Programming Project for AI Programming / KI-Programmierung, Option B

Solving Systems of Diophantine Equations with Constraints

Cezar Ionescu

WS 2022

Project Description

The aim of this project is to implement a function that solves systems of *diophantine* equations (i.e., the variables involved are integers) under constraints. The system is given in a file having the form shown in the following example:

```
Solve
  x + y + z = 10,
  x     - z =  5
such that
  x > 5 or x < -5 and y > 0,
  z < 0.
```

The `solve` function takes as argument the file name and returns *one* of the solutions to the system, represented *as a dictionary*, or prints an error message if no solutions exist. For example, if the above system is in the file `sys.txt`, the `solve` function could act as follows:

```
>>> sol = solve(”sys.txt”)
>>> sol
{’x’: -6, ’y’: 27, ’z’: -11}
```

while the system

```
Solve
  x + y + z = 10,
  x     - z =  5
such that
  x > 5 or (x < -5 and y < 0),
  z < 0.
```

would lead to

```
>>> sol = solve(”sys.txt”)
No solution!
```

Equations and Constraints

The equations and constraints are made of symbolic expressions that combine arithmetical and boolean values.

Arithmetical Expressions

Arithmetical expressions are built from integer constants, variables, addition, subtraction and multiplication.

Examples of arithmetical expressions:

```
42
```

```
x

x + 2*y

15 + x * x

(x - y) - z
```

All operations are assumed to associate to the right (so `x - y - z` would result in `x - (y - z)`).

Boolean Expressions

The atomic propositions are

- equality statements $e_1 = e_2$
- strict inequality $e_1 < e_2$ or $e_1 > e_2$

where e_1 and e_2 are arithmetical expressions.

The equality statements represent the equations to be solved, while the inequalities represent the constraints.

Boolean expressions are either atomic propositions, or combinations of boolean expressions by means of and and `or`. Examples:

```
x = y

x < 15 and 7 > x

x < 15 or 20 < -x
```

Constraints can be arbitrarily complex, e.g.: `x < 15 or x > 20 and y < 10 or y > 5 and z < 0`. In these combinations, and binds tighter than `or`. Thus:

```
x < 15 or x > 20 and y < 10 or y > 5 and z < 0
```

is equivalent to

```
x < 15 or ((x > 20 and y < 10) or (y > 5 and z < 0))
```

As this example show, using parantheses, even if not necessary, may be a good idea. Your parser should allow for such cases as well!

System Description

A system description starts with the keyword `Solve` followed by one or more equations (separated by commas), followed optionally by the keywords `such that` followed by one or more constraints separated by commas. The system description ends with a full stop. Examples:

- only equations:

  ```
  Solve
    x*x -   z = u,
    x   +   u = 0,
    y   - 5*z = 3.
  ```

  Example solution:

  ```
  >>> sol = solve(”only_eqs”)
  >>> sol
  {’y’: 3, ’x’: -1, ’u’: 1, ’z’: 0}
  ```

- a simple system:

  ```
  Solve
    2*x*x     = -y,
  ```

```
  x     - z =  5
such that
  x > 0,
  z < 0.
```

Example solution:

```
>>> sol = solve(”simple”)
>>> sol
{’z’: -3, ’x’: 2, ’y’: -8}
```

- a system with no solutions:

```
Solve
  x + y + z = 10,
  x     - z =  5
such that
  x > 5 or x < z and y > 0,
  z < 0.
```

- a system with one equation and more complex constraints:

```
Solve
  2 * x + y*y + (z - u) - 3*v = t
such that
  x > 0 and v < 0 and (z < 0 or u > 0),
  y < 0 and (z > 0 or t > 0 and v < 0).
```

Example solution:

```
>>> sol = solve(”complex”)
>>> sol
{’z’: 0, ’u’: 7, ’v’: -1, ’x’: 1, ’y’: -3, ’t’: 7}
```

A Grammar of Systems

The first task is to create a BNF grammar for the systems of equations with constraints. This should be based on the description given above and on the examples. You may use any “reasonable” BNF-variant that you wish: EBNF, BNFC, Russell-Norvig style, etc.

Note: The examples given above are expected to parse without errors!

Implementing Equations and Constraints

The next task is to implement classes for representing the equations and constraints. Feel free to reuse code we implemented in the course, but avoid bringing in dead code (e.g., there is no use here for differentiation!).

Parsing

The result of parsing a system description should be a list of equations and constraints. I recommend using our parser combinators to implement the parsing, but you are allowed to use other methods as well. However, other methods will be graded “all or none”: full marks if the parsing works perfectly, otherwise zero!

Using the Z3 Solver

The heavy-lifting of actually solving the systems should be left to the Z3 solver. In order to achieve this, you should translate the parsed expressions to their equivalent Z3 form. You could achieve this by adding a `toZ3` method to the various classes (similar in many ways to the `__str__` method). This method would translate an arithmetic or boolean expression to a corresponding Z3 expression.

Note: the variables should be translated to Z3 integer variables.

Summary and Assessment

The project consists of several parts:

- a BNF grammar for systems of diophantine equations with constraints
- classes for representing symbolic expressions (arithmetic and boolean)
- a parser for system descriptions
- a method of translating the equations and constraints to Z3 expressions
- an application of Z3 to solving the systems.

The project must be completed **in pairs** (or individually). I recommend splitting the work along the parsing/Z3 divide, working together on the grammar and designing the classes. This way, the bulk of the implementation can be parallelized, since Z3 doesn't connect directly to the parser.

Each submission must be accompanied by two documents, each one to two pages long:

- a joint description of the grammar and the design of the classes that implement the task descriptions, and of the code that was produced jointly
- an **individual paper** containing
  - a clear identification of the parts for which they were responsible
  - a brief description of their code (e.g., the parsing, or the `toZ3` methods, etc.)
  - additional remarks that can help the reader understand and use the code
  - a list of any deviations from the specification that you needed to make
  - potential extensions that could be made to the code, or alternative designs or implementations
  - a list of bugs that you are aware of.

In brief, the description should summarize the information you would communicate in an oral presentation in front of a code review team.

The final grade will be computed from a weighted average of marks awarded for code correctness, clarity, consistency, and efficiency, and the marks awarded for the description.

Please note:

- You may collaborate freely on the code and the joint document with your team partner, **but not on the individual paper**. **No collaboration is permitted outside the team.**
- Each participant must submit an archive (`zip`, `rar`, `tar`, `tgz`, etc.) containing:
  - a README file with the name **and** matriculation number of the participant, and the name of the other team member
  - the work description (both joint and individual) in plain text or pdf format (**not doc, docx, odt, rtf, html, ...**) Markdown formats are acceptable, as long as they are contained in plain text files,
  - the code produced by the team.

  Thus, both participants will submit the entire code and the joint work description, and each participant submits their own individual paper.

  The archive must be uploaded to iLearn by the specified deadline. **Solutions sent to me by email will not be considered!**
- You may use any code we implemented in the course, but you **must only include what is necessary** for the assignment (e.g., no code for differentiation of symbolic expressions!)

- Follow the specification to the letter! Do not, for example, use & instead of and, or introduce extra punctuation to separate equations or constraints, etc.
- You may assume that the input file is well formed, i.e.
  - there are no dangling parentheses or missing commas
  - no variables are called 'such' or 'that'
  - there are no non-integer numbers, etc.

  Therefore, you do not need to check for such errors in input. However, make sure you do not parse ungrammatical inputs! For example, `Solve x = 3 and y = 0.` should fail (because the section before 'such that' is not allowed to contain composite boolean expressions). The resulting parse is empty and the program is allowed to crash because of "user error". If you do parse this erroneous input and even return a solution, then points will be deducted!
- Code must be commented to help the reader understand what is going on and explain design decisions not listed in the work description. You should include testable examples in docstrings, at least for the main functions (such as `solve`).

  In general, you are strongly encouraged to test your project before submitting it! In the past, a surprisingly large number of submissions contained errors that would have easily been caught by a reasonable testing process.
- The only acceptable format for Python code is plain `*.py` scripts! **Jupyter notebooks and other formats will not be taken into account under any circumstances!**

  Make sure that your code is legible and aim for clarity and consistency (which are more important in this context than efficiency). Be aware that tools that automatically extract tools from notebooks often leave the code in a messy state. Submitting such code will lead to significant point deductions!

Failure to observe these points will likely lead to a failing mark on this project!