APL - Assignment n°1

By Arthur ADAM and Robin LOTODE

Part 1 - Alphametics

In order to test Z3 and its functionalities, we went first to the alphametics that we already experimented in the course ALG1.

We took 3 differents alphametics we found on this website.

Alphametic n°1: "AS + A = MOM"

In order to find a satisfying assignment to the variables, we choose the following input.txt file:

```
; Declaring constants
(declare-const A Int)
(declare-const S Int)
(declare-const M Int)
(declare-const O Int)

; A, S, M and O must be distinct, we must use the function 'distinct' from
SMTLIB
(assert (distinct A S M O))

; Now, we must ask our variables not to go higher than 9 and lower than 0.
; As variables A and M are the leftmost letters of each words, they can't
be equal to zero.
; Thus, we must make them higher than 0.
(assert (<= 1 A 9))
(assert (<= 0 S 9))
(assert (<= 1 M 9))
(assert (<= 0 O 9))

; Alphametic equation: AS + A == MOM
(assert (= (+ (* A 10) S A) (+ (* M 100) (* O 10) M)))

; Since, we're writing this code with SMTLIB and we're trying to solve a
SAT problem.
(check-sat-using smt)

; We're getting a satisfying assignment.
(get-model)
; AS + A = MOM
```

With this input.txt file, we're having the following output:

```
sat
(
  (define-fun A () Int
    9)
  (define-fun S () Int
    2)
  (define-fun O () Int
    0)
  (define-fun M () Int
    1)
)
```

This gives us the following equation: `92 + 9 = 101` which is true.

Alphametic n°2: "SEND + MORE = MONEY"

In order to find a satisfying assignment to the variables, we choose the following input.txt file:

```
; Declaring constants
(declare-const S Int)
(declare-const E Int)
(declare-const N Int)
(declare-const D Int)
(declare-const M Int)
(declare-const O Int)
(declare-const R Int)
(declare-const Y Int)

; All variables must be distinct, we must use the function 'distinct' from
SMTLIB
(assert (distinct S E N D M O R Y))

; Asking Z3 making all variables be between 0 and 9, except for S and M
the first letters of each words that in theory can't be equal to 0
; Since variables S and M are the leftmost letters of each words, they
can't be equal to zero.
; Thus, we must make them higher than 0.
(assert (<= 1 S 9))
(assert (<= 0 E 9))
(assert (<= 0 N 9))
(assert (<= 0 D 9))
(assert (<= 1 M 9))
(assert (<= 0 O 9))
(assert (<= 0 R 9))
(assert (<= 0 Y 9))

; Alphametic equation: AS + A == MOM
(assert (= (+ (+ (* S 1000) (* E 100) (* N 10) D) (+ (* M 1000) (* O 100)
(* R 10) E)) (+ (* M 10000) (* O 1000) (* N 100) (* E 10) Y)))

; Since, we're writing this code with SMTLIB and we're trying to solve a
```

```
SAT problem.
(check-sat-using smt)

; We're getting a satisfying assignment.
(get-model)
; SEND MORE MONEY
```

With this input.txt file, we're having the following output:

```
sat
(
  (define-fun D () Int
    7)
  (define-fun N () Int
    6)
  (define-fun R () Int
    8)
  (define-fun E () Int
    5)
  (define-fun M () Int
    1)
  (define-fun S () Int
    9)
  (define-fun O () Int
    0)
  (define-fun Y () Int
    2)
)
```

This gives us the following equation: `9567 + 1085 = 10652` which is true.

Alphametic n°3: "BEER + AND = SODAS"

In order to find a satisfying assignment to the variables, we choose the following input.txt file:

```
; Declaring constants
(declare-const B Int)
(declare-const E Int)
(declare-const R Int)
(declare-const A Int)
(declare-const N Int)
(declare-const D Int)
(declare-const S Int)
(declare-const O Int)

; All variables must be distinct, we must use the function 'distinct' from
SMTLIB
(assert (distinct B E R A N D S O))
```

```
; Now, we must ask our variables not to go higher than 9 and lower than 0.
; As variables A and M are the leftmost letters of each words, they can't
be equal to zero.
; Thus, we must make them higher than 0.
(assert (<= 1 B 9))
(assert (<= 0 E 9))
(assert (<= 0 R 9))
(assert (<= 1 A 9))
(assert (<= 0 N 9))
(assert (<= 0 D 9))
(assert (<= 1 S 9))
(assert (<= 0 O 9))

; Alphametic equation: AS + A == MOM
(assert (= (+ (+ (* B 1000) (* E 100) (* E 10) R) (+ (* A 100) (* N 10)
D)) (+ (* S 10000) (* O 1000) (* D 100) (* A 10) S)))

; Since, we're writing this code with SMTLIB and we're trying to solve a
SAT problem.
(check-sat-using smt)

; We're getting a satisfying assignment.
(get-model)

; BEER AND SODAS
```

With this input.txt file, we're having the following output:

```
sat
(
  (define-fun A () Int
    5)
  (define-fun R () Int
    7)
  (define-fun N () Int
    6)
  (define-fun D () Int
    4)
  (define-fun E () Int
    8)
  (define-fun S () Int
    1)
  (define-fun B () Int
    9)
  (define-fun O () Int
    0)
)
```

This gives us the following equation: `9887 + 564 = 10451` which is true.

Part 2: Logic Equations

Now that we've done 3 alphametics, we decided to work on 3 logic equations.

We chose 3 logic equations that had a pretty average difficulty.

Logic Equation n°1

To access the logic equation, just click this link.

The first logic equation we worked on had 4 variables and the following requirements:

- Variable A must not be equal to **3 or 4**.
- Variable B must not be equal to **4**.
- Variable C must not be equal to **4**.
- Variable D must not be equal to **2**.
- The equation `A + C = B + D` must be **true**.

Since we have 4 variables, all variables must have their value between 1 and 4.

In order to find a satisfying assignment to those variables, we choose the following input.txt file:

```
; Declaring the variables
(declare-const A Int)
(declare-const B Int)
(declare-const C Int)
(declare-const D Int)

; Once again, all variables must be distinct.
(assert (distinct A B C D))

; We apply the different requirements listed above
(assert (not (= A 3)))
(assert (not (= A 4)))
(assert (not (= B 4)))
(assert (not (= C 4)))
(assert (not (= D 2)))
(assert (= (+ B D) (+ A C)))

; Since we have 4 variables, all variables must be located between 1 and
4.
(assert (<= 1 A 4))
(assert (<= 1 B 4))
(assert (<= 1 C 4))
(assert (<= 1 D 4))

; We're using SMT Tactic to solve an SAT problem otherwise Z3 would crash.
(check-sat-using smt)

; We're getting the right assignment.
(get-model)
```

With this input.txt file, we're having the following output:

```
sat
(
  (define-fun A () Int
    2)
  (define-fun D () Int
    4)
  (define-fun B () Int
    1)
  (define-fun C () Int
    3)
)
```

We can check that all requirements are validated:

- A = 2 is not equal to 3 or 4.
- B = 1 is not equal to 4.
- C = 3 is not equal to 4.
- D = 4 is not equal to 2.
- `2 + 3 = 4 + 1` is equivalent to `5 = 5`. `A + C = B + D` is true with this assignment.

Logic Equation n°2

To access the logic equation, just click this link.

The first logic equation we worked on had 4 variables and the following requirements:

- Variable B must be greater than Variable C.
- The equation `2C = A + D` must be **true**.
- The equation `2A = B + C` must be **true**.

Since we have 4 variables, all variables must have their value between 1 and 4.

In order to find a satisfying assignment to those variables, we choose the following input.txt file:

```
; Declaring the variables
(declare-const A Int)
(declare-const B Int)
(declare-const C Int)
(declare-const D Int)

; Once again, all variables must be distinct.
(assert (distinct A B C D))

; We apply the different requirements listed above
(assert (= (* 2 C) (+ A D)))
(assert (= (* 2 A) (+ B C)))
(assert (> B C))
```

```
; Since we have 4 variables, all variables must be located between 1 and
4.
(assert (<= 1 A 4))
(assert (<= 1 B 4))
(assert (<= 1 C 4))
(assert (<= 1 D 4))


; We're using SMT Tactic to solve an SAT problem otherwise Z3 would crash.
(check-sat-using smt)

; We're getting the right assignment.
(get-model)
```

With this input.txt file, we're having the following output:

```
sat
(
  (define-fun A () Int
    3)
  (define-fun D () Int
    1)
  (define-fun B () Int
    4)
  (define-fun C () Int
    2)
)
```

We can check that all requirements are validated:

- B = 4 is greater than C = 2.
- `2 * 2 = 3 + 1` is equivalent to `4 = 4`. `2C = A + D` is true with this assignment.
- `2 * 3 = 4 + 2` is equivalent to `6 = 6`. `2A = B + C` is true with this assignment.

Logic Equation n°3

To access the logic equation, just click this link.

The first logic equation we worked on had 4 variables and the following requirements:

- Variable A must be lower than Variable C.
- The inequation `B + C <= 5` must be **true**.
- The equation `C = B + D` must be **true**.

Since we have 4 variables, all variables must have their value between 1 and 4.

In order to find a satisfying assignment to those variables, we choose the following input.txt file:

```
; Declaring the variables
(declare-const A Int)
(declare-const B Int)
(declare-const C Int)
(declare-const D Int)

; Once again, all variables must be distinct.
(assert (distinct A B C D))

; We apply the different requirements listed above
(assert (= C (+ B D)))
(assert (<= (+ B C) 5))
(assert (> C A))

; Since we have 4 variables, all variables must be located between 1 and
4.
(assert (<= 1 A 4))
(assert (<= 1 B 4))
(assert (<= 1 C 4))
(assert (<= 1 D 4))

; We're using SMT Tactic to solve an SAT problem otherwise Z3 would crash.
(check-sat-using smt)

; We're getting the right assignment.
(get-model)
```

With this input.txt file, we're having the following output:

```
sat
(
  (define-fun A () Int
    2)
  (define-fun D () Int
    3)
  (define-fun B () Int
    1)
  (define-fun C () Int
    4)
)
```

We can check that all requirements are validated:

- A = 2 is lower than C = 4.
- `1 + 4 <= 5` is equivalent to `5 <= 5`. `B + C <= 5` is true with this assignment.
- `4 = 1 + 3` is equivalent to `4 = 4`. `C = B + D` is true with this assignment.

Part n°3: Inverse Matrix

The third part of this assignment is a inverse matrix satisfiability.

We had to find 2 inverse matrixes based on a start matrix.

In order to setup our input file, we're going to design 8 variables:

- `a`, `b`, `c` and `d` are the variables of the start matrix.
- `aa`, `bb`, `cc` and `dd` are the variables of the inverse matrix.

The second step we need to design is the determinant calculation to find out if the matrix can have an inverse matrix of itself.

Matrix n°1

The matrix n°1 is the following:

3	1
5	2

To find out a satisfying assignment to variables `aa`, `bb`, `cc` and `dd`, we will use the following input.txt file:

```
; Declaring base matrix numbers
(declare-const a Int)
(declare-const b Int)
(declare-const c Int)
(declare-const d Int)

; Declare inverse matrix numbers
(declare-const aa Int)
(declare-const bb Int)
(declare-const cc Int)
(declare-const dd Int)

; Defining the values of base matrix numbers
(assert (= a 3))
(assert (= b 1))
(assert (= c 5))
(assert (= d 2))

; Defining a function to get the matrix determinant
(define-fun get-det () Int
    (- (* a d) (* b c))
)

; Defining a function to check if the matrix can have an inverse matrix
(define-fun check-good () Bool
    (if (= get-det 0)
        false
        true
    )
)
```

```
; Asserting that the matrix must have an inverse matrix
(assert check-good)

; Asserting values of the inverse matrix numbers
(assert (= aa (* (/ 1 get-det) d)))
(assert (= bb (* (/ 1 get-det) (* -1 b))))
(assert (= cc (* (/ 1 get-det) (* -1 c))))
(assert (= dd (* (/ 1 get-det) a)))

; Checking if the values are satisfiable
(check-sat)

; Getting the values of the variables
(get-model)
```

This input file for z3 gives us the following output:

```
sat
(
  (define-fun aa () Int
    2)
  (define-fun dd () Int
    3)
  (define-fun bb () Int
    (- 1))
  (define-fun cc () Int
    (- 5))
  (define-fun d () Int
    2)
  (define-fun c () Int
    5)
  (define-fun b () Int
    1)
  (define-fun a () Int
    3)
  (define-fun check-good () Bool
    true)
  (define-fun get-det () Int
    1)
)
```

Since all values of the inverse matrix are integers, a solution exists! Our first inverse matrix will be:

2	-1
-5	3

Matrix n°2

The matrix n°2 is the following matrix:

4	7
2	6

To find out a satisfying assignment to variables `aa`, `bb`, `cc` and `dd`, we will use the following input.txt file:

```
; Declaring base matrix numbers
(declare-const a Int)
(declare-const b Int)
(declare-const c Int)
(declare-const d Int)

; Declare inverse matrix numbers
(declare-const aa Int)
(declare-const bb Int)
(declare-const cc Int)
(declare-const dd Int)

; Defining the values of base matrix numbers
(assert (= a 3))
(assert (= b 1))
(assert (= c 5))
(assert (= d 2))

; Defining a function to get the matrix determinant
(define-fun get-det () Int
    (- (* a d) (* b c))
)

; Defining a function to check if the matrix can have an inverse matrix
(define-fun check-good () Bool
    (if (= get-det 0)
        false
        true
    )
)

; Asserting that the matrix must have an inverse matrix
(assert check-good)

; Asserting values of the inverse matrix numbers
(assert (= aa (* (/ 1 get-det) d)))
(assert (= bb (* (/ 1 get-det) (* -1 b))))
(assert (= cc (* (/ 1 get-det) (* -1 c))))
(assert (= dd (* (/ 1 get-det) a)))

; Checking if the values are satisfiable
(check-sat)
```

```
; Getting the values of the variables
(get-model)
```

Unfortunately, even with the best will in the world, we can't achieve a result and z3 will return an error (with our z3 experience, we can't figure out how to perform a sys.exit kind of operation with SMTLIB language).

```
Error: unsat
(error "line 45 column 11: model is not available")
```

This error occurs because the model can't find a good assignment to the inverse matrix variable.

Therefore, a solution exists! If you change the type of the inverse matrix variables to Real, a satisfying assignment exists.

You would be able to get a satisfying assignment with the following input file:

```
; Declaring base matrix numbers
(declare-const a Int)
(declare-const b Int)
(declare-const c Int)
(declare-const d Int)

; Declare inverse matrix numbers
(declare-const aa Real)
(declare-const bb Real)
(declare-const cc Real)
(declare-const dd Real)

; Defining the values of base matrix numbers
(assert (= a 3))
(assert (= b 1))
(assert (= c 5))
(assert (= d 2))

; Defining a function to get the matrix determinant
(define-fun get-det () Int
    (- (* a d) (* b c))
)

; Defining a function to check if the matrix can have an inverse matrix
(define-fun check-good () Bool
    (if (= get-det 0)
        false
        true
    )
)

; Asserting that the matrix must have an inverse matrix
(assert check-good)
```

```
; Asserting values of the inverse matrix numbers
(assert (= aa (* (/ 1 get-det) d)))
(assert (= bb (* (/ 1 get-det) (* -1 b))))
(assert (= cc (* (/ 1 get-det) (* -1 c))))
(assert (= dd (* (/ 1 get-det) a)))

; Checking if the values are satisfiable
(check-sat)

; Getting the values of the variables
(get-model)
```

This input file gets the following output file:

```
sat
(
  (define-fun dd () Real
    3.0)
  (define-fun cc () Real
    (- 5.0))
  (define-fun bb () Real
    (- 1.0))
  (define-fun aa () Real
    2.0)
  (define-fun d () Int
    2)
  (define-fun c () Int
    5)
  (define-fun b () Int
    1)
  (define-fun a () Int
    3)
  (define-fun check-good () Bool
    true)
  (define-fun get-det () Int
    1)
)
```