185.A05 Advanced Functional Programming SS 22

Monday, 05/02/2022 [1ex] **Assignment 5**

Model Checking Project: MiniCheck

All Chapters

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

Submission deadline: Friday, 06/13/2022 [.5ex]

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.
- a cycle detection module which (dis)proves the formula on a given Transition System.

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 modules 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

We assume a general knowledge of model checking and its purpose, as well as basis of logics, such as propositional logic. Besides this, some previous knowledge in related fields helps and makes the project smaller, but it's not mandatory. This includes also:

- Automata Theory
- Graph Algorithms

2 MiniCheck Core (200P)

Model Checking consists of formally and exhaustively verifying a formula against all the possible executions of a particular type of automata. The automata-based approach we are going to follow has been originally proposed by Vardi and Wolper It has been proposed originally by Wolper and Vardi [**VarWo**]. In the following, we provide some informal information about the type the formalisms you are required to support, and then present the project to carry out. For a complete and deeper introduction to our approach in model checking, we refer the reader to [1]. You can find a PDF version on the Github repository.

2.1 Preliminaries - Transition Systems

As modelling formalism, we use *Transition Systems* (TS). They are a particular variant of Finite State Automata which allow to describe adequately both hardware and sofware systems. They are defined over infinite runs, and they do not have a set of final states. Formally, a TS is a tuple $(S, Act, \rightarrow, I, AP, L)$, where

- S is a set of states,
- Act is a set of actions,
- $\longrightarrow \subseteq S \times Act \times Act$ is a transition relation,
- AP is a set of atomic propositions, and
- $L : S \rightarrow 2^{AP}$ is a labelling function

Actions are used to specify how the system evolves from one state to another. The labelling function maps every state to the set of atomic propositions holding in that state. The initial state is chosen nondeterministically between all states $\in I$. We adopt a **state-based** approach: we consider (and want to verify) the labels in the state sequence of a run of the TS, and abstract from actions.

Note that, for simplicity sake, we consider only TS with no terminal states, i.e., every state has at least one outgoing edge. While this prevents some technical problems from ocurring, it is not a limit to the expressive power of TS: for every state s without outgoing edge, define a transition to a sink state s_{sink}, and then define a self transition from s_{sink} to itself.

2.1.1 Resources

[1, Paragraph 2.1]

2.1.2 Task

Define:

- a suitable plain-text representation of TS to provide as input to out tool, and
- a data type to represent them.
- the type of the set of atomic propositions that can be used to label states of the Transition System.

Implement:

- A parsing function from the plain-text representation to the data type.

Take care of making sure that there are no terminal states, that there is at least an initial state, and that your formalisation of the set of atomic proposition is respected.

Add to the helper function the parsing of only this, and giving back results in case of failure

2.2 Preliminaries - Linear Temporal Logic

To express properties to verify on TS, we consider a temporal extension of propositional logic, i.e., the introduction of temporal modalities on top of propositional logic. In particular, the temporal modalities we consider are $\bigcirc$ (pronounced "next") and $\mathcal{U}$ (pronounced "until").

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.

2.3 Syntax

2.4 Semantics

2.5 Examples

2.6 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:

- MiniCheck formulas parser
- MiniCheck cycle detection module

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!

3.2 Returning Counterexamples (50P)

write me!

3.3 An *On-the-fly* implementation (50P)

write me!

4 Bonus Task: Satisfiability Checking (10P)

On `--sat` flag, the Model Checker should construct the FSA starting from the input formula and perform the fair-cycle detection module on **this automaton only**. This procedure amounts at verifying whether the formula is satisfiable at all by any string.

It is up to you to decide whether changing the format of the input file (e.g., by forbidding to include the Transition System in the file).

5 Bonus Task: Project Management (25P)

evaluate whether this should be removed

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.

5.1 Command-Line Interface

The program so far only has very rudimentary argument parsing, allowing a single filepath. However, anyone who did not write this program has no idea about how to invoke the Model Checker 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 `--extensions`, a list of supported MINI extensions is printed.
- Other flags as you see appropriate. (optional. not graded)

5.2 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.

5.3 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`

5.4 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?

6 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.

6.1 Project Implementation

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

6.2 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`.

6.2.1 Project Documentation

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

Add some appropriate questions here

- 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?

References

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN: 026202649X.