POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Tesi di Laurea Magistrale in
Computer Science and Engineering
Ingegneria Informatica

Title of the thesis

Author:
Name Surname

Student ID:
xxxxxx

Advisor:
Prof. Name Surname

Academic Year:
20xx-xx

Dedicated to my family.

Abstract

Here goes the abstract.

Keywords: key, words, go, here

Abstract in lingua italiana

Qui va inserito l'abstract in italiano.

Parole chiave: qui, vanno, le, parole, chiave

Contents

1 | Introduction

2 State of the art

3 | Background

3.1. Description of RL

RL is an imperative programming language and its structure is similar to other common imperative languages such as Python or C. An RL program is composed of functions, and the entry point is a special function with signature `fun main() -> Int`. Each function has a list of statements to be executed sequentially. These statements include what one might expect to find in an imperative language, such as variable declarations, assignments, if statements, while loops, and function calls. RL extends this familiar language structure with a couple key features.

3.1.1. Actions

(TODO: decide on a wording / voice to use for executing statements of a function body.) In addition to regular functions, RL features another function-like construct called actions. The main difference between a function and an action is in their control flow semantics. When a function is called, it runs until it executes a return statement. Then, all variables local to the function body are discarded and solely the return value is returned to the caller. In contrast, when an action is invoked it runs until it executes a special kind of statement which causes it to suspend. Then, it returns an action object that describes the current state of execution, including the values of local variables as well as where the execution has suspended. The caller can invoke methods of this returned object to resume the execution from its previous state. Action bodies do not contain explicit return statements, actions are meant to model processes, not values.

Two kinds of statements suspend an action's executions: Action statements and actions statements (TODO: the naming is very awkward here.) An action statement has the syntax `act action_name( param_type param_name, ...)`. When the action suspends on an action statement, it can be resumed by calling the method of the action object that shares a signature with the action statement. The action then resumes execution starting from the statement immediately succeeding the action statement. In addition, the action statement's parameters are accessible by successive statements. The parameters assume the value of arguments passed to the function that resumed the execution.

An actions statement has the following syntax

```
actions:
    action_statement
    non_action_statement
    non_action_statement
    ...

    action_statement
    non_action_statement
    non_action_statement
    ...
```

An action suspended on an actions statement can be resumed on any of the action statements contained in it. When resumed, only statements until the next action statement are executed before proceeding to the actions statement's successor.

(TODO: Should I mention subaction statements here?)

Not all variables in an action's body are exposed to callers. Variables declared with the keyword `frm` are stored in the action's frame. They persist across suspensions and are reachable from outside the action. On the other hand, variables declared with the keyword `let` are temporary values. They are discarded when the action's execution suspends, just like local variables in regular functions.

As an example, consider the following program:

```
act nim(Int num_sticks) -> Nim:
    frm winner : Int
    frm current_player = 0
    frm remaining_sticks = num_sticks

    while remaining_sticks > 0:
        act pick_up_sticks(Int count)
        remaining_sticks = remaining_sticks - count
        current_player = 1 - current_player

    winner = current_player

fun main() -> Int:
    let game = nim(14)
    game.pick_up_sticks(3)
    game.pick_up_sticks(4)
```

```
17     game.pick_up_sticks(4)
18     game.pick_up_sticks(3)
19     if(!game.is_done()):
20         return 1
21     # This would result in a crash:
22     # game.pick_up_sticks(1)
23     if(game.winner != 0):
24         return 1
25     return 0
```

The main function invokes `nim(14)`. The action initializes the variables `current_player` and `remaining_sticks`, enters the loop and suspends at line 7. When it suspends, the action returns a `Nim` object, which holds the variables `current_player` and `remaining_sticks`, exposes the functions `pick_up_sticks(Int count)` and `is_done` and "remembers" that the action is suspended at the `pick_up_sticks` action statement. The main function then stores this object in the variable `game`, and calls the method it exposes multiple times. Each of these calls but the last one execute one iteration of the while loop before suspending on line 7 once again. During the last `pick_up_sticks` call at line 18, the action's execution exits the while loop and terminates, since there are no other action statements in the action's body. After this point, `game.is_done()` returns `True` and calling `pick_up_sticks` again will result in a crash since the action is not suspended on that statement. The action's frame variables are still accessible through the `Nim` object.

3.1.2. Preconditions

In RL, every function, action and subaction statement can have a list of preconditions attached to it. Any expression that evaluates to a boolean value can be a precondition, and preconditions can use the parameters of the function, action or action statement they are associated with. Unless optimizations are turned on while compiling, a function call or action instantiation with arguments that violate its target's preconditions results in a crash.

Being able to express preconditions is crucial for describing simulations. For instance, consider the nim example. The action

```
act nim(Int num_sticks) -> Nim:
    frm winner : Int
    frm current_player = 0
    frm remaining_sticks = num_sticks

    while remaining_sticks > 0:
        act pick_up_sticks(Int count)
        remaining_sticks = remaining_sticks - count
        current_player = 1 - current_player

    winner = current_player
```

allows players to pick up more sticks than there are in the game, or even a negative number of sticks. Moreover, the game is not restricted to start with a positive number of sticks in the first place. Enhancing the description with preconditions increases the description's readability, eases debugging and boosts the potential of automated analysis methods.

```
act nim(Int num_sticks) {num_sticks > 0} -> Nim:
    frm winner : Int
    frm current_player = 0
    frm remaining_sticks = num_sticks

    while remaining_sticks > 0:
        act pick_up_sticks(Int count) {count > 0 , count <=
   remaining_sticks}
        remaining_sticks = remaining_sticks - count
        current_player = 1 - current_player

    winner = current_player
```

4 | Solution design

5 | Evaluation

5.1. Goals

5.2. Conditions

5.3. Baseline

In this section, we describe the baseline we use to evaluate the efficiency of fuzzers generated by RLC. As a baseline, we need a simpler method of automatically generating fuzzers for game descriptions. Any such method needs to establish an abstraction for how a game is described. We have chosen to use the abstraction of OpenSpiel, Google DeepMind's framework for applying reinforcement learning methods to games (TODO: I probably need to describe OpenSpiel in greater detail.).

The OpenSpiel repository includes some example games. We describe a simple method of generating fuzz targets for these games. These games, modified minimally to introduce bugs the fuzzer can find, form the baseline of our evaluation.

Generating fuzzers for OpenSpiel games

(TODO: I adapted this from the OpenSpiel paper, should figure out how to cite that.) Games in OpenSpiel are described as producedural extensive-form games. Where a game has:

finite set of players. Including a special player representing chance.

A finite set of all possible actions players can take in all game states.

A finite set of histories. Each history is a sequence of actions that were taken from the start of the game.

A finite set of terminal histories that represent a finished game.

A utility for each player for each terminal history.

A player assigned to take the next action for each non-terminal history. Including a special player representing simultaneous states, where players act simultaneously choosing a joint action.

A set of states, where each state is a set of histories such that histories in the same state can not be distinguished by the acting player.

Complete game descriptions are objects that implement some interface methods exposing the elements described above, along with some utility methods to simplify moving from a history to its successors. In particular, OpenSpiel game descriptions implement the following methods that are useful for generating a fuzzer:

- `Game::NewInitialState`
- `State::isTerminal`
- `State::isChanceNode`
- `State::isSimultaneousNode`
- `State::LegalChanceOutcomes`
- `State::LegalActions`
- `State::ApplyAction`

Depending only on these functions, we can generate a fuzz target analogous to the ones we generate for RL descriptions as shown in Algorithm 5.1, where *pickOne* is a function that picks an element of the given set consuming the next $log_2(n)$ bits of the fuzz input. After generating the fuzz target, we plug the fuzz target into LLVM's libFuzzer to generate the complete fuzzer.

Algorithm 5.1 Fuzzing and OpenSpiel game

```
1: $state \leftarrow game.NewInitialState()$
2: while $state.isTerminal()$ do
3:    if $state.isChanceNode()$ then
4:       $nextAction \leftarrow$ pickOne$(state.LegalChanceOutcomes())$
5:    else if $state.isSimultaneousNode()$ then
6:       $actions \leftarrow []$
7:       for $player \in players$ do
8:          $actions.append($pickOne$(state.LegalActions(player)))$
9:       end for
10:      $nextAction \leftarrow ApplyAction(actions)$
11:   else
12:      $nextAction \leftarrow$ pickOne$(state.LegalActions())$
13:   end if
14:   $state.ApplyAction(nextAction)$
15: end while
```

It should be noted that OpenSpiel games may have two kinds of chance nodes. Explicit stochastic chance nodes expose multiple legal actions, as well as a probability distribution over those actions. On the other hand, sampled stochastic chance nodes expose a single action with non-deterministic behaviour. This fuzz target is only suitable for games with no sampled stochastic chance nodes since the fuzz target has to be deterministic with respect to the fuzz input.

Tic tac toe

5.4. Results

6 Conclusion

Bibliography

List of Figures

List of Tables

Acknowledgements

Here you may want to acknowledge someone.