Programming Languages: Project 1

This document describes the first project for the course Programming Languages (2023/24).

This file might be updated before the deadline (**last changed 25 Apr 2024**). Any changes that occur will be announced in Slack.

Quick Summary

- Goals:
    1. Extend a simple imperative language with non-deterministic choice and a new type of conditional guard
    2. Define the semantics of the extended language in two different ways
    3. Use the semantics to formally prove some relevant properties
- Deadline: **10 May 2024**
- To be done in groups of **three students**
- Submission is via Fénix (see instructions below)
- If you have any questions, please do not hesitate to contact João or Henrique. You are encouraged to ask questions in the course's Slack (channel `#projects`)!

Tasks

0. Download the Project Pack from Fénix

You must work on the provided project pack (`P1-PL.zip`). The pack contains files that were taken/adapted from the Software Foundations book (Volume 1):

- `Maps.v`: defines total and partial maps
- `Imp.v`: defines the simple imperative language that will be extended
- `Interpreter.v`: this is where a step-indexed evaluator and proofs using it should be defined
- `RelationalEvaluation.v`: this is where the operational semantics should be defined (as a relation)
- `ImpParser.v`: defines a parser for the language
- `Extraction.v`: extraction to OCaml (same as in book)
- `interpreter.ml`: simple interpreter written in OCaml
- `examples/*.lpro`: examples of programs that can be executed using the interpreter
- `README.md`: file that you should fill in with your group's information and contributions
- `INSTRUCTIONS.md`: this file

You should work on the files `Imp.v`, `Interpreter.v`, `RelationalEvaluation.v`, and `ImpParser.v`. Apart from the parser, all other files are marked with TODO comments to identify the places where you are supposed to work. You are also required to add information to the `README.md` file.

You can use the Makefile to compile the project (but note that it will only compile after you complete some of the tasks). If you add new files, you should edit the file `_CoqProject` and regenerate the Makefile (see the official documentation)

1. Extend Imp

You are required to extend the Imp language with two new constructs: non-deterministic choice and a new conditional guard that backtracks to alternative choices.

Non-deterministic choice You are required to extend the Imp language with a new binary operator that allows non-deterministic choice between two programs. If we denote that operator as `!!`, then `c1 !! c2` is a program that chooses non-deterministically between `c1` and `c2`. For example, after executing the program `x`

`:= 1 !! x := 2`, the value of variable `x` can be either `1` or `2` (they are both possible, but only one of them is chosen).

A new conditional guard You are required to extend the Imp language with a new guard operator `b -> c`, where `b` is a test and `c` is a command (i.e., a sub-program). Its semantics is as follows: if `b` is true, then `c` should be executed; if `b` is false, the program should backtrack to the point of the last non-deterministic choice and choose a different option. For example, consider the program `(x := 1 !! x := 2); x=2 -> skip`. If the non-deterministic choice chooses `x := 1`, then the guard will be false; therefore, the program should backtrack and choose `x := 2`, in which case the guard is true and `skip` is executed. On termination, the value of variable `x` must be 2. Note that executing `(x := 1 !! x := 2); x=2 -> skip` is the same as executing `x := 2`.

On the other hand, the program `(x := 1 !! x := 3); x=2 -> skip` will fail. If the non-deterministic choice chooses `x := 1`, then the guard will be false; therefore, the program should backtrack and choose the assignment `x := 3`. However, in that case the guard will still false and since there is no other choice to be explored, the program fails to execute.

Tasks You should edit the file `Imp.v` and perform the following tasks:

1. Extend the datatype `com` with the two new constructs described above
2. Define a new notation for the new constructs
3. Define examples `p1` and `p2` as specified (they correspond to the programs shown above)

2. A Step-Indexed Evaluator

2.1. Implementation Extend the step-indexed evaluator `ceval_step` defined in the file `Interpreter.v` to evaluate a program `c` given as parameter. In particular, you are required to use the following type for `ceval_step`:

```
Fixpoint ceval_step (st : state) (c : com) (continuation: list (state * com)) (i : nat)
                    : interpreter_result
```

The interpreter receives as parameters an initial state `st`, a program `c`, a list of possible `continuations`, and an index `i` that limits the number of execution steps.

The type of the result is provided:

```
Inductive interpreter_result : Type :=
  | Success (s: state * (list (state*com)))
  | Fail
  | OutOfGas.
```

The result can either be successful (in which case we have a final state and, potentially, a list of continuations that were never explored); it can fail (e.g., if a conditional guard is false and there are no continuations to explore); and it can "run out of gas", which happens when the index given is not large enough for the program to terminate. For example, if the parameter `i` is 0, the function should return `OutOfGas`.

Assuming that the index is large enough to run the program `x:=1; x=2 -> y:=1` until termination, the interpreter should return `Fail`, since the guard is false and there are no options/continuations to explore.

Assuming that the index is large enough to run the program `(x := 1 !! x := 2); x=2 -> skip` until termination, the interpreter should return `Success (s,q)`, where `s` is the state where the value of `x` is `2` and `q` is empty if `x:=1` was chosen first or non-empty if `x:=2` was chosen first.

IMPORTANT NOTE: To simplify your solution, you can assume that some programs will not behave as one would expect (e.g., programs with non-deterministic choices and conditional guards inside while loops or certain patterns that mix non-deterministic choice, conditional guards, and sequential composition). **The**

only requirement is that your semantics makes sense for programs similar to the examples shown and it is strong enough to prove all the properties required in this project.

2.2. Properties You are required to prove the two properties stated without proof in the file `Interpreter.v`: `p1_equals_p2` and `ceval_step_more`.

The ImpCEvalFun chapter might guide you on how to implement the interpreter and how to structure your own proofs.

Add a succint comment before each property explaining the property in your own words.

3. Relational Evaluation

You are required to define a relational semantics for the extended Imp language (the `ceval` relation). The semantics is similar to the relational semantics shown in the Imp chapter, but here we deal with executions that can backtrack and follow alternative continuations, and we deal with continuations that can fail.

We'll use the notation `st1 / q1 =[ c ]=> st2 / q2 / r` for the `ceval` relation: it means that executing program `c` in a starting state `st1` with continuations `q1` results in an ending state `st2` with unexplored continuations `q2`. Moreover the result of the computation will be `r`, which is either `Success` or `Fail`:

```
Inductive result : Type :=
  | Success
  | Fail.
```

3.1. Proving concrete examples Use the new relational semantics to prove the examples `ceval_example_if`, `ceval_example_guard1`, `ceval_example_guard2`, `ceval_example_guard3` and `ceval_example_guard4`.

Note: The examples `ceval_example_guard3` and `ceval_example_guard4` are the same because the idea is to prove them using different choices. In one of them, one should choose first the first command `X:=1` (backtracking required) and in the other one should choose `X:=2` (no backtracking required).

3.2. Properties of the new constructs

The file `RelationalEvaluation.v` contains a definition of so-called *behavioral equivalence* (`cequiv`), denoted with the infix symbol `==`. We say that two programs `c1` and `c2` are equivalent if and only if `c1 == c2`.

We use this equality to state a few interesting and useful properties of non-deterministic choice. You are required to prove that these properties hold.

First, we start with some concrete examples that involve conditional guards:

```
Lemma cequiv_ex1:
  <{ X := 2; X = 2 -> skip }> == <{ X := 2 }>.

Lemma cequiv_ex2:
  <{ (X := 1 !! X := 2); X = 2 -> skip }> == <{ X := 2 }>.
```

Second, we state that the non-deterministic choice is idempotent, commutative, and associative:

```
Lemma choice_idempotent: forall c,
  <{ c !! c }> == <{ c }>.

Lemma choice_comm: forall c1 c2,
  <{ c1 !! c2 }> == <{ c2 !! c1 }>.

Lemma choice_assoc: forall c1 c2 c3,
  <{ (c1 !! c2) !! c3 }> == <{ c1 !! (c2 !! c3) }>.
```

Next, we state that sequential composition distributes (on the left) over non-deterministic choice:

```
Lemma choice_seq_distr_l: forall c1 c2 c3,
  <{ c1 ; (c2 !! c3)}> == <{ (c1;c2) !! (c1;c3) }>.
```

Finally, the following congruence property holds:

```
Lemma choice_congruence: forall c1 c1' c2 c2',
  c1 == c1' -> c2 == c2' ->
  <{ c1 !! c2 }> == <{ c1' !! c2' }>.
```

4. Standalone Interpreter

Create a standalone interpreter for your extension of Imp by extending the parser in `ImpParser.v`. We already provide a basic OCaml program that you can use (`interpreter.ml`). It assumes that the result of computations are stored in a variable `res` (the value of this variable is shown by the interpreter).

To compile the interpreter, you need to compile the file `Extraction.v` and then you can run the following command:

```
ocamlc -o interpreter imp.mli imp.ml interpreter.ml
```

You can then store programs in files and execute them. For example:

```
$ ./interpreter --help
interpreter <file.lpro> [-n interpreter_steps]
  -n Number of steps
  -help  Display this list of options
  --help  Display this list of options

$ cat examples/ex1.lpro
(x := 1 !! x := 2); x = 2 -> res := 42

$ ./interpreter examples/ex1.lpro
42

$ ./interpreter examples/ex1.lpro -n 5
Still running after 5 steps

$ ./interpreter examples/ex1.lpro -n 6
42

$ cat examples/ex2.lpro
(x := 1 !! x := 2); x = 3 -> res := 42

$ ./interpreter examples/ex2.lpro
Failed!
```

It might be helpful to read the chapters on Parsing and Extraction.

5. Extras

You are encouraged to extend your work with more features. In terms of grades, the extensions might only be considered if everything else was attempted.

Here are some suggestions for extra features:

1. Improve the step-indexed evaluator so that: i) when it fails, instead of just returning `OutOfGas` or `Fail`, it returns an appropriate error message; ii) when it succeeds, it shows the resulting state and

continutation, but also the number of "steps" taken. You can also adapt the standalone interpreter to also show this information.

2. Explore the semantics involving while loops, non-deterministic choice and conditional guards. You can start by writing some interesting examples and checking whether your formal semantics matches your intuition. This might lead you to improve or change your semantics.
3. Imperative languages often include a `break` or similar statement for interrupting the execution of loops. Extend the language Imp with a `break` construct. Extend the semantics, the step-indexed evaluator, and adapt all the relevant properties. (See also the exercise `break_imp` in the Imp chapter.) You can also extend the standalone interpreter.
4. We have seen simple transformations and optimizations in the lectures that can also be applied to Imp. Implement a few optimizations that you find interesting and prove them correct with respect to the semantics defined.
5. Be as creative as you want: improve the language or standalone interpreter in any way that you see fit!

Submission

The project is due on the **10th of May, 2024**. You should follow the following steps:

- Submit only one file per group. Make sure your submitted file is named `P1-PL-GNN-2024.zip`, where `NN` is the group number. Always use two digits (e.g., Group 8's submitted file should be named `P1-PL-G08-2024.zip`).
- `PL-P1-GNN-2024.zip` is a zip file containing the solution and a `README.md` file where all group members and contributions are identified.
- Upload the file to Fénix before the deadline.

Assessment

To assess your submission, the following grid will be used:

Task	Marks (max)
README file properly filled in	0,25
Task 1 (Imp.v)	
Extend `com`	1
New notation	0,5
Examples `p1` and `p2`	0,5
Task 2 (Interpreter.v)	
Implementation of step-indexed evaluator	3
Proof of `p1_equals_p2`	1
Proof of `ceval_step_more`	2,5
Task 3 (RelationalEvaluation.v)	
Definition of `ceval`	3
Proof of the examples `ceval_example_*`	2,5
Proof of `cequiv_ex1` and `cequiv_ex2`	1,5
Idempotence	0,5
Commutativity	0,5
Associativity	0,5
Distributivity (left)	0,5
Congruence	0,5
Task 4 (Standalone Interpreter)	
Extend `ImpParser.v`	1
Add at least three new `.lpro` programs to `examples/`	0,75

If any of the above items is only partially developed, the grade will be given accordingly. If you are unable to finish a proof, you can hand in partially developed proofs by using `admit` or `Admitted`.

You are encouraged to comment your submission, so that we can understand your decisions. You might get additional points for that (e.g., if you describe in a comment exactly what needs to be done, even though a proof is incomplete).

Other Forms of Evaluation

After submission, you may be asked to present individually your work or to develop the solution of a problem similar to the one used in the project. This decision is solely taken by the teaching team.

Fraud Detection and Plagiarism

The submission of the project assumes the commitment of honour that the project was solely executed by the members of the group that are referenced in the files/documents submitted for assessment. Failure to stand up to this commitment, i.e., the appropriation of work done by other groups or someone else, either voluntarily or involuntarily, will have as consequence the immediate failure of all students involved (including those who facilitated the occurrence).