Lab Report – Z3 & Program Analysis Using Constraints
Joe Emenaker
CSC 530 – Spring 2021

Preface: When someone in the class asked about deliverables for this assignment, the most concrete thing I recall you saying was that you wanted us to see what we could figure out with Z3 and then present a "lab report" of our findings. So this is what I discovered.

Explorations with Verification Criteria

My original goal was to see if I could duplicate the invariant from the paper by deriving all of the coefficient-based criteria from the verification constraints. As you cautioned, this was quite ambitious.

What I did do was convert the third verification constraint from part 1c in the paper to some equalities constraining the coefficients of proposed polynomial used to represent the invariant (which I expect to include at the end of this document). However, I had to *weaken* some of the constraints in order to apply Farkas' lemma. Since we ultimately need to get our criteria into a form of

$$\neg[\, (e_1 \geq 0) \land (e_2 \geq 0) \land (e_3 \geq 0) \ldots \,]$$

We can use deMorgan's laws to remove the outer negation:

$$(e'_1 \geq 0) \lor (e'_2 \geq 0) \lor (e'_3 \geq 0) \ldots$$

Where e'_i is related to e_i in a way that preserves the inequalities (and these are shown in my derivation). Therefore, when representing our verification criteria, we want to aim for *disjuntive* normal form. However, I ended up at a point where I had a solitary conjunction, like this:

$$(e'_1 \geq 0) \land (e'_2 \geq 0) \lor (e'_3 \geq 0) \lor (e'_4 \geq 0) \ldots$$

And my only solution was to add those two inequalities:

$$(e'_1 + e'_2 \geq 0) \lor (e'_3 \geq 0) \lor (e'_4 \geq 0) \ldots$$

This, of course, *weakens* what we can deduce about the coefficients. There might be another way to eliminate the conjunction without resorting to addition, but I couldn't figure it out.

Explorations with Z3

Good lord... and I thought ML was terse. I've never seen if-then-else reduced to "(ite <cond> <expr> <expr>)"

The first choice was whether to use Z3, natively, or the Python library. After looking at some Python examples, I chose to just go with plain Z3. The Python examples look a little spooky, since they seem to overload (or delay evaluation of) some comparison operators. For example:

```
# 9x9 matrix of integer variables
X = [ [ Int("x_%s_%s" % (i+1, j+1)) for j in range(9) ]
      for i in range(9) ]
```

```
    # each cell contains a value in {1, ..., 9}
    cells_c  = [ And(1 <= X[i][j], X[i][j] <= 9)
                 for i in range(9) for j in range(9) ]
```

The first list comprehension is using "`i+1`" and "`j+1`" normally to create an array of "Int" Z3 objects. The *second* comprehension is passing the "<=" and "<=" operators *into Z3* (i.e. they're not being evaluated by Python in the call to And(). Lacking a solid understanding of Z3, I decided that this would be too confusing to have strange behavior like this, so I decided to go with straight Z3.

Unfortunately, there seem to be many more references for the Python z3-solver library than for "z3" when you search the web. Any searches for syntax help (like "z3 length list") yield a hodgepodge of stackoverflow posts, the rise4fun.com site, or the stanford.edu site. The later two, although useful tutorials, don't seem to be a comprehensive reference for syntax and built-in functions/constants.

Then, I discovered that Z3 is just using a more-widely used smt library, and I should be searching for "smt-lib" instead of "z3". I found a lot more *reference* material (as opposed to tutorials) like a grammar for the syntax: http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r10.12.21.pdf As a bonus, VSCode has a smt-lib syntax highlighter, but nothing for "z3".

Motivating Example

I usually need some objective that I'm trying to accomplish in order to motivate me to explore a language or framework, otherwise, I don't force myself to push through places where I get stuck. I could have just tried some systems of inequalities, but it would have been more work to devise/verify interesting problems than it would be to actually put them into Z3 syntax. So, what I decided to try was making something which could produce possible goal solutions in the game "Mastermind". In case you've never played it, here's a description: https://www.youtube.com/watch?v=5jtcsBERDEQ

The general approach was to create a 'possible_solution' constant and a GuessResult function which would take a guess and the feedback (number of pegs in proper position and number of correct *color* but wrong *position*) and return a Boolean reflecting whether the guess and possible_solution could yield the given feedback. By asserting that all of the calls to GuessResult for all guesses are *true*, this constrains what values *possible_solution* could take.

As it turns out, coding the logic to support the notion of "correct color but wrong position" got really tough, so I simplified the problem so that the *only* feedback is the number of pegs that are *both* correct color and correct position. *That* was fairly straightforward to code, and I was able to try one guess to get a proposed solution, then introduce that as a new guess (with the appropriate feedback) to get a new proposed solution. To my pleasant surprise, the solution started converging on the hidden solution. Unfortunately, when it was *just* about to solve it, Z3 gave me an "unsat", so I may have messed up some of my feedback on the guesses.

Revisiting Program Verification

After learning quite a bit about using Z3 in the Mastermind project, I decided to turn my attention back to program verification. Specifically, I wanted to see if it would be possible to *skip* all of the derivation and manipulation of our criteria, and just hand the criteria (as they appeared in section 1c of the paper) directly to Z3. It turns out that this looks like it might be possible.

Because Z3 allows for a "forall" directive, I was able to create statements like:

```
; 1st Verification Constraint
(assert (forall ((y Int))
  (Invar -50 y)
))
```

Where we require that, for all y, Invar (a function which takes (x,y) and returns (a1x + a2y + a3 >= 0) || (a4x + a5y + a6 >= 0)), when passed (-50, y), must be true.

When I tried asserting all three criteria from 1c, it gave me an "unsat". *However*, when I gave it just the 1st and 3rd criteria, Z3 generated coefficients for a1-a6 which translated to the following invariant:

Invar = (-x + 2y >= 1) OR (-x >= 1)

That last condition can be rewritten as (x < 0), *which is one half of the invariant from the paper*!
The first condition can be rewritten as (y > x/2), which looks to be a weaker version of the other criteria from the paper. So, I suspect that this technique (of avoiding having to rewrite a lot of the criteria and employ Farkas' lemma, etc) might be workable.

Specific Interesting Things Learned

Here are a few interesting quirks of Z3 that I encountered which would have been nice to know ahead of time.

Most of the namespace seems to be flat

It's a little interesting that even values which seem 'internal' to datatypes are put directly into the top-level namespace (in fact, it seems that top-level is the *only* namespace). This means you can't do things like:

```
(declare-datatype Colors ( orange blue green ))
(declare-datatype Fruits ( orange apple pear ))
```

as we'll get a collision on 'orange'. This means we have to be careful about prefixing disambiguators to our names, eg.

```
(declare-datatype Colors ( col_orange col_blue col_green))
```

Total functions are expected, yet non-trivial

Z3 expects that all functions are total, *yet it doesn't crash if they aren't*. It just seems to ignore any assertion involving an undefined value. Maybe there's a flag to change this behavior, but I didn't find it. This makes it a little tricky to craft functions which don't have unhandled corner cases (since those cases just result in assertions not being made, which is much harder to detect than a crash or exception)

With that in mind, I tried to avoid combining too many criteria in individual if-then-else guards to prevent developing gaps. For example

```
(ite
   p  ; if p...
   (ite
      (not q)  ; if not q...
   )
)
```

is probably preferable to:

```
(ite
   (and p (not q))
   ...something...
```

```
            <- What do we do here for all cases where p is false or q is true?
         )
```

Enumerated types and bounds are your friends when using quantifiers

To avoid Z3 having to go through, say, all integers, when possible, resort to enumerated types (or put bounds on infinite types). For example, with Mastermind, rather than use an integer to represent the position of a peg, we could use:

```
(declare-datatype Position ( p0 p1 p2 p3 ))
```

Or, we could use bounds like:

```
(declare-const position Int)
(assert (>= position 0))
(assert (<= position 3))
```

As these are likely to interrupt any runaway forall/exists pursuits by the solver.

Type-inference isn’t fully fleshed out, yet

I ran into a few cases where Z3 reported that some of my sort references were ambiguous. Specifically, this happened when I was using *nil* to build a List:

```
(declare-const my_real_list (List Real))
(declare-const my_int_list (List Int))
(assert (= my_int_list (insert 2 (insert 1 nil))))
```

Because there are two different parameterized sorts of List in-scope, there are two different *nil* constants, and Z3 doesn’t know which one to use. So, you must “cast” or disambiguate the nil with:

```
(declare-const my_real_list (List Real))
(declare-const my_int_list (List Int))
(assert (= my_int_list (insert 2 (insert 1 (as nil (List Int))))))
```

What’s odd, however, is that we don’t have to disambiguate *insert* in the same way; Z3 is able to infer which one we want (possibly from the fact that we’re feeding it a List Int. Yet, how come Z3 can’t infer the type of *nil* from the fact that I’m equating it with my_int_list, a known List Int?

Example Derivation of Existential Form of Invariant Constraints

Derivation of third VC in figure 1(c) to Prop logic

$I \wedge x \geq 0 \Rightarrow y > 0$ — Remove implication

$\neg(I \wedge x \geq 0) \vee y > 0$ — deMorgan's law

$\neg I \vee \neg(x \geq 0) \vee y > 0$

Assuming $I = (a_1 x + a_2 y + a_3 \geq 0 \vee a_4 x + a_5 y + a_6 \geq 0)$

$\neg I = -1 - a_1 x - a_2 y - a_3 \geq 0 \wedge -1 - a_4 x - a_5 y - a_6 \geq 0)$

Inequality Conversions for integers

$\neg(e \geq 0) \Leftrightarrow e < 0 \Leftrightarrow -1 - e \geq 0$

$\neg(e > 0) \Leftrightarrow e \leq 0 \Leftrightarrow -e \geq 0$

$\neg(e \leq 0) \Leftrightarrow e > 0 \Leftrightarrow e - 1 \geq 0$

$\neg(e < 0) \Leftrightarrow e \geq 0$

$e_1 \geq 0 \wedge e_2 \geq 0 \Rightarrow e_1 + e_2 \geq 0$
(with some loss of specificity)

To use Farkas' lemma, we need to have our expression in the form of $\neg(\bigwedge_i e_i \geq 0)$

The leading negation suggests that we can use deMorgan's law on a DNF form $\neg\neg(\bigvee_i e_i' \geq 0)$ where $(e_i' \geq 0) \Leftrightarrow \neg(e_i \geq 0)$. So, we work toward a DNF form:

$$-1 - a_1 x - a_2 y - a_3 \geq \underbrace{0 \wedge -1}_{} - a_4 x - a_5 y - a_6 \geq 0 \vee \neg(x \geq 0) \vee y > 0$$

to eliminate this, we can add the two expressions

$$-2 - (a_1 + a_4) x - (a_2 + a_5) y - (a_3 + a_6) \geq 0 \vee \neg(x \geq 0) \vee y > 0$$

Add two negations & push one inward

$$\neg\neg(-2 - (a_1 + a_4) x - (a_2 + a_5) y - (a_3 + a_6) \geq 0 \vee \neg(x \geq 0) \vee y > 0)$$

$$\neg(\neg(-2 - (a_1 + a_4) x - (a_2 + a_5) y - (a_3 + a_6) \geq 0) \wedge \neg\neg(x \geq 0) \wedge \neg(y > 0))$$

Apply inequality conversions

$$\neg(-1 + 2 + (a_1 + a_4) x + (a_2 + a_5) y + (a_3 + a_6) \geq 0 \wedge x \geq 0 \wedge -y \geq 0)$$

Apply Farkas' lemma

$$\exists \lambda > 0, \lambda_i \geq 0 \left[\forall_{x,y} \left(\lambda_1 (1 + (a_1 + a_4) x + (a_2 + a_5) y + (a_3 + a_6)) + \lambda_2 x + \lambda_3 (-y) = \lambda \right) \right]$$

Because the part inside the universal quantifier must be constant for any x and any y, their respective coefficients must sum to zero (i.e. $\frac{d}{dx} = \frac{d}{dy} = 0$).

$$\lambda_1 (a_1 + a_4) + \lambda_2 = 0 \qquad \lambda_1 (a_2 + a_5) - \lambda_3 = 0$$