Luke Attard

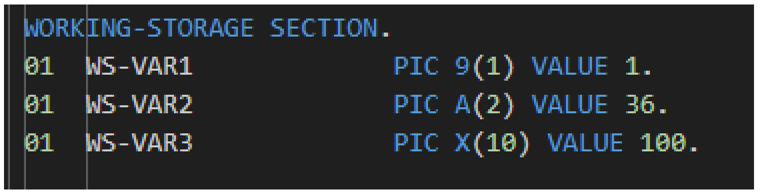
12/5/2025

Final Project README.

For my final project I chose to create a static analysis tool for the COBOL language. Particularly built on the GNUCOBOL translation compiler, as that is what I am familiar with. In my experience working with the language I have had issues with the variable type checking in this compiler, as it is very loose and doesn’t check much beyond basic types.

Picture clauses:

COBOL uses picture clauses such as these ones to declare data types:



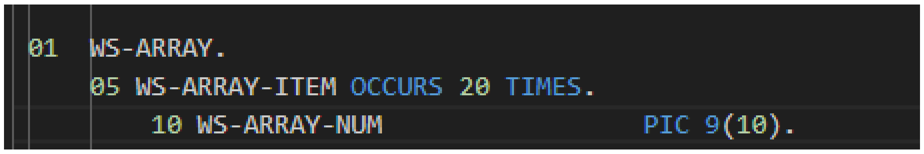
```
WORKING-STORAGE SECTION.
01  WS-VAR1            PIC 9(1) VALUE 1.
01  WS-VAR2            PIC A(2) VALUE 36.
01  WS-VAR3            PIC X(10) VALUE 100.
```

Datatypes include 9 (numerical), A (alphabetical), X (alphanumerical), S9 (signed numerical) and many more. The size of the variable is declared as the number of characters it occupies; this is written inside the parenthesis.

This compiler will do some basic testing for things like moving an “A” into a “9” but it will not check if variables share similar sizes before moving them, so you can move a value 10 characters long into a variable only 5 characters long, and all excess data is simply lost. Note that this is only an issue when data is moved from a larger to a smaller variable, I made sure to account for this in my solver. Sometimes this is useful behavior, but other times it can lead to unexpected errors. I wanted to make an analysis tool that could spot these errors ahead of time. And just for fun I thought I would code it in COBOL as well.

Challenges:

It is unfortunately a very bad idea to make a parser with such a static language, so that in and of itself was quite a challenge. Some drawbacks of my solver include not parsing variables within function calls, not being able to properly handle all literals, no support for divide calls (I ran out of time) and some inconcistencies with arrays (array values can be called from different levels, meaning you cannot always expect the same text, for example here:



```
01  WS-ARRAY.
    05 WS-ARRAY-ITEM OCCURS 20 TIMES.
       10 WS-ARRAY-NUM              PIC 9(10).
```

Using WS-ARRAY-ITEM (1) is equivalent to WS-ARRAY-NUM(1) and my solver only accounts for the latter.

How it works:

The solver parses the program with a very basic tokenizer that breaks the program into words (by spaces). There are drawbacks to this approach I did not foresee that I will discuss later. Once the program has been parsed the parser passes the array of words to the prover. The prover then loops over the program to find all variables. These variables are then stored in a table with their name, type, and size. The prover knows it has finished finding variables once it begins reading "Procedure Division" in which all the core logic of a COBOL program is located.

Now that a list of program tokens exists, and all environment variables have been declared the prover can begin looping over the main logic to look for type and size mismatches. It looks for keywords (MOVE, ADD, SUBTRACT, MULT, AND COMPUTE) and anytime one is encountered, it checks the types and sizes of the arguments included. If a mismatch in type is found, it will always report it. If a mismatch in size is found, only potentially harmful ones are reported (moving from large data variables to low ones, causing truncations).

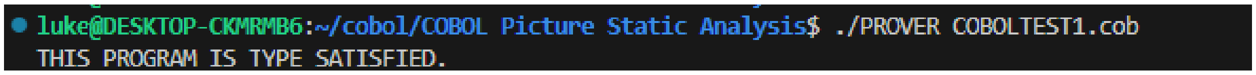
To work through an example.

In COBOLTEST1.cob

First the program calls the parser with the command line argument. The parser returns a sanitized list of all words in the program. The prover then loops over those words until it finds the procedure division, along the way it has caught var1 – var4. Now it loops over the procedure calls, and finds that all variables are of matching type/size.

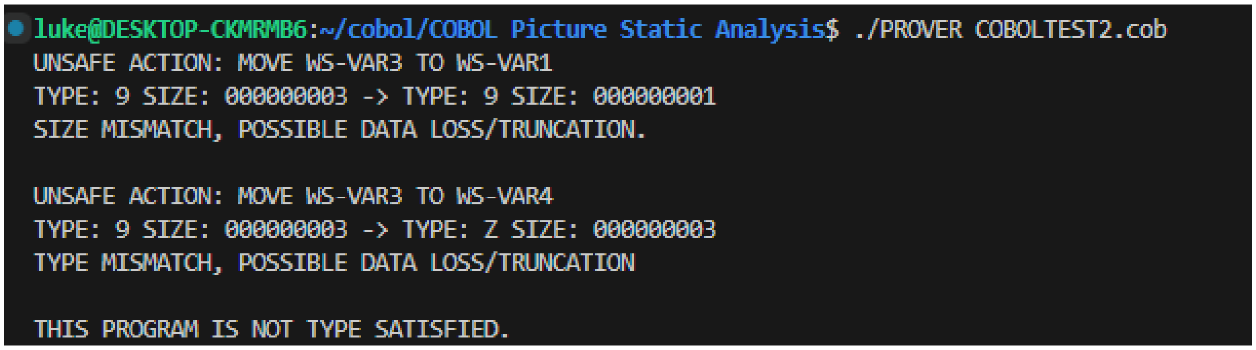
Tests and Outputs:

Test 1 aims to show that the prover can properly parse a basic test program, and identify that none of the move clauses are incorrect/unsafe. This is the output:



```
● luke@DESKTOP-CKMRMB6:~/cobol/COBOL Picture Static Analysis$ ./PROVER COBOLTEST1.cob
  THIS PROGRAM IS TYPE SATISFIED.
```

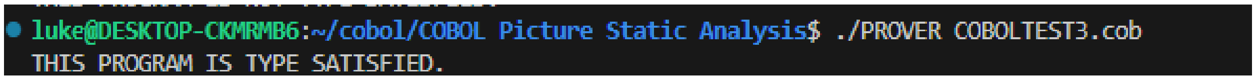
Test 2 aims to show the same program as program 1 but with unave moves introduced, this is the output



```
● luke@DESKTOP-CKMRMB6:~/cobol/COBOL Picture Static Analysis$ ./PROVER COBOLTEST2.cob
  UNSAFE ACTION: MOVE WS-VAR3 TO WS-VAR1
  TYPE: 9 SIZE: 000000003 -> TYPE: 9 SIZE: 000000001
  SIZE MISMATCH, POSSIBLE DATA LOSS/TRUNCATION.

  UNSAFE ACTION: MOVE WS-VAR3 TO WS-VAR4
  TYPE: 9 SIZE: 000000003 -> TYPE: Z SIZE: 000000003
  TYPE MISMATCH, POSSIBLE DATA LOSS/TRUNCATION

  THIS PROGRAM IS NOT TYPE SATISFIED.
```

We have a type and size mismatch in separate places here as we try to move values from one type to another and one size to another.

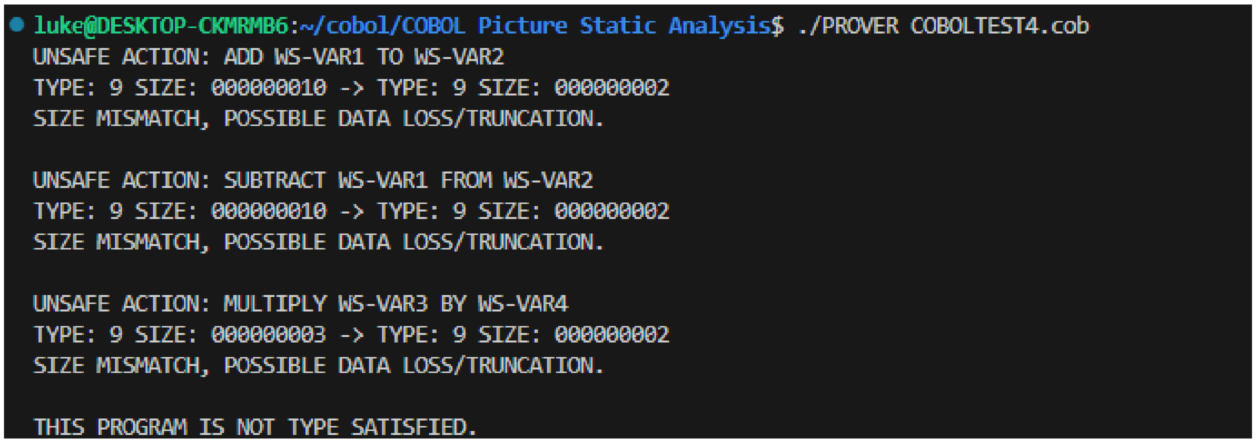
Test 3 aims to show how the same principle from the previous two tests also applies to some of COBOL's other statements, namely the add, subtract, multiply. As they move data from variable to variable it is important our program works here too.



```
● luke@DESKTOP-CKMRMB6:~/cobol/COBOL Picture Static Analysis$ ./PROVER COBOLTEST3.cob
  THIS PROGRAM IS TYPE SATISFIED.
```

The output is satisfied as no unsafe code exists.

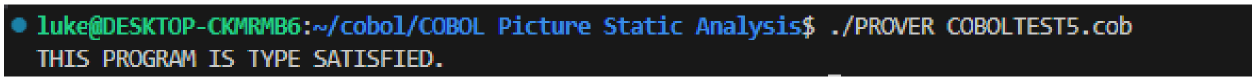
Test 4 again aims to foil the previous test, showing examples of failing satisfaction.



```
● luke@DESKTOP-CKMRMB6:~/cobol/COBOL Picture Static Analysis$ ./PROVER COBOLTEST4.cob
  UNSAFE ACTION: ADD WS-VAR1 TO WS-VAR2
  TYPE: 9 SIZE: 000000010 -> TYPE: 9 SIZE: 000000002
  SIZE MISMATCH, POSSIBLE DATA LOSS/TRUNCATION.

  UNSAFE ACTION: SUBTRACT WS-VAR1 FROM WS-VAR2
  TYPE: 9 SIZE: 000000010 -> TYPE: 9 SIZE: 000000002
  SIZE MISMATCH, POSSIBLE DATA LOSS/TRUNCATION.

  UNSAFE ACTION: MULTIPLY WS-VAR3 BY WS-VAR4
  TYPE: 9 SIZE: 000000003 -> TYPE: 9 SIZE: 000000002
  SIZE MISMATCH, POSSIBLE DATA LOSS/TRUNCATION.

  THIS PROGRAM IS NOT TYPE SATISFIED.
```

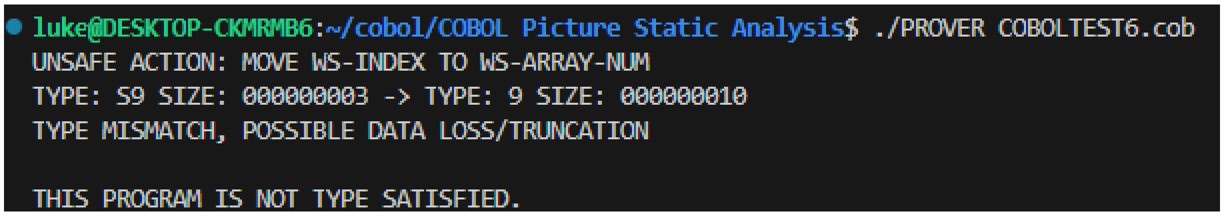
Here we see all size mismatches are caught. It is important to note that when adding var2 to var1 no error is encountered as var1 is of size 10 and var2 is of size 2. (smaller into bigger is not an issue).

Test 5 only exists to show the parser and solver can understand how arrays and literals (for the most part) are defined inside of COBOL, and the program satisfies moving literals into arrays



```
● luke@DESKTOP-CKMRMB6:~/cobol/COBOL Picture Static Analysis$ ./PROVER COBOLTEST5.cob
  THIS PROGRAM IS TYPE SATISFIED.
```

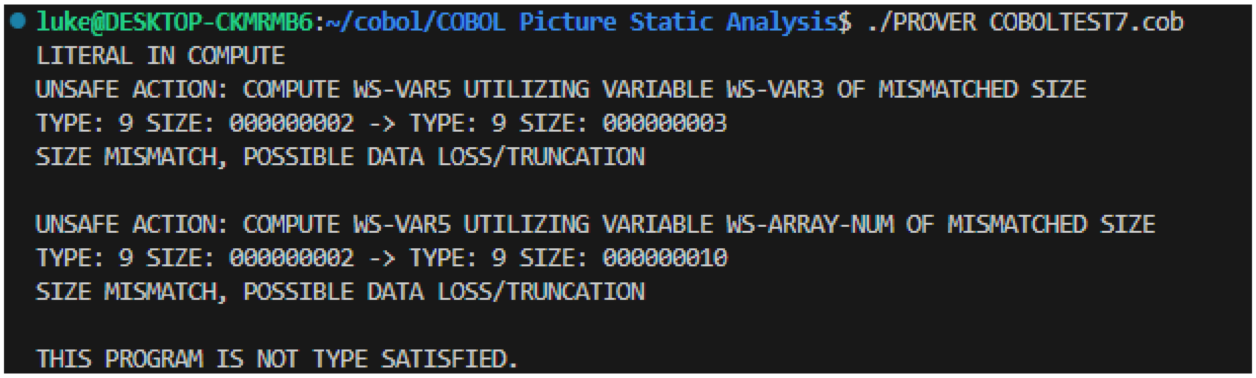
Test 6 is a much more interesting program as it includes a loop. It aims to show the inverse of the previous test, that a type/size error can be found even with array reads and writes.



```
● luke@DESKTOP-CKMRMB6:~/cobol/COBOL Picture Static Analysis$ ./PROVER COBOLTEST6.cob
  UNSAFE ACTION: MOVE WS-INDEX TO WS-ARRAY-NUM
  TYPE: S9 SIZE: 000000003 -> TYPE: 9 SIZE: 000000010
  TYPE MISMATCH, POSSIBLE DATA LOSS/TRUNCATION

  THIS PROGRAM IS NOT TYPE SATISFIED.
```

Moving a signed number to an unsigned variable can lead to some issues (that I have personally struggled with unknowingly in the past) so here the type mismatch is marked. Note that even though the inverse wouldn't be an issue (moving a non signed to a signed) my solver is not advanced enough to know that, and will always flag type mismatches.

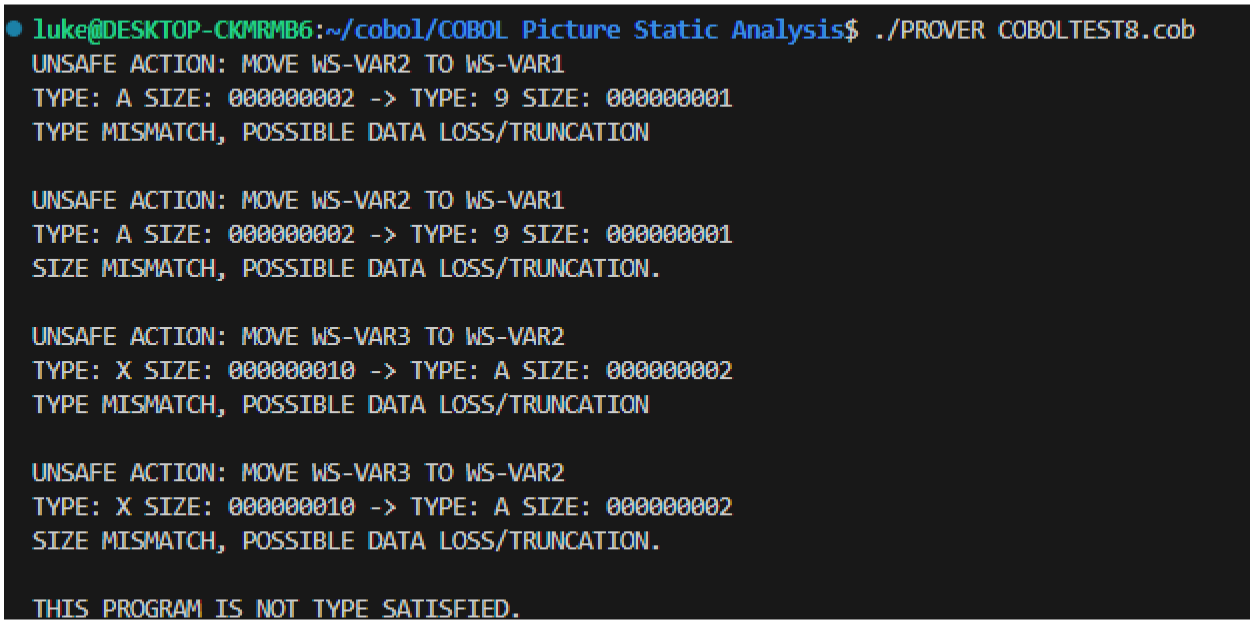
Test 7 shows off computes, both satisfied and unsatisfied. Compute statments are extremely useful in COBOL and are probably more common than add, subtract, multiply etc. put together because you can do all these things in one line with a compute statement. Parsing and proving these statements was a goal of mine and it was quite a bit more complicated than the others, as they can have any number of arithmetic expressions (and this is not a very dynamic language). Here is the output:



```
luke@DESKTOP-CKMRMB6:~/cobol/COBOL Picture Static Analysis$ ./PROVER COBOLTEST7.cob
LITERAL IN COMPUTE
UNSAFE ACTION: COMPUTE WS-VAR5 UTILIZING VARIABLE WS-VAR3 OF MISMATCHED SIZE
TYPE: 9 SIZE: 000000002 -> TYPE: 9 SIZE: 000000003
SIZE MISMATCH, POSSIBLE DATA LOSS/TRUNCATION

UNSAFE ACTION: COMPUTE WS-VAR5 UTILIZING VARIABLE WS-ARRAY-NUM OF MISMATCHED SIZE
TYPE: 9 SIZE: 000000002 -> TYPE: 9 SIZE: 000000010
SIZE MISMATCH, POSSIBLE DATA LOSS/TRUNCATION

THIS PROGRAM IS NOT TYPE SATISFIED.
```

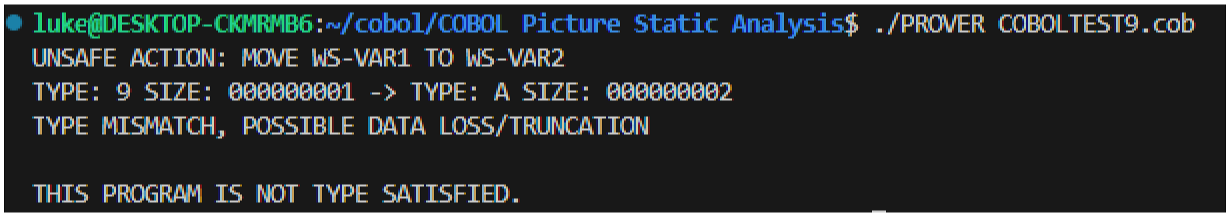
The second compute contains two mismatches of different kinds, and both are caught.

Test 8 shows off some mismatch catches between string types (A) and (x) as well as between numbers (9) even though that isn't legal code, and the compiler would find it.



```
luke@DESKTOP-CKMRMB6:~/cobol/COBOL Picture Static Analysis$ ./PROVER COBOLTEST8.cob
UNSAFE ACTION: MOVE WS-VAR2 TO WS-VAR1
TYPE: A SIZE: 000000002 -> TYPE: 9 SIZE: 000000001
TYPE MISMATCH, POSSIBLE DATA LOSS/TRUNCATION

UNSAFE ACTION: MOVE WS-VAR2 TO WS-VAR1
TYPE: A SIZE: 000000002 -> TYPE: 9 SIZE: 000000001
SIZE MISMATCH, POSSIBLE DATA LOSS/TRUNCATION.

UNSAFE ACTION: MOVE WS-VAR3 TO WS-VAR2
TYPE: X SIZE: 000000010 -> TYPE: A SIZE: 000000002
TYPE MISMATCH, POSSIBLE DATA LOSS/TRUNCATION

UNSAFE ACTION: MOVE WS-VAR3 TO WS-VAR2
TYPE: X SIZE: 000000010 -> TYPE: A SIZE: 000000002
SIZE MISMATCH, POSSIBLE DATA LOSS/TRUNCATION.

THIS PROGRAM IS NOT TYPE SATISFIED.
```

Test 9 is unfortunately here to show off some of the drawbacks of my naïve approach to parsing. Because my parser simply breaks lines apart into words in its "tokenizer" it does not store strings together, and as such the solver can possibly read literals as variables and statements.



```
● luke@DESKTOP-CKMRMB6:~/cobol/COBOL Picture Static Analysis$ ./PROVER COBOLTEST9.cob
  UNSAFE ACTION: MOVE WS-VAR1 TO WS-VAR2
  TYPE: 9 SIZE: 000000001 -> TYPE: A SIZE: 000000002
  TYPE MISMATCH, POSSIBLE DATA LOSS/TRUNCATION

  THIS PROGRAM IS NOT TYPE SATISFIED.
```

Because I do not have time to write a better parser, the output declares this an unsatisfied program.

How to run this tool:

I have compiled the Prover and included it in this submission as PROVER. To run it all you have to do is run ./PROVER [TestFileName] and it will type check the file you provide. If you want to compile it yourself, you will first need to install GNUCOBOL in your environment. This can be done with apt install if your distro allows you to.