Executive Summary - Non Determinism in RLisp

Raphael Kreft, Olivier Mattmann

September 13, 2021

This Executive-Summary describes the workflow and accomplishments of the Project "Non-Determinism in RLisp", which was part of the Seminar "Structure and Interpretation of Computer Programs" in the Spring Semester 2021 at the University of Basel.

Starting with a brief overview of basic concepts of Non-Determinism, we present a common way to implement this paradigm into a programming language. It follows a comparison between the reference implementation taken from the Book and our implementation into the self-developed RLisp interpreter. We will depict the main differences, structure as well as pros and cons of both sides.

To finish up, we will denote the main issues we faced during the project, followed by a summary of the project's accomplishments and a brief look into possible improvements.

1 Non Determinism and the amb special-form

First, we need to clarify the term "Non-Determinism", which is known from automaton theory. There it describes automatons, that follow different paths at the same time during computation. The same term in the field of Programming Languages is not far off. There "Non-Determinism" is a programming-paradigm, where expressions can have multiple values. It gives the illusion that the computer tries multiple paths at once and that time branches.

One way to implement Non-determinism in a programming language proposed by John McCarthy in 1961 is the **amb special-form**. This special-form represents a non-deterministic choice point in a program. Every time the nondeterministic evaluation finds an amb, it chooses one of the given alternatives(arguments of amb). The evaluator keeps track of all choices and choice points. Thus it can backtrack and try other choices in case of an error or insufficient choice.

All this enables the programmer to solve problems differently as you would normally do in a purely imperative language since non-determinism frees the programmer from the details of how choices are made.

2 Comparison of two amb implementations

The first version in this comparison is the amb special-form implemented in the "analyze-evaluator", which is completely described in chapters 3 and 4 of the book.

The evaluator uses a concept that is known as "continuations". A continuation is a function that represents a way of how the evaluation proceeds. A fail continuation, for example, is called when an error arises or a choice does not meet requirements. A success continuation gets called when an evaluation was successful. The "analyze" stage of the evaluator creates a complex construct of continuations, that gets evaluated by the "eval" process.

Our implementation is different for two main reasons: First, our Interpreter is not implemented in Lisp, like the evaluator from the book, but Rust. That means we needed to create new data structures and mechanisms that allow the evaluator to support non-determinism. Second, we needed to convey a different approach to backtracking due to our different evaluation principle. Instead of using continuations, RLisp keeps track and organizes choices in the Host-language. When a new non-deterministic path gets evaluated, the whole expression gets reevaluated. During that process, RLisp automatically manages values of non-deterministic sub-expressions.

In comparison, our implementation in RLisp is easier to understand and thus more expandable and modular. There is no need to modify or completely change the evaluation of existing deterministic expressions, like the solution in the book required. On the other hand, the solution from the book seems to be a bit more efficient, since the exploration of paths is more straightforward thanks to the continuations.

3 Lessons Learned, main issues and preview

First, we want to depict one of the main issues we faced during the work on the project, which was the lack of information. No matter how long we tried, we just found the amb implementation from the book, which uses an interpreter that uses the analyze-eval technique as a framework. Our self-implemented interpreter RLisp works entirely different so that we had to start from scratch. Thus it took us a decent amount of time and many attempts to find an appropriate solution. At this point, we kindly want to thank our lecturer Marcel Lüthi, who helped us with this problem with interesting new ideas.

In addition to that, we remembered our Software-Engineering lesson in SCRUM and held at least two meetings a week. These frequent meetings helped our understanding of what each other was doing and further empowered our team communication. In each short cycle, we focused on one specific part of our project and thus were able to solve problems quite fast.

One possible addition to our project is to optimize the evaluation of non-deterministic problems. Currently, a program gets evaluated with each possible choice which is inefficient. A first improvement would be the implementation of a more sophisticated backtracking model which makes choices more intelligent. Another way to improve performance is caching the results of every function that has no side effects.