Program Verification 2024
Project 1

27 March, 2024

1 General Information

- This is the first graded project of the Program Verification course, which will count for 45% of the final grade.
- You can earn up to 50 points.
- *The project is to be completed individually by each student participating in the course.*
- The submission deadline is 23:59 (CEST) on 1 May, 2024.
- Skeleton files are available on the course website[1].
- Submit your solution by sending an e-mail containing a single `.zip` file with the completed skeleton files to `aurel.bily@inf.ethz.ch`. Use the subject "[PV] Submission Project 1".

2 Tools

This project is to be implemented in Haskell. The recommended way to install Haskell is to use GHCup[2], although your OS distribution may provide a Haskell package as well. The skeleton files require no additional Haskell libraries to be installed.

The output of your program is Viper code, and it should work with at least one Viper backend (Silicon or Carbon). For the Viper installation itself, we strongly recommend using the VS Code Viper IDE extension[3]. Make sure you are using the most recent version of Viper available (release 2024.1).

For a gentle introduction to Haskell, we recommend the Learn You a Haskell for Great Good! tutorial[4]. For detailed information about Viper's assertions and expressions, we recommend the Viper tutorial[5].

[1] `https://www.pm.inf.ethz.ch/education/courses/program-verification.html`
[2] `https://www.haskell.org/ghcup/`
[3] `https://www.pm.inf.ethz.ch/research/viper/downloads.html`
[4] `https://learnyouahaskell.github.io/chapters.html`
[5] `https://viper.ethz.ch/tutorial/`

3 A spreadsheet language

3.1 Spreadsheets

In this project, you will implement a Viper frontend for spreadsheets. Each spreadsheet consists of a two-dimensional grid of cells, with a given width and height. Each cell in the grid is one of the following:

- an empty cell (or a comment);
- a constant integer value;
- an integer user input, optionally with an associated precondition; or
- a program, optionally with an associated postcondition or a transparency annotation, with an integer result. Further description of program cells is provided in Sec. 3.2.

The *value* of constant cells, input cells, and program cells is always an integer[6]. Empty cells and comments cannot be referred to, so they do not contain a value.

Evaluation of "programs" is similar to the evaluation of formula cells in spreadsheet applications you may be familiar with, such as Microsoft Excel or Google Spreadsheets, where a formula may refer to other cells in the spreadsheet, combining them using a library of mathematical functions, resulting in a final value for the current cell. The programs in our cells can likewise refer to other cells of the spreadsheet, but will be written in an imperative programming style.

Here is an example of a spreadsheet:

	A	B	C	

1	constant
	`42`
		program `{`
		`  return A2 + A1`
		`} ensures value < 52`
			program `{`
			`  var s: Int := 0`
			`  s := s + B1`
			`  s := s - B3`
			`  return s`
			`}`	

2	user input `  value < 10`	comment `  hello world`		...

3	constant
	`5`
		program `{`
		`  return A2 + A3`
		`} ensures value < 15`		

		...		

This spreadsheet has two constant cells, `A1` and `A3`. The cell `A2` is a user input, which must be a number that is less than 10. The programs in cells `B1`

[6]Boolean-valued cells are an elective task.

and `B3` add the constants from cells `A1` and `A3`, respectively, to the user input. Both of the programs are associated with a postcondition on the result. The program in cell `C1` uses the results of the previous programs, but does not have a postcondition. The source code for this example can be found in the skeleton files, in the file `tests/required/04-sheet/example.spr`.

3.2 Programming language

Programs in program cells consist of local variable declarations, assignments, assertions, and arbitrarily nested conditional blocks. Comments consist of two forward slashes (`//`), and are ignored. Variable declarations, assignments, and conditions contain expressions, which include reading local variables, performing unary and binary operations, and reading the value of other cells.

The file `tests/required/04-sheet/syntax.spr` contains a program cell with all the statement and expression kinds accepted in the programming language, with explanations in comments and assertions demonstrating the expected behaviour.

Program cells may additionally have a postcondition, or be declared *transparent*. In your encoding (Sec. 3.5), program cells with postconditions must be treated *modularly*, *i.e.*, the postcondition of a cell should be sufficient to prove any properties in other program cells which use that cell. On the other hand, the specific implementation of a transparent program cell is known to all program cells which use it.

To demonstrate the difference, consider the following two spreadsheets:

	A
1	program { return 10 } ensures value < 20

2	program {
	  assert A1 == 10
	}

	A
1	program { return 10 } transparent

2	program {
	  assert A1 == 10
	}

In the above, the left spreadsheet *should not* verify: the cell `A2` can only assume that the value of cell `A1` is less than 20, but it cannot assume that the value is specifically 10. By contrast, the right spreadsheet *should verify*: because cell `A1` is declared as *transparent*, the cell `A2` can verify the assertion against the concrete implementation.

3.3 Well-formedness

A spreadsheet may or may not be well-formed. Well-formedness consists of the following properties:

- Every program in the spreadsheet must be syntactically well-formed and well-typed.
- Any reference to another cell refers to a cell that exists in the spreadsheet. For example, a spreadsheet with a cell referring to cell `A5` when there are only 4 columns is invalid. Any reference to a comment or empty cell is also invalid.
- The dependencies between cells, as expressed in their contained programs, must not form a cycle. For example, a spreadsheet with the cell `A1` containing a program which depends on cell `A2`, and the cell `A2` containing a program which depends on cell `A1`, is invalid.[7]

The syntax check in the first property is already implemented in the skeleton files; the well-typedness check is "implemented" by encoding the program into Viper. However, you will implement checks for the second and third properties.

3.4 Interpreter

Any well-formed spreadsheet can be *interpreted*, by providing values for all the user input cells of the spreadsheet and evaluating program cells. The interpreter is implemented in the skeleton files, and can be used as a reference for the behaviour of the spreadsheet language.

3.5 Encoding

The main goal of this project is to develop an *encoding* of spreadsheets into Viper programs. The resulting program should reflect the spreadsheet's behaviour, *i.e.*, express which values are expected in the user input cells, and assert that the program cells indeed satisfy their postconditions and assertions. Your encoding should be *sound*: if the Viper verifier verifies the output program, then the input spreadsheet must be correct. In this context, correctness is a combination of the following properties:

- Every program cell with a declared postcondition satisfies that postcondition with the value it returns.
- Every assertion in a program cell must pass.
- Invalid operations, such as division by zero, are not performed.

Your encoding should also be relatively complete, *i.e.*, an encoding which always indicates a verification error, while sound, is not very useful.

[7] This property is refined in one of the elective tasks, to account for *conditional* depen-

4 Project

For this project, you will be implementing an encoding of spreadsheets to Viper. In the skeleton files, you are given many parts of such an encoder, most importantly:

- `Spreadsheet/Ast.hs`: an abstract syntax tree (AST) definition for spreadsheets;
- `Spreadsheet/Lexer.hs`: a lexer for spreadsheets, which converts strings to token trees (defined in `Spreadsheet/Tokens.hs`);
- `Spreadsheet/Parser.hs`: a parser, which converts token trees to ASTs;
- `Viper/Ast.hs`: an AST definition for (a subset of) Viper;
- `Viper/Printer.hs`: a pretty-printer for Viper, which converts Viper ASTs to strings;
- `Interp/Interp.hs`: an interpreter, to serve as a reference for the semantics of the spreadsheet language; and
- `Main.hs`: the command-line interface and entrypoint for the project.

The missing part, to be implemented by you, is in `Encoder/Encoder.hs`. It should take as as input a spreadsheet AST and output a Viper AST. Modification of the skeleton files is allowed, especially when implementing extensions.

Implementations in other languages are allowed. You might not receive support with the implementation, especially if choosing a non-mainstream language. Your implementation must still exhibit the same input/output behaviour, and you cannot take shortcuts, such as using existing libraries to perform some of the tasks. However, using libraries for tasks which are already implemented in the skeleton files, such as parsing, is allowed.

4.1 Quick start

To compile the project, run this command from the project directory (which contains `Main.hs`):

```
ghc -o Project.o Main.hs
```

Once compiled, you can run the interpreter with:

```
./Project.o run <path/to/example.spr>
```

Other commands are explained if you run `./Project.o` without arguments.

dencies between cells.

4.2 Required tasks (35 points)

The required tasks for this project are:

- Implement the encoding of spreadsheet expressions.
- Implement the encoding of spreadsheet program cells.
- Implement the well-formedness checks described in Sec. 3.3.
- Implement the encoding of spreadsheets.

In the skeleton files, you are given the directory `tests`, containing spreadsheets. These files contain comments which explain various aspects of the code in detail. The `tests/required` directory contains test cases for the required tasks, organised further into subdirectories for each of the four tasks above. The recommended workflow is as follows:

- Use the lines in `tests/required/01-expr` and the `encode-expr` command to test your implementation of expression encoding.
- Use the file in `tests/required/02-program` and the `encode-program` command to test your implementation of program cell encoding.
- Use the files in `tests/required/03-wf` and the `encode` command to test your implementation of well-definedness checks.
- Use the files in `tests/required/04-sheet` and the `encode` command to test your implementation of full spreadsheet encoding.

Make sure to read the test case files, as they contain many comments explaining the syntax and intended functionality of the spreadsheets. You can also use the interpreter to quickly check how programs should behave, which assertions pass and which do not, *etc.* Test cases named `*.pass.spr` should successfully verify, ones named `*.fail.spr` should not.

4.3 Elective tasks (15 points)

Here are some *possible* extensions that you can implement on top of the required tasks. Note that you are not expected to implement all of the following. As a general guideline, we provide the approximate difficulty of each extension goal; aim for a total of 5 or more stars with your extensions. Some of the following require extensions of the parser and AST, but updating the interpreter is optional (though useful for debugging). Test cases for the elective tasks are provided in the `tests/elective` directory.

- ($\star$) Basic aggregation operations: sums and products over cell ranges. (See `tests/elective/aggregation`)

- ($\star$) User-defined aggregation operations.
  (See `tests/elective/aggregation-adv`)
- ($\star$) Boolean-typed cells.
  (See `tests/elective/bool-cells`)
- ($\star\star$) Iterated cell ranges: use an initial value and a repeatedly applied unary function to define the value of multiple cells. This is inspired by an "autofill" feature in existing spreadsheet software.
  (See `tests/elective/iterated`)
- ($\star\star$) More lenient cycle detection. Some spreadsheets contain cycles in the cell dependencies, but nevertheless interpreting these spreadsheets does not result in a cycle at runtime. Improve the cycle detection such that cycle detection is delegated to verification time.
  (See `tests/elective/cycles-adv`)
- ($\star\star$) Constant cell annotation: indicates that the value of a program cell must not depend on the value of user input cells. It should still be possible to *refer* to user input cells, as long as the result is the same.
  (See `tests/elective/const`)
- ($\star\star\star$) Linear user inputs: check that every user input affects the spreadsheet in some way. The encoding should prove that changing *any* user input cell changes the value of at least one program cell.
  Hint: research self-composition of programs.
  (See `tests/elective/linearity`)
- ($\star\star\star$) Indirect cell references, *i.e.*, using the value of a local integer variable to select *which* other cell in the spreadsheet should be read. Note that this should also be accompanied by an extended well-definedness check, to make sure that the indirect cell reference never goes out of bounds or refers to an invalid cell.
  (See `tests/elective/indirect-ref`)

We also encourage you to discuss other possible extensions to implement during the office hours.

4.4 Non-goals

We also note some things you are *not* expected to implement:

- Type checking. You may translate types and expressions directly to Viper and rely on Viper to spot type errors.
- User-friendly error reporting of any kind.