        $(bit[W]) = $(bit[W]) ^ $(bit[W]);
    }
}
```

- Can use regular expression generators [5] to specify what variables should be considered when finding a solution

```
generator Vec Gen(bit domul, Mat a, Mat b, Vec w,
    Vec x, Vec y, Vec z, double d1, double d2){
    Vec tt = {|w|x|y|z|};
    if(??){
        tt = scale(tt, {|d1|d2|});
    }
    if(domul && ??){
        tt = mvmul({|a|b|}, tt);
    }
    if(??){
        tt = vsub({|w|x|y|z|tt|}, {|w|x|y|z|tt|});
    }
    return tt;
}
```

- As sketches grow in size, it can be complicated to track all variables

```
generator Vec Gen(bit domul, Mat a, Mat b, Vec w,
    Vec x, Vec y, Vec z, double d1, double d2){
    Vec tt = $(Vec);
    if(??){
        tt = scale(tt, $(double));
    }
    if(domul && ??){
        tt = mvmul($(Mat), tt);
    }
    if(??){
        tt = vsub($(Vec), $(Vec));
    }
    return tt;
}
```

- Using local variables construct

- Simpler and cleaner method to specify variables needed
- Easier to read and debug

Lambda Expressions

Grammar

lambda_expr : **(*vars*) → *right_expr*;**
vars : ***comma delimited variables***
right_expr : ***any valid expression in the language***

- Supported in many other languages

```
(\x -> x + 1) 4
```
Haskell [2]

```
g = lambda x: x**2
```
Python [3]

- Syntactically lighter than declaring a function with a name, a return type, and formal parameters

```
harness void main() {
    Circle c1 = new Circle {...};
    ...
    assert circ(c1) != circ(c3);
}

float circ(Circle c) {
    return 2 · 3.1415 · c.radius;
}
```

- Need to know that the function *circ* takes a Circle struct as a parameter and that the return type is a float

- Lambda function performs the same operation, but it is simpler to write

```
harness void main() {
    Circle c1 = new Circle {...};
    ...
    fun circ = (c) -> 2 · 3.1415 · c.radius;
    assert circ(c1) != circ(c3);
}
```

Results and Conclusions

- Enhanced expressiveness and increased the domain of applications that programmers can develop with this engine
- Increased productivity and made sketches easier to write and debug
- Local variables construct provides a cleaner method to instruct the synthesizer what variables it should consider when finding an answer
- Lambda expressions are a syntactically lightweight method to create a function that can be reused and passed to high-order functions
- Automatic casting of expressions combines the power of the previous features to avoid implementing specific and complicated functions

Acknowledgements

I would want to thank Professor Armando Solar-Lezama for his valuable advice and guidance, the Computer-Aided Programming group for their support, and the MIT Summer Research Program and the MIT Office of the Dean for Graduate Education for funding this project.

Casting of Expressions

- Allows to pass expressions in high-order function calls

```
int apply(fun f, int x){
    return f(x);
}
```

- Combines the previous features to increase reuse of these functions
- Useful in high-order generators that allow flexibility

```
generator int rec(fun choices){
    if(??){
        return choices();
    }else{
        return {|rec(choices) (+ | - | *) rec(choices)|};
    }
}
```

- To pass an arbitrary number of variables to the generator, the user needs to define a generator and use a regular expression

```
harness void sketch( int x, int y, int z ){
    generator int F(){
        return {| x | y | z |};
    }
    assert rec(F) == (x + x) * (y - z);
}
```

- By combining the features of lambdas and local variables construct, we do not need to declare a generator with choices

```
harness void sketch( int x, int y, int z ){
    assert rec($(int)) == (x + x) * (y - z);
}
```

Future Work

- Currying, which allows easier creation of anonymous functions
- Built-in functions that take lambdas expression and modify arrays
- Local variable construct automatically infer the type of variable
- Optimize and develop systems that benefit from these features
- Systems include, but are not limited to, an automatic grader for programming assignments [4], a system that automatically transforms fragments of application logic into SQL queries [1].

References

[1] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. PLDI, 2013.
[2] Haskell Anonymous function. https://wiki.haskell.org/Anonymous_function.
[3] Python Expressions. https://docs.python.org/2/reference/expressions.html#lambda.
[4] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated semantic grading of programs. PLDI, 2013.
[5] Sketch. http://people.csail.mit.edu/asolar/manual.pdf.
[6] A. Solar-Lezama. Program Synthesis By Sketching. PhD thesis, EECS Dept., UC Berkeley, 2008.