Z3 Lab exercises

Leonardo de Moura and Nikolaj Bjørner
Microsoft Research, One Microsoft Way, Redmond, WA, 98074, USA
{leonardo, nbjorner}@microsoft.com

June 18, 2008

Chapter 1

Introduction

This document contains a few hands-on experiments with Z3. We use `F#` and `C#` several places for writing code and pseudo-code for illustrating the use of the managed binary API.

1.1 Acknowledgments

We wish to thank Utkarsh Upadhyay, Wolfgang Grieskamp and Ruzica Piskac for invaluable feedback on preparation of these exercises.

Chapter 2

SAT encodings

This chapter contains exercises for using SAT and bit-vector encodings into Z3. You can use bit-vectors directly to save energy for encoding common idioms that can otherwise directly be encoded in SAT.

2.1 n-queens

The classical n-queens puzzle is to place n queens on an $n \times n$ chess-board so that no two queens attack each other. The exercise asks you to encode an n queens placement problem and then have Z3 enumerate solutions. The following text contains a guided walk through of how this can be accomplished. You can try your own encoding ideas as well.

2.1.1 Placing queens on a chess-board

1. We will allocate an n bit bit-vector per row. So let $r_1, \ldots, r_n$ be n n-bit bit-vectors. In SMT-LIB the declaration for $r_1, \ldots, r_8$ looks as follows:

```
:extrafuns ((r1 BitVec[8]) (r2 BitVec[8]) (r3 BitVec[8])
            (r4 BitVec[8]) (r5 BitVec[8]) (r6 BitVec[8])
            (r7 BitVec[8]) (r8 BitVec[8]))
```

2. Each vector r_i should have at most one bit set. There are various ways of encoding this. The simplest is to create $\binom{n+1}{2}$ axioms per row of the chessboard. For example, for row r_1 one can assert the axioms:

$$r_1[7] \rightarrow \neg r_1[6],\ r_1[7] \rightarrow \neg r_1[5],\ \ldots$$
$$r_1[7] \rightarrow \neg r_1[0], \ldots$$
$$r_1[6] \rightarrow \neg r_1[5],\ r_1[6] \rightarrow \neg r_1[4],\ \ldots$$
$$r_1[6] \rightarrow \neg r_1[0], \ldots$$

We use the notation $r_1[7]$ for accessing the most significant bit of r_1. The other bits are at positions 0 to 6. This is not the most succinct way, however. Consider the formula

$$bv0[8] = (r_1 \& (r_1 - 1))$$

where $bv0[8]$ is a bit-vector of length 8 consisting of all zeros. It says that taking the bit-wise and of r_1 and $r_1 - 1$ results in 0. The arithmetical circuit for this formula is much smaller.

3. Now consider the columns; for each column k there should be exactly one row r_i, such that $r_i[k]$ is set. How would you encode this using as few constraints as possible?

4. Finally consider the diagonals. Also at most one bit should be set on diagonals. How can you express succinctly that in each diagonal there is at most one bit set?

5. Write a program that takes a number n as input and generates a problem in SMT-LIB format for the n-queens placement.

6. Write a program that takes a number n as input, uses the Z3 API to enumerate placements of queens.

2.1.2 Enumerating solutions

This section applies if you are using the programmatic API. Suppose we wish to enumerate several solutions for n-queens. We would need to *block* previous solutions when resuming search. For this purpose Z3 exposes *models* that assigns values to variables. One can take the values and construct new formulas to block with.

With the managed API, the relevant calls are:

- `LBool result = z3.CheckAndGetModel(ref model);` If the current context is satisfiable, then the result is `LBool.True`. The reference argument `model` is populated with an object containing the satifiable assignment.
- `int r1_val = model.GetNumeralValueInt(model.Eval(`r_1`))` retrieves the numeric value assigned to rown r_1.
- `TermAst eq1 = z3.MkEq(z3.MkNumeral(r1_val, row_type),` r_1`);` is the equation stating that r_1 has value `r1_val`.
- `z3.AssertCnstr(z3.MkNot(z3.MkAnd(assignment)));` . If `assignment` is an array containing the current assignment to the variables as equalities, then the current assignment gets blocked by asserting the negation of these.

2.1.3 Symmetry reduction

We might not care about enumerating solutions that are symmetric. That is, we don't need to enumerate solutions that can be obtained by turning the chess-board around.

A *symmetry breaking predicate* is an additional constraint that restricts the search space by ruling out symmetric alternative solutions.

1. Find a symmetry breaking predicate that reduces the placements on the first row.

2. Can you think of other symmetry breaking predicates for n-queens?

2.2 Longest path

The shortest path between two nodes in a graph is a classical graph algorithm problem. Dijkstra's algorithm uses a heap to solve the problem in $O(n \log n)$; and when edge weights are integers, there are even more efficient solutions. To find a longest path between two nodes in a directed graph is on the other hand an NP complete problem. In this exercise we will convert longest path problems into a Boolean constraint satisfaction problem.

2.2.1 Primes and Grey codes

We first need a graph. Of course there are many graphs to pick from. The graph we use here is constructed in a peculiar way. Let V be the set of primes between 1000 and 10000. We connect two nodes in $v, w \in V$, such that $v < w$, iff the decimal representation of v and w has one digit differing by only one.

A sample program for generating the graph, courtesy of Utkarsh Upadhyay, is provided below.

```
let rec sieve (arr:int array ref) step idx=
    if idx>=(!arr).Length then () else
       ((!arr).[idx] <- 0; sieve arr step (step+idx))

let main from till =
    let numbers = ref (Array.init (till+1) (fun i -> i)) in
    (!numbers).[1] <- 0;
    for ii = 2 to int(ceil(sqrt(float(till)))) do
        if (!numbers).[ii]>0 then sieve numbers ii (2*ii);
    done;
    Array.filter
      (fun v -> v>0)
        (Array.sub (!numbers) from (till-from+1))

let get_pairs from till =
    let primes = main from till in
    let rec grey diff =
        if    diff=0 then false    // Same number.
        elif  diff=1 then true
        elif  diff%10 <> 0 then false
        else  diff / 10 |> grey
    in
    let pairs = ref [] in
    for i = 0 to (primes.Length-1) do
        for j = 0 to (primes.Length-1) do
            if (grey (abs(primes.[j]-primes.[i]))) then
```

```
                pairs := (i, j)::(!pairs);
        done;
    done;
   (primes, !pairs)

let prime_index, prime_conn = get_pairs 1000 10000
```

2.2.2 Encoding the graph

The next task we have to face is how one can encode a graph and a path finding problem. So far we have a graph $\mathcal{G} = (V, E)$, where V is a set of n verticies, and E is a set of m edges. We now sketch one possible encoding of the problem.

Suppose the set of vertices V is $\{v_1, \ldots, v_n\}$, then associate n bit-vectors $ord_1, \ldots, ord_n$, each of $\lceil \log n \rceil$ bits (the nearest natural number that is greater than or equal to $\log n$). We will refer to these bit-vectors as *ordinals*. The purpose of the ordinals is to guess an ordering of each vertex, a prefix of the ordering will correspond to a path in the graph. This can be encoded by asserting for each vertex v_i, and for each number $k = 1, \ldots, n$:

$$((ord_i \simeq k) \wedge (k < max\_path)) \rightarrow \bigvee_{v_j \in V} ((v_i, v_j) \in E \ \wedge \ ord_j = k + 1)$$

The bit-vector max_path is the length of the maximal path in $\mathcal{G}$.

We also require that some vertex has ordinal 0:

$$\bigvee_{v_i \in V} ord_i \simeq 0$$

There is a problem with the encoding.

- What is the practical problem, assuming there are around 1064 vertices?
- Find a compact encoding.
- Write a program that converts the graph `prime_index`, `prime_conn` into a constraint for Z3.
- Write a wrapper around `CheckAndGetModel` to find the maximal value of max_path.

2.3 Sudoku

The popular Sudoku puzzle is to place numbers 1-9 on a 9×9 board, such that each number occurs only once in every row and column. Furthermore, if you dividde the board into 9 sub-grids each of size 3×3, then each of these sub-grids should also be covered by different numbers. Sudoku puzzles come with boards that have been partially occupied with numbers. The puzzle is to occupy the remaining fields in such a way that the constraints on rows, columns, and sub-grids are satisfied. A sample Sudoku problem is given in Figure 2.1.

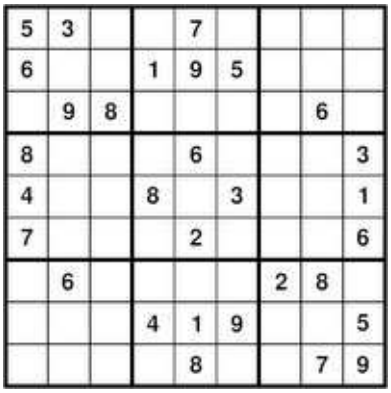

Figure 2.1: Sample Sudoku board

1. Assume that Sudoku problems are given in the form:

```
..1.2....
..9.63...
3....814.
.9....83.
..4.7.6..
.37....2.
.159....2
...48.5..
....1.3..
```

2. Write a program that converts Sudoku problems into:
   (a) an SMT formula using Bit-vectors,
   (b) an SMT formula using integers,
   (c) calls over the Z3 binary API, and use it to enumerate solutions.
3. Test the program on your favorite puzzle.

Note: If you don't really want to do this exercise, you can find a solution and a set of puzzles on `http://modante.googlepages.com/sudokusolver`.

Chapter 3

Integer difference logic

3.1 Rush Hour

Rush Hour is a popular sliding puzzle board game. It was invented by Nob Yoshigahara in the late 1970s. It is sold in the US by ThinkFun since 1996. The problem is to move a car out of a traffic jam. Cars and trucks can only be moved back or forth, they cannot turn or fly. The Rush Hour problem is PSPACE complete [1].

This exercise asks you to encode the rush-hour problem using difference logic and bounded model checking.

3.1.1 Bounded model checking

Let $\rho_1, \ldots, \rho_n$ be a set of transitions. That is they are binary relations over current and next-state variables $\overline{x}$ and $\overline{x}'$. Let Θ be a formula that summarizes an initial state. We can check whether a predicate $p(\overline{x})$ can be satisfied in k steps by checking satisfiability of the formula:

$$\Theta \wedge \bigwedge_{j=1}^{k-1} \left(p(\overline{x}_j) \wedge \overline{x}_j = \overline{x}_{j+1} \wedge \bigvee_{i=1}^{n} \rho_i(\overline{x}_j, \overline{x}_{j+1}) \right) \wedge p(\overline{x}_k) \tag{3.1}$$

That is, $p(\overline{x}_1)$ holds in Θ, or some transition is taken to reach state $\overline{x}_2$. Then p eventually holds in state k.

3.1.2 The exercise

- How would you encode a Rush Hour puzzle on an $N \times N$ board with a set of cars and trucks $c_1, \ldots, c_k$, each having length $\ell_1, \ldots, \ell_k$ and orientation $o_1, \ldots, o_k$ (vertical or horizontal), and row or column affinity $a_1, \ldots, a_n$ (where $a_i \in [1..N]$).
- Encode the bounded reachability problem using Z3.

- Use it to solve Rush Hour problems. You can find a few from `http://www.puzzles.com/products/rushhour.htm`.

Chapter 4

Arrays

4.1 Encoding queues with arrays

Suppose we have support for the theory of arrays. Z3 does in fact have support for arrays. If you specify an SMT-LIB file in the theory `QF_AUFLIA` you will be able to refer to functions `select` and `store`.

$$\forall a, i, v \;.\; store(a, i, v)[i] \simeq v \tag{4.1}$$

$$\forall a, i, j, v \;.\; i \simeq j \;\vee\; store(a, i, v)[j] \simeq a[j] \tag{4.2}$$

$$\forall a, b \;.\; (\forall j \;.\; a[j] \simeq b[j]) \;\rightarrow\; a \simeq b \tag{4.3}$$

We can encode queues using arrays by associating with each queue a triple

$$\langle a, hd, tl \rangle$$

where a is an array holding the queue elements, hd is a pointer to the front element in the queue, and tl is a pointer to the end of the queue.

1. Axiomatize the following queue operations in SMT-LIB.
   (a) $append(\langle a, hd, tl \rangle, e)$ - append element e to the queue $\langle a, hd, tl \rangle$, return the resulting queue.
   (b) $head(\langle a, hd, tl \rangle)$ - extract the head of the queue.
   (c) $tail(\langle a, hd, tl \rangle)$ - return queue where the head has been removed.
   (d) $empty(\langle a, hd, tl \rangle)$ - test if the queue is empty.
2. Prove using Z3 $\neg empty(q) \rightarrow head(q) = head(append(q, e))$
3. Add an operation $prepend(e, \langle a, hd, tl \rangle)$ that pre-pends a value to a queue.
4. Why can't you prove:

$$prepend(e, empty) = append(empty, e)?$$

Chapter 5

Soft-coding theories

5.1 Recursive data-types

The theory of lists is not built into Z3 v1.3. If you want to use it, then you need to supply a sufficient number of additional axioms. The following exercise goes over the steps required to axiomatize a theory of integer lists. Furthermore, if you want to extract meaningful models from the solver, then you need to maintain additional book-keeping.

5.1.1 Axioms for a theory of lists

We will be working with a standard theory of lists axiomatized below.

$$\forall x, \ell \;.\; nil \neq cons(x, \ell) \tag{5.1}$$
$$\forall x, \ell \;.\; hd(cons(x, \ell)) = x \tag{5.2}$$
$$\forall x, \ell \;.\; tl(cons(x, \ell)) = \ell \tag{5.3}$$
$$\forall \ell \;.\; \ell = nil \;\vee\; \exists x, \ell' \;.\; \ell = cons(x, \ell') \tag{5.4}$$
$$\forall \ell \neg \exists n, x_1, \ldots, x_n \;.\; \ell = cons(x_1, cons(x_2, \ldots, cons(x_n, \ell))) \tag{5.5}$$

1. The first (5.1) axiom says that elements produced by different constructors are different.
2. The second and third axiom (5.2), (5.3) says the constructor $cons$ is injective. Injectivity is implied because if $cons(x, \ell) = cons(x', \ell')$, then the two axioms imply that $x = x'$ and $\ell = \ell'$.
3. The fourth axiom (5.4) is domain closure. Every variable ranging over lists is obtained by applying nil or $cons$.
4. The last axiom (5.5) is the *occurs* check. A recursive list cannot contain itself.

5.1.2 Lists of integers in Z3

As shown below, we can declare an abstract type `list`, the constant `nil` and constructor `cons`:

```
let int_type = z3.MkIntType()
let list_type = z3.MkType("list")
let nil = z3.MkConst("nil",list_type)
let cons = z3.MkFuncDecl("cons", int_type, list_type, list_type)
let mk_cons x l = z3.MkApp(cons, x, l)
```

5.1.3 Programmatic axiomatization in Z3

Axiom (5.1)

We could state the axiom directly as given.

```
let x = z3.MkBound(1ul, int_type)
let l = z3.MkBound(0ul, list_type)
let types = [| int_type; list_type |]        // types of bound variables
let names = [| "x"; "l" |]                   // names of bound variables
let patterns = [| z3.MkPattern [|mk_cons x l|] |]
do assert_cnstr (z3.MkForall(0ul, patterns, types, names,
     z3.MkNot(z3.MkEq(nil, mk_cons x l)))) // body
```

We used the auxiliary function `assert_cnstr` to assert and print a constraint. The auxiliary functions `assert_cstr` and `prove` are provided below.

```
let assert_cnstr (fml:TermAst) =
  Console.WriteLine("Assert: {0}", fml);
  z3.AssertCnstr(fml)

let prove (fml:TermAst) =
  z3.Push();
  Console.WriteLine("Prove:  {0}", fml);
  z3.AssertCnstr(z3.MkNot fml);
  assert (z3.Check() = LBool.False);
  z3.Pop()
```

The console output generated from the first axiom is:

```
Assert: (forall (x int) (l list) { ((cons x l)) } (not (= nil (cons x l))))
```

This approach does not scale too well for data-types with multiple constructors. Instead we could use a trick by introducing a *representation* function and obtain a linear number of axioms:

```
let rep = z3.MkFuncDecl("Rep", list_type, int_type)
let mk_rep x = z3.MkApp(rep, (x:TermAst))
let num0 = z3.MkNumeral(0,int_type)
let num1 = z3.MkNumeral(1,int_type)
```

```
do assert_cnstr(z3.MkEq(mk_rep nil, num0))
do assert_cnstr(z3.MkForall(0ul, patterns, types, names,
               z3.MkEq(mk_rep (mk_cons x l), num1))) // body
```

This time we generate the console output:

```
Assert: (= (Rep nil) 0)
Assert: (forall (x int) (l list) { ((cons x l)) } (= (Rep (cons x l)) 1))
```

Axiomatization in the Simplify format

Those familiar with the Simplify format, will recognize the above axioms in the form:

```
(BG_PUSH (FORALL (x y) (PATS (cons x y)) (NOT (EQ nil (cons x y)))))
(BG_PUSH (EQ (rep nil) 0))
(BG_PUSH (FORALL (x y) (PATS (cons x y)) (EQ (rep (cons x y)) 1)))
```

Axioms (5.2) (5.3)

- Exercise: State the axioms for injectivity.

Domain closure (5.4)

Exercise:

- Encode the domain closure axiom.
- What pattern should be used for it?
- What is the problem with the pattern, assuming you identified one?

As you should have observed, the domain closure axiom is not directly amenable to an axiomatization using pattern-based triggers.

We can encode the domain closure axiom by using the following observation. Let $t_1, \ldots, t_n$ be the terms of type `int_list` in the ground formula φ we wish to check. For each of the terms add the axiom

$$t_i = nil \ \vee \ t_i = cons(x_i, \ell_i)$$

where x_i and ℓ_i are fresh variables (we removed the existential quantifier).

Exercise

- Write a function to traverse terms and enumerate terms of type `list_type`.
- Write a function to instantiate the domain closure axioms with the resulting terms.
- Which resulting sub-terms may not be covered by domain closure at this point?
- Why is this not harmful?

Occurs check (5.5)

There are infinitely many instances of the occurs check axiom schema. It is of course impractical to enumerate them all. So which axioms do we need?

There are two possible approaches:

Approach 1 - using axioms, Exercise

- Let $t_1, \ldots, t_n$ be the terms using `cons` in φ. So there are n sub-terms whose top-most function symbol is `cons` in the input formula. Prove that the following axioms suffice:

$$\begin{aligned}
\forall x_1, \ldots, x_n, \ell \,.\, \ell \neq cons(x_1, cons(x_2, \ldots, cons(x_n, \ell))) \\
\forall x_1, \ldots, x_{n-1}, \ell \,.\, \ell \neq cons(x_1, cons(x_2, \ldots, cons(x_{n-1}, \ell))) \\
\ldots \\
\forall x_1, x_2, \ell \,.\, \ell \neq cons(x_1, cons(x_2, \ell)) \\
\forall x_1, \ell \,.\, \ell \neq cons(x_1, \ell)
\end{aligned}$$

- Indicate what patterns should be used in the axioms.
- Encode the axioms using Z3.

Approach 2 - using model checking, Exercise

- Suppose Z3 returns a model where some list term in φ evaluates to a cyclic list.
- How would you block this term, or any other cycle of that length to re-appear?
- Does it matter if the model introduces a cycle only invovling terms not in φ (but introduced later for domain closure)?
- Write the function that checks a model for cycles and adds blocking clauses.

5.1.4 Working with model generation

The decision procedure we have outlined for recursive data-types reduces the satisfiability problem from recursive lists to plain first-order satisfiability. In other words, for every quantifier free formula φ over the signature of recursive lists, we produce a quantifier-free formula ψ, such that

$$\mathcal{T}_E \cup \mathcal{T}_L \cup \{\varphi\} \text{ is satisfiable iff } \mathcal{T}_E \cup \{\varphi,\ \psi\} \text{ is satisfiable}$$

In other words, φ is satisfiable in the theory $\mathcal{T}_L$ of recursive lists, and the theory $\mathcal{T}_E$ of equality, iff $\varphi \wedge \psi$ is satisfiable in the theory of equality. Unfortunately, this does not mean that the model obtained for $\varphi \wedge \psi$ corresponds directly to a model where every subterm of φ evaluates to a list. It may be the case that the model for $\varphi \wedge \psi$ does not specify fully how subterms in φ should be extended to a list, it just says that there is some extension. So how do we find a reasonable extension?

The Z3 API supplies a collection of methods for inspecting models. We summarize some of the main relevant methods here:

1. `let graphs = model.GetFunctionGraphs()` returns a dictionary from function declarations to the finite *graph* of the functions. Only functions that take at least one argument are listed in this dictionary. Functions, or rather constants, that don't take any arguments are not listed.
2. `graphs[func_decl]` results in an object with two fields:
   - `Entries` - a finite array of argument/value pairs.
   - `Else` - a default value. The function evaluates to the `Else` value on all argument combinations *not* listed in `Entries`.

Exercise:

- Let t be a sub-term in φ. Suppose that `model.Eval(t)` is a value that is in the range of the array `graphs[cons_decl].Entries`. What can be said about the value of t in `model`?
- Suppose `cons`(hd, tl) is a sub-term in ϖ, and that `model.Eval(cons(hd,tl))` is a value equal to `graphs[cons_decl].Entries`. What can be said about all sub-terms of type `int_list` in φ?
- The function `model.GetValueType(v)` returns the type associated with a model value `v`. Write a function that given a `model` and a model value `v` of type `int_list` determines whether `model` evaluates it to a value of the form `cons(ht,tl)` or `nil`.
- Suppose `t` is a sub-term in φ, and let `v = model.Eval(t)`. Why does `v` correspond to either a `cons` or `nil`?
- Write a function that for a value corresponding to a `cons`, returns the *tail* value (that is, the second argument of `cons`).
- Do the tail values necessarily correspond to a `cons` or `nil`?
- `(*)` Suppose we are given a model where some tail value does not correpond to a `cons` or `nil`. We can refine the model by adding some additional assertions and calling `CheckAndGetModel` again to refine the model further to either bind the tail value to `nil` or `cons`. Write the API calls that perform this model refinement.

Bibliography

[1] Gary William Flake and Eric B. Baum. Rush hour is pspace-complete, or ”why you should generously tip parking lot attendants”. *Theor. Comput. Sci.*, 270(1-2):895–911, 2002.