Homework: Sketching

The goal of this assignment is to build the essence of a program synthesizer, in the style of Sketch. Before you proceed with the assignment, you need to be intimately familiar with the following paper:
Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit A. Seshia. Combinatorial Sketching for Finite Programs. In proceedings of *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* 2006.

The language of this assignment will not support arrays, so you can ignore Section 5.3.1 of the paper.

1 Setup

To do this assignment, you will need a copy of the Z3 Theorem Prover:

`https://github.com/Z3Prover/z3/releases`

The packages on the website include Z3 bindings for several languages. However, you will only need the `z3` executable. (In particular, you do not need the OCaml bindings for Z3.)

2 Requirements

Write a program synthesizer that takes three arguments:

`./synth.d.byte` *w sketch* `spec`

The arguments *sketch* and `spec` should refer to files that contain programs in the grammar defined in fig. 27.1 (the spec should not contain any holes (`??`). You should assume that all values and variables in the program are bit-vectors of size w.

If the synthesizer succeeds, it should print a copy of `sketch` where holes are replaced with concrete values to standard out. (Feel free to print any debugging output to standard error.) If the synthesizer fails, it should exit with code 1.

3 Support Code

The module `Sketching` defines the abstract syntax of the language shown in fig. 27.1 and includes functions to parse and pretty-print programs in the language. The module `Smtlib` provides an API for communicating with Z3.

4 Directions

We suggest proceeding in the following way:

1. Start by ignoring loops and first get the synthesizer to work on straight-line code.
2. You'll need two instances of Z3: one for the synthesizer and one for the verifier. Most of the functions you write will invoked for either solver, so pass the solver as an argument to all functions.
3. Use a global variable to store the width of bit-vectors.
4. Implement the main CEGIS loop, which will require several helper functions.
5. Once your verifier works for loop-free programs, you should tackle loops as follows.
    (a) Assume that loops in the bounded by a constant and fully unfold all loops in the specification.

Expressions

exp	::=	n	`EInt n`	where n is a decimal integer
	|	x	`EId x`	where x has letters, digits, and underscores
	|	exp_1`+`exp_2	`EOp2(Add, exp1, exp2)`	
	|	exp_1`-`exp_2	`EOp(Sub, exp1, exp2)`	
	|	exp_1`*`exp_2	`EOp2(Mul, exp1, exp2)`	
	|	exp_1`/`exp_2	`EOp2(Div, exp1, exp2)`	
	|	exp_1`%`exp_2	`EOp2(Mod, exp1, exp2)`	
	|	exp_1`<<`exp_2	`EOp2(LShift, exp1, exp2)`	
	|	exp_1`>>`exp_2	`EOp2(RShift, exp1, exp2)`	
	|	exp_1`&`exp_2	`EOp2(BAnd, exp1, exp2)`	
	|	exp_1`||`exp_2	`EOp2(BOr, exp1, exp2)`	
	|	exp_1`and`exp_2	`EOp2(LAnd, exp1, exp2)`	1 if both arguments are non-zero and 0 otherwise
	|	exp_1`or`exp_2	`EOp2(LOr, exp1, exp2)`	1 if either argument is non-zero and 0 otherwise
	|	exp_1`==`exp_2	`EOp2(Eq, exp1, exp2)`	
	|	`not`*exp*	`EOp1(LNot, exp)`	
	|	`~`*exp*	`EOp1(BNot, exp)`	
	|	`W`	`EWidth`	represents the width of bit-vectors
	|	`??`	`EHole n`	where n uniquely identifiers this hole
	|	`(`*exp*`)`	`exp`	

Programs

cmd	::=	`skip;`	`CSkip`	
	|	`abort;`	`CAbort`	
	|	x `=` *exp*`;`	`CAssign (x, exp)`	
	|	`if (`*exp*`)` cmd_1 `else` cmd_2	`CIf (bexp, cmd1, cmd2)`	
	|	`repeat` x `:` *exp cmd*	`CRepeat (x, exp, cmd)`	Evaluates *cmd exp* times, with x set to $0, 1, \cdots,$ `exp` $- 1$.
	|	`{` *cmd* `}`	`cmd`	
	|	`assert` *bexp*`;`	`CIf (bexp, CSkip, CAbort)`	
	|	cmd_1 cmd_2	`CSeq (cmd1, cmd2)`	

Figure 27.1: The concrete syntax and abstract syntax of the language.

(b) In the sketch, loops that are not bounded by a constant need to be unfolded one iteration at a time. You may find OCaml's `Stream`s to be useful for implementing this feature.

5 Debugging

Debugging this assignment can be challenging, since a significant portion of the work is being done by Z3. You will almost certainly have to inspect commands and output of Z3 to debug your synthesizer. The support code doesn't allow you to directly examine its interaction with Z3. However, instead of supplying the actual path to Z3 to `make_solver`, you can use the following shell script to log Z3 commands and responses:

```
tee -a z3-commands.txt | z3 -in -smt2 | tee -a z3-responses.txt
```

The file `z3-commands.txt` is a valid Z3 script. Alternatively, to see commands and responses interleaved:

```
tee -a interaction.txt | z3 -in -smt2 | tee -a interaction.txt
```

Since you need to create two solvers, it may help to have two different scripts to log the interaction with Z3.

Finally, the Z3 interactive console is quite impoverished and doesn't support history, arrow keys, etc. I recommend using "rlwrap" to wrap Z3 before you try to use the console directly:

https://github.com/hanslub42/rlwrap

Rlwrap is available on standard Linux package repositories and via Homebrew for Mac OS X.