185.A05 Advanced Functional Programming SS 22

[1ex] Assignment 5
Model Checking Project MiniCheck
All Chapters

Add date here

Topic: Building an LTL Model Checker plus Implementing a Choice of Modelling/Tool support extension

Submission deadline: [.5ex]

add submission deadline here!

The goal of the project is to implement an automata-based LTL Model Checker, *MiniCheck*. The project consists of two parts: the core model checker and the elective modular extensions. In turn, the core model checker consists of a parser which builds an appropriate automaton from the input LTL formula and a cycle detection module which (dis)proves the input formula. The core is mandatory but is only worth a portion of the points achievable in this project.

In contrast to previous exercises, you are not asked to implement functions with given signatures. Instead, the functionality of the executed program is evaluated. You will have to design your own function signatures and data types and decide which data structures you want to use.

The program should not be written in a single file, but as a `stack` or `cabal` project consisting of multiple modules. It is encouraged to use custom and predefined data structures, as well as external libraries and Haskell language extensions **where appropriate**. It is not required to get the project to run on g0.

There will be a Q&A session where help will be provided in case there are troubles with the selected technologies. The date will be announced via TISS news.

1 Grading

The parser and the cycle detection module of MiniCheck are mandatory. Implementing the core awards up to 200 points. The project presentation will be worth up to 100 points. To gain more points on top of the aforementioned, you can choose from some modelling and verification extensions of the MiniCheck tool.

For a positive evaluation of this project at least 200 points are required. A positive grade on the project is necessary in order to receive a positive grade in this course. The total points for the project are calculated by adding up the points on the core, the selected extensions and the submission talk. A positive grade on the core itself is *not* required. At most 300 points are awarded for the project implementation.

In contrast to previous exercises, there is no second submission.

1.1 Note

Some previous knowledge in the field helps, but it's not mandatory. This includes also:

- Automata theory
- Graph Algorithms

2 MINI Core (200P)

2.1 Informal Description

This section informally describes the core of the MINI programming language, only covering the language core.

MINI is a procedural programming language. The syntax and semantics of MINI are inspired by the programming language C.

All code is contained in a single procedure `main` and variables are of the same type, namely the arbitrarily sized integer (Haskell type `Integer`). This means that the type of the variables is not explicitly specified in MINI. Variables are not required to be declared before definition.

The procedure `main` implements a pure function, input/output functionality is not included in the core of MINI. Procedure `main` takes a defined set of named variables and returns a result. The last statement of the procedure must be a `return` statement and no other `return` statement can occur in the procedure. In contrast to common programming languages, `return` statements in MINI are syntactically restricted to return the value of a single variable instead of a general expression.

The `while` and `if` statements are defined as in C: the `while` statement implements a loop that executes the given block in a loop as long the test-expression evaluates to true. The `if` statement executes either the first or optionally the second block.

It is not necessary to declare variables before usage, however, it is necessary to define their value before usage, i.e. a variable must be initialised before it can be used in arithmetic and boolean expressions. Variables declared in the argument list are initialised with the values the procedure has been invoked with.

2.2 Syntax

$$
\begin{aligned}
\mathit{program} &:= \texttt{procedure main(}\mathit{argument\_vars}\texttt{)\{}\mathit{procedure\_body}\texttt{\}}\\
\mathit{argument\_vars} &:= \mathit{var} \mid \mathit{var}\texttt{,}\mathit{argument\_vars}\\
\mathit{procedure\_body} &:= \mathit{statementsreturn\_stmt}\\
\mathit{statements} &:= \varepsilon \mid \mathit{statementstatements}\\
\mathit{statement} &:= \mathit{while\_stmt} \mid \mathit{if\_stmt} \mid \mathit{assign\_stmt}\\
\mathit{while\_stmt} &:= \texttt{while(}\mathit{bool\_expr}\texttt{)\{}\mathit{statements}\texttt{\}}\\
\mathit{if\_stmt} &:= \texttt{if(}\mathit{bool\_expr}\texttt{)\{}\mathit{statements}\texttt{\}}\\
&\mid \texttt{if(}\mathit{bool\_expr}\texttt{)\{}\mathit{statements}\texttt{\}else\{}\mathit{statements}\texttt{\}}\\
\mathit{assign\_stmt} &:= \mathit{var}\texttt{=}\mathit{int\_expr}\texttt{;}\\
\mathit{int\_expr} &:= \mathit{int\_expr\_nested} \mid \texttt{-}\mathit{int\_exp\_nested}\\
&\mid \mathit{int\_expr\_nestedoperatorint\_expr\_nested}\\
\mathit{int\_expr\_nested} &:= \mathit{int} \mid \mathit{var} \mid (\mathit{int\_expr})\\
\mathit{operator} &:= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}\\
\mathit{bool\_expr} &:= \mathit{int\_exprrelatorint\_expr}\\
\mathit{relator} &:= \texttt{==} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{<} \mid \texttt{>}\\
\mathit{return\_stmt} &:= \texttt{return}\mathit{var}\texttt{;}\\
\mathit{int} &:= \mathit{digit} \mid \mathit{digitint}\\
\mathit{var} &:= \mathit{ident}\\
\mathit{ident} &:= \mathit{charident\_rest}\\
\mathit{ident\_rest} &:= \varepsilon \mid \mathit{ident\_charident\_rest}\\
\mathit{ident\_char} &:= \mathit{char} \mid \mathit{digit} \mid \texttt{\_}\\
\mathit{char} &:= \texttt{a} \mid \ldots \mid \texttt{z}\\
\mathit{digit} &:= \texttt{0} \mid \ldots \mid \texttt{9}
\end{aligned}
$$

Additional whitespace is allowed everywhere except within identifier names, integer literals, the binary comparison operators, and the keywords (`procedure`, `return`, `if`, `else`, `while`).

2.3 Semantics

The value returned by the procedure $p\texttt{(}v_1, \ldots, v_n\texttt{)\{}s\texttt{\}}$, called with the values $x_1, \ldots, x_n$, is defined as $\mathcal{V}[s](\{v_1 \mapsto x_1, \ldots, v_k \mapsto x_k\})$.

The function $\mathcal{V}$ evaluates a given snippet of code (first argument in square brackets) on a given program state (second arguments, in parentheses). The state of a running program can be described by a mapping from all variables to their values.

$$\mathcal{V}[s_1 s_2](\sigma) := \mathcal{V}[s_2](\mathcal{V}[s_1](\sigma)) \text{ if } s_1 \text{ is a statement} \tag{1}$$

$$\mathcal{V}[\texttt{while(}b\texttt{)\{}s\texttt{\}}](\sigma) := \begin{cases} \mathcal{V}[\texttt{while(}b\texttt{)\{}s\texttt{\}}](\mathcal{V}[s](\sigma)) & \text{if } \mathcal{V}[b](\sigma) \\ \sigma & \text{otherwise} \end{cases} \tag{2}$$

$$\mathcal{V}[\texttt{if(}b\texttt{)\{}s\texttt{\}}](\sigma) := \begin{cases} \mathcal{V}[s](\sigma) & \text{if } \mathcal{V}[b](\sigma) \\ \sigma & \text{otherwise} \end{cases} \tag{3}$$

$$\mathcal{V}[\texttt{if(}b\texttt{)\{}s_1\texttt{\}else\{}s_2\texttt{\}}](\sigma) := \begin{cases} \mathcal{V}[s_1](\sigma) & \text{if } \mathcal{V}[b](\sigma) \\ \mathcal{V}[s_2](\sigma) & \text{otherwise} \end{cases} \tag{4}$$

$$\mathcal{V}[v\texttt{=}e\texttt{;}](\sigma) := \sigma\{v \mapsto \mathcal{V}[e](\sigma)\} \tag{5}$$

$$\mathcal{V}[\texttt{return }v\texttt{;}](\sigma) := \sigma(v) \tag{6}$$

$$\mathcal{V}[e_1\texttt{==}e_2](\sigma) := \mathcal{V}[e_1](\sigma) = \mathcal{V}[e_2](\sigma) \tag{7}$$

$$\mathcal{V}[e_1\texttt{!=}e_2](\sigma) := \mathcal{V}[e_1](\sigma) \neq \mathcal{V}[e_2](\sigma) \tag{8}$$

$$\mathcal{V}[e_1\texttt{<=}e_2](\sigma) := \mathcal{V}[e_1](\sigma) \leq \mathcal{V}[e_2](\sigma) \tag{9}$$

$$\mathcal{V}[e_1\texttt{>=}e_2](\sigma) := \mathcal{V}[e_1](\sigma) \geq \mathcal{V}[e_2](\sigma) \tag{10}$$

$$\mathcal{V}[e_1\texttt{<}e_2](\sigma) := \mathcal{V}[e_1](\sigma) < \mathcal{V}[e_2](\sigma) \tag{11}$$

$$\mathcal{V}[e_1\texttt{>}e_2](\sigma) := \mathcal{V}[e_1](\sigma) > \mathcal{V}[e_2](\sigma) \tag{12}$$

$$\mathcal{V}[e_1\texttt{+}e_2](\sigma) := \mathcal{V}[e_1](\sigma) + \mathcal{V}[e_2](\sigma) \tag{13}$$

$$\mathcal{V}[e_1\texttt{-}e_2](\sigma) := \mathcal{V}[e_1](\sigma) - \mathcal{V}[e_2](\sigma) \tag{14}$$

$$\mathcal{V}[e_1\texttt{*}e_2](\sigma) := \mathcal{V}[e_1](\sigma) * \mathcal{V}[e_2](\sigma) \tag{15}$$

$$\mathcal{V}[e_1\texttt{/}e_2](\sigma) := \mathcal{V}[e_1](\sigma) / \mathcal{V}[e_2](\sigma) \tag{16}$$

$$\mathcal{V}[\texttt{-}e](\sigma) := -\mathcal{V}[e](\sigma) \tag{17}$$

$$\mathcal{V}[\texttt{(}e\texttt{)}](\sigma) := \mathcal{V}[e](\sigma) \tag{18}$$

$$\mathcal{V}[v](\sigma) := \sigma(v) \text{ if } v \text{ is a variable name} \tag{19}$$

$$\mathcal{V}[i](\sigma) := \text{int}(i) \text{ if } i \text{ is a sequence of digits} \tag{20}$$

We use the notation $\sigma\{v \mapsto x\}$ to denote the state that is obtained when updating the value of the variable v to x. The expression $\sigma(v)$ describes the value of the variable v in state σ. The operation / describes integer division. The function int converts a sequence of digits into an integer based on its natural base-10 interpretation. Integer types have arbitrary size (Haskell type `Integer`).

The interpreter aborts the execution of invalid programs. A program is invalid in one of the following cases:

- A variable is not defined but used in an expression.
- Division by 0.
- The main procedure is called with an incorrect number of arguments.

2.4 Examples

A procedure calculating the difference of two numbers:

```
procedure main(a, b) {
    c = a - b;
    return c;
}
```

A procedure calculating the minimum of two numbers:

```
procedure main(a, b) {
    if (a < b) {
        c = a;
    } else {
        c = b;
    }
    return c;
}
```

A procedure calculating the greatest common divisor for non-negative integers:

```
procedure main(a, b) {
    r = b;
    if (a != 0) {
        while (b != 0) {
            if (a < b) {
                b = b - a;
            } else {
                a = a - b;
            }
        }
        r = a;
    }
    return r;
}
```

2.5 Tasks

The project implementation should consist of the parts listed in this section.

Abstract Syntax Tree Design an appropriate datatype for describing MINI programs.

Parser Implement a function that parses the source code and computes an abstract syntax tree of the program.

You are allowed to choose one of the parsing approaches presented in the lecture or use an external monadic parsing library from Hackage. Document your choice.

Interpreter The interpreter function executes the main procedure with the given command-line arguments. It is assumed that the MINI procedure has already been parsed into an abstract syntax tree as described by the MINI semantics before execution.

You have to choose an approach to handle the state of the running program. There are several approaches available. The following list describes some, not all, of the approaches. Document your choice.

- continuation: the interpreter function calls itself recursively with its updated state and remaining program
- monadic: use the state monad presented in the lecture or the state monad as defined in the `transformers` library
- transformer: use monad transformers as defined by the `transformers` and `mtl` libraries (advanced)
- effect: use an effect library to handle state (advanced)
  - `polysemy`
  - `freer-simple`

The Main Function Instead of specifying a single function that should be implemented, the behavior of the running binary is given: The program should read the command-line arguments and interpret the first argument as the path of the source file. The program source code is parsed and interpreted using the remaining command-line arguments (parsed as integers) as procedure arguments.

The program should abort with a nonzero exit code in case of errors such as:

- missing command-line arguments,
- format errors of command-line argument integers,
- parsing errors,
- interpretation errors.

Test Suite Projects of this size profit from an automatic test-suite. Write a test-suite for the major components of the project using one of the following frameworks:

- `hspec`, a tutorial can be found `here`
- `tasty`
- `HTF`
- `HUnit`

As before, it is encouraged to utilise external libraries for writing tests.

It is required to write at least three unit tests per major component, where major components are:

- MINI parser
- MINI interpreter

Additionally, use QuickCheck property tests to verify correctness for programs written in MINI against an equivalent Haskell implementation. Write your own MINI programs for solving the following numerical problems:

- Given a parameter `n`, calculate the nth Fibonacci number.
- Check for a given parameter `n` whether it is a prime number. Efficiency is not a concern.
- Given two positive nonzero numbers `a` and `b`, calculate their least common multiple.

3 MINI Extensions

This section describes some extensions to MiniCheck. Each extension is worth the given amount of points. Note that none of the described extensions are necessary for a positive grade.

3.1 Parsing Mini Programs (100 P)

write me!

MINI core programs only consist of a single procedure, named `main`. However, this is unwieldy as you can not re-use code. This extension allows multiple *named procedures* that can be called from any other procedure and can also be used to implement *recursion* patterns.

Syntax The following grammar rules are added/updated:

$$\mathit{program} := \mathit{main\_procedureprocedures} \tag{21}$$

$$\mathit{main\_procedure} := \texttt{procedure main(}\mathit{argument\_vars}\texttt{)\{}\mathit{procedure\_body}\texttt{\}} \tag{22}$$

$$\mathit{procedures} := \varepsilon \mid \mathit{procedureprocedures} \tag{23}$$

$$\mathit{procedure} := \texttt{procedure}\mathit{ident}\texttt{(}\mathit{argument\_vars}\texttt{)\{}\mathit{procedure\_body}\texttt{\}} \tag{24}$$

$$\mathit{call\_expr} := \mathit{ident}\texttt{(}\mathit{argument\_list}\texttt{)} \tag{25}$$

$$\mathit{argument\_list} := \mathit{int\_expr} \mid \mathit{int\_expr}\texttt{,}\mathit{argument\_list} \tag{26}$$

The option *call_expr* is added to *int_expr_nested*

If the extension “IO System” is also implemented, `read_int` and `print_int` are invalid procedure names.

Semantics The following rule is added to the definition of the program evaluation function $\mathcal{V}$.

$$\mathcal{V}[p(x_1, \dots, x_k)](\sigma) := \mathcal{V}[s](\{v_1 \mapsto x_1, \dots, v_k \mapsto x_k\}) \tag{27}$$
$$\text{if } p(v_1, \dots, v_n)\{s\} \text{ is a defined procedure} \tag{28}$$

The interpreter aborts the execution if an unknown procedure is called or an incorrect number of arguments is given.

Tasks

- Add the definition and invocation of named procedures to the abstract syntax tree.
- Modify the parser to correctly parse named procedure definitions and invocations.
- Modify the interpreter to correctly call named procedures.
- Provide test-cases for your parser and interpreter to showcase the functionality. Use QuickCheck if possible.
- Implement a simple recursive function in MINI and add the program to your test suite.
- Change the implementation of your gcd test-procedure to use a named procedure that calls itself recursively.

3.2 IO System (50P)

Modern programming languages usually have a way to interact with a user via terminal or file system. This extension explores simple console interaction.

The extension adds two new statements: one that prints an integer and one that reads an integer. *Note*: By using a separate statement for reading integers, we avoid having multiple read-calls within a nesting expression.

Synatx The following grammar rules are added:

$$\textit{print\_int\_stmt} := \texttt{print\_int(}\textit{int\_expr}\texttt{);} \tag{29}$$
$$\textit{read\_int\_stmt} := \textit{var}\texttt{=read\_int();} \tag{30}$$

The options *print_int_stmt* and *read_int_stmt* is added to *statement*.

Semantics Side effects are not easily expressible with the evaluation function $\mathcal{V}$. Therefore, this feature is only described informally.

- The statement `print_int(`e`);` evaluates e in the current state and prints the value as formatted signed integer to the standard output stream, followed by a newline character (‘
  n’).
- The statement var`=read_int();` reads a line from standard input, converts the input into a number and then assigns the value to the given variable. If the input string is invalid or no further line can be read, the interpreter aborts the execution.

Tasks

- Add the IO statements to the abstract syntax tree
- Modify the parser to correctly parse IO statements.
- Modify the interpreter to correctly execute IO statements.
- Provide test-cases for your parser to test the syntax extension.

Bonus Task: Stubbable IO in tests (10P) Write test cases that test the functionality of the new IO statements of the interpreter. This should be done without having to access the program's standard input or output streams. Instead, the interpreter should be instructed to read integers defined by a test-case and capture written integers to be used in assert-statements at the end of the test.

3.3 Source Code Formatting (50P)

A common format of the source-code is essential for collaborative work on a project, as it improves readability and may help keeping new changes as small as possible. For our language, we want a single format to layout all our programs. The exact format is up to you, but with the following requirements:

- Code blocks must be indented, and all statements within a block must be on the same indentation level. Nested code blocks must be more deeply indented than their parent block.
- Introduce line-breaks after statements.

Implementation Suggestions You may use special purpose libraries (we provide an example below) for this task or provide your own implementation. Note, when you use a library, you still need to be able to explain the high-level theory of it.

- `Chapter 17 Pretty Printer`
- `prettyprinter`

Tasks

- Implement a source code formatting function.
- Implement support for a command-line flag `--format` that is expected before the filename.
- Use QuickCheck to test the "round-trip" of pretty-printing and parsing your program:
  `program -> parse (pretty program) == program`

3.4 Sensible Error Messages (50P)

When the parser fails in any way, it usually prints something that is hard to decipher for anyone who hasn't written the parser.

In this extension, you will provide better error messages, i.e. messages that contain the error code and explain roughly what went wrong in at least the following cases:

- E01: Mismatched parentheses,
- E02: Unknown keyword,
- E03: Missing semicolon,
- E04: invalid identifier (starting with digit, e.g. `12ab` is not a valid identifier and not a valid number),
- E05: missing return statement at the end of a procedure.

For all other parse errors, the error code E00 can be used.

It is not required to provide any specific diagnostics, but it needs to be apparent which error case occurred and what snippet of source code causes the issue.

Moreover, implement a graceful crash message reporting for the interpreter. The following crash cases are to be reported:

- C01: variable used before definition,
- C02: division by 0,
- C03: procedure is called with the incorrect number of arguments,
- C04: unknown procedure called (optional, 1pt),
- C05: invalid input string on `read_int` (optional, 1pt),
- C06: `eof` reached when calling `read_int` (optional, 1pt).

4 Bonus Task: Project Management (25P)

There is more to a successful project than just writing code. Usually, you also need to write proper documentation, distribute it, etc...

In this section, you will try some project management mechanisms for Haskell: In particular, you will provide documentation for your program, find the coverage of your test-suite and learn more about basic profiling with GHC.

4.1 Command-Line Interface

The program so far only has very rudimentary argument parsing, allowing a single filepath, and arguments for that file. However, anyone who did not write this program has no idea about how to invoke this program correctly, thus, we want to have a proper command-line interface

Implementation Suggestions Common libraries for such tasks are `optparse-applicative` and `cmdargs` but it is also valid to not use any libraries at all and design your own solution.

Required Flags The program should be able to understand the following flags:

- On `-h`/`--help`, a help message should be displayed, explaining how the program can be invoked correctly.
- On `--sat`, the input code .
- On `--extensions`, a list of supported MINI extensions is printed.
- Other flags as you see appropriate. (optional. not graded)

4.2 Documentation

Document the most important types and functions of your project `haddock`-conformly and provide the documentation via HTML.

Helpful commands:

- `cabal haddock` for building the documentation and inspecting it locally.
- `cabal haddock --haddock-for-hackage` for building a `.tar.gz` containing the documentation.
- `stack haddock --open`: builds documentation and opens it in the browser upon completion.

4.3 Test Coverage

Generate the test-coverage of your program's test-suite. Discuss your findings and investigate any unexpected results.

Helpful commands:

- `cabal test --enable-coverage`
- `stack test --coverage`

4.4 Profiling

Being able to profile your code is of great importance in real-world projects. Thus, we want you to experiment with some profiling in Haskell.

To do that, you might have to build your project in profiling mode:

- `cabal build --enable-profiling`
- `stack build --profile`

Then you can pass RTS arguments to the GHC program to obtain run-time information, such as which function you spend the most amount of time in, etc...

Refer to the GHC documentation for the exact `profiling flags` to obtain relevant information.

Answer the following questions:

- What is the memory usage over time?
- What is the peak memory usage?
- Which function is the most time spent in?
- Which type requires the most amount of memory?

5 Submission artefacts

You should submit a zip-archive containing all project source files and a PDF with detailed project documentation in your group submission directory.

5.1 Project Implementation

The project should be written as a `cabal` or `stack` project consisting of multiple modules.

5.2 Test Suite

Unit tests and property tests need to be submitted as well, and it must be possible to run the whole test-suite with either `cabal test` or `stack test`.

5.2.1 Project Documentation

The project documentation pdf should cover at least these topics and explain your choices.

- Which project build tool is used for the project? (`cabal` or `stack`)
- Which GHC version is used?
- How can the program binary be built? How can it be run?

- Which libraries are included as dependencies and which Haskell language extensions are enabled?
- Which Framework and libraries are used for writing tests?
- Which MiniCheck extensions are implemented?
- How is the functionality partitioned into different modules?
- How do you test your program? Which parts are the focus of your tests? Do there exist parts of the code that cannot be tested?
- Are there known issues and limitations of your program?