Report for project from Programming Languages

Krzysztof Bednarek

May 31, 2020

1 Introduction

In this project, the task was to read the provided paper and try to implement the described solution. This report has been created based on [1]. In this article, authors describe their approach to lazy evaluation of simple programming language. In the second part, they explain, how one can simulate it using letrec-calculus and delimited control operations.

2 The call-by-need lambda calculus

The authors decided to create an abstract machine. It calculates the value of a program by systematically unpacking the reduction relation of calculus. It models call-by-need laziness and, rather than storing calculated values, it manipulates its evaluation context.

3 Language

Created language has only five types of expressions:

- Application – consists of two expressions separated by at least one whitespace. For example: "*e1 e2*".
- Constant value – created by writing an underscore, at least one whitespace and identifier of value. For example: "_ *b*".
- Constant function – is made up of slash, at least one whitespace, identifier of function and expression it should be applied to. For example: "/ *f e*".
- Lambda – consists of backslash, at least one whitespace, an identifier of variable, dot, and body of the lambda. For example: "|*x. f x*".
- Variable – created by writing identifier of variable. For example "*x*".

To write a comment, one need to surround intended content of comment with symbols "(*" and "*)".

3.1 Constant functions

There have been defined only one constant function – succ. It adds character "s" to the front of the name of the constant variable it has been applied to. It has been added mainly for purposes of testing programs on some actual values.

4 Implementation details

4.1 Abstract Machine

The file *AbstractMachine.hs* contains the actual implementation of calculus and machine described in [1]. First of all, there is the definition of several types. Most of them are representations of types specified in calculus:

- NVar – consists of variable name and number. Allows to easily rename to fresh names, and, at the same time, to not lose the original name.
- Term – representation of term of considered calculus. Uses the type NVar instead of Var from AST.
- Val – values in considered calculus.
- Ans – represents partial answer. It is created from value and a list of applications wrapping one around another.
- Frame – serve as a single level of context.
- Context – consists of many frames.
- Redex – represents redex from calculus.
- Config – configuration of the abstract machine. It contains a list of used variables names, context and one of the following types, depending on the type of configuration: term, value, variable or redex.

The most important function in this file is *step*. It takes the actual configuration of the machine and returns a new one, based on rules from [1]. This result is wrapped in State Monad. That allows us to hold the number of the last generated variable, and base on it, we can generate a new unique name for the next variable. It has one helper function – *rename*. It takes variable which should be replaced, variable to replace with, and term in which it should be done.
Function *runMachine* calls *step* repeatedly, until it acquires final answer. Another important function is *delta*. It specifies the behavior of constant functions.
This file also contains function *evaluateTerm*, which translates provided term to expression and evaluates it. In this file, there are also functions converting between some of the types: expression to term, term to expression, value, and answer to term.

4.2 Check Well Formed

File *CheckWellFormed.hs* contains function *checkWellFormed*, which checks, if all used variables has been previously defined, and if any variable has not been overridden. Result is return using *Maybe* monad. In case of error, function returns constructor *Just* with value of custom type – *FormnesError*.

4.3 Parser

In this file is the definition of language. It uses Haskell's module *Parsec*.

4.4 AST

File *AST.hs* contains definition of type *Expr* to which input files are parsed. It also contains definitions of *Var*, *CFn* – constant function and *CVl* – constant variable.

4.5 Main

Main file. It contains functions created to run different parts of the whole project.

5 Using implementation

5.1 Compilation

Implementation can be compiled using commands:

```
$ stack build
```

To use implementation, use command:

```
$ stack exec lazyEvaluation filename
```

where filename is name of a file containing program in this language.

5.2 Usage

Implementation has two modes:

- Parsing – parse provided program and test if it is well-formed. Activated with flag *--well-formed*.
- Evaluation – parse provided program, test if it is well-formed and evaluate its value. It is the default mode.

Additionally, called with flag *--help*, the program will show a list of options. Example programs can be found in directory *examples/*.

5.3 Testing

Tests provided in directory *tests/* can be run using commands:

```
$ stack test
```

6 Results

The final result of the project is correctly working parser and lazy evaluator of simple language. We can see it working in the following example:

```
(\ n1. \ n2. \ n3. / succ n2)
_ z
(/ succ _ z)
(/ succ / succ _ z)
```

The first function reads three numbers and returns the successor of the second of them:

```
[
    (λ(n1_1). □ ) "z",
    (λ(n2_2). □ ) "sz",
    (λ(n3_3). □ ) succ succ "z"]
"ssz"
```

To read the final result, we have to plug the last line (which is the most inner value of this answer) into the list of contexts, from bottom to the top. As we can see, the second argument of this function has been evaluated, but the third has been not. It is correct because inside this function we used only its second argument.

The main disadvantage of this machine is the fact, that it leaves applications. It does that, even after replacing all the occurrences of a variable introduced by associated lambda. It makes reading and understanding the final results of calculations very hard. We can see it in the following example. It calculates the sum of squares of two numbers:

```
(\ mult. \ add. \ a. \ b. add (mult a a)  (mult b b))
(\ a. \ b. \ f. \ x. (a (b f)) x)
(\ a. \ b. \ f. \ x. (a f) ((b f) x))

(\ f. \ x. f (f (f x)))
(\ f. \ x. f (f (f (f  x))) )

(\ x. / succ x)
_ z
```

Its result has a stack of 54 frames. Around 40 of them is already calculated and not used in any other frame:

```
[(λ(mult_1). □ ) λ(a_0). λ (b_0). λ (f_0). λ (x_0). A(A(a_0) -> A((b_0) -> f_0)) -> x_0,
(λ(add_2). □ ) λ(a_0). λ (b_0). λ (f_0). λ (x_0).
    A(A(a_0) -> f_0) -> A((A(b_0) -> f_0) -> x_0),
(λ(a_3). □ ) λ(f_0). λ (x_0). A(f_0) -> A((f_0) -> A((f_0) -> x_0)),
(λ(b_4). □ ) λ(f_0). λ (x_0). A(f_0) -> A((f_0) -> A((f_0) -> A((f_0) -> x_0))),
(λ(a_9). □ ) λ(f_0). λ (x_0). A(f_0) -> A((f_0) -> A((f_0) -> x_0)),
(λ(b_10). □ ) λ(f_0). λ (x_0). A(f_0) -> A((f_0) -> A((f_0) -> x_0)),
(λ(a_5). □ ) λ(f_0). λ (x_0). A(A(a_9) -> A((b_10) -> f_0)) -> x_0,
(λ(a_28). □ ) λ(f_0). λ (x_0). A(f_0) -> A((f_0) -> A((f_0) -> A((f_0) -> x_0))),
(λ(b_29). □ ) λ(f_0). λ (x_0). A(f_0) -> A((f_0) -> A((f_0) -> A((f_0) -> x_0))),
(λ(b_6). □ ) λ(f_0). λ (x_0). A(A(a_28) -> A((b_29) -> f_0)) -> x_0,
(λ(f_7). □ ) λ(x_0). succ x_0,
(λ(x_8). □ ) "z",
(λ(f_11). □ ) λ(x_0). succ x_0,
(λ(f_30). □ ) λ(x_0). succ x_0,
(λ(x_31). □ ) "z",
(λ(f_34). □ ) λ(x_0). succ x_0,
(λ(f_32). □ ) λ(x_0). A(f_34) -> A((f_34) -> A((f_34) -> A((f_34) -> x_0))),
(λ(x_33). □ ) "z",
(λ(x_50). □ ) "z",
(λ(x_54). □ ) "z",
(λ(x_53). □ ) "sz",
(λ(x_52). □ ) "ssz",
(λ(x_51). □ ) "sssz",
(λ(x_45). □ ) "ssssz",
(λ(x_49). □ ) "ssssz",
(λ(x_48). □ ) "ssssssz",
(λ(x_47). □ ) "sssssssz",
(λ(x_46). □ ) "ssssssssz",
(λ(x_40). □ ) "sssssssssz",
(λ(x_44). □ ) "sssssssssz",
(λ(x_43). □ ) "ssssssssssz",
(λ(x_42). □ ) "sssssssssssz",
(λ(x_41). □ ) "ssssssssssssz",
(λ(x_35). □ ) "sssssssssssssz",
(λ(x_39). □ ) "sssssssssssssz",
(λ(x_38). □ ) "ssssssssssssssz",
(λ(x_37). □ ) "sssssssssssssssz",
(λ(x_36). □ ) "ssssssssssssssssz",
(λ(x_12). □ ) "sssssssssssssssssz",
(λ(f_15). □ ) λ(x_0). succ x_0,
(λ(f_13). □ ) λ(x_0). A(f_15) -> A((f_15) -> A((f_15) -> x_0)),
(λ(x_14). □ ) "ssssssssssssssssz",
(λ(x_24). □ ) "ssssssssssssssssz",
(λ(x_27). □ ) "ssssssssssssssssz",
(λ(x_26). □ ) "sssssssssssssssssz",
(λ(x_25). □ ) "ssssssssssssssssssz",
(λ(x_20). □ ) "sssssssssssssssssssz",
(λ(x_23). □ ) "sssssssssssssssssssz",
(λ(x_22). □ ) "ssssssssssssssssssssz",
(λ(x_21). □ ) "sssssssssssssssssssssz",
(λ(x_16). □ ) "ssssssssssssssssssssssz",
(λ(x_19). □ ) "ssssssssssssssssssssssz",
(λ(x_18). □ ) "sssssssssssssssssssssssz",
(λ(x_17). □ ) "ssssssssssssssssssssssssz"]
"sssssssssssssssssssssssssz"
```

Program “factorial“ in directory “examples/“ is an even more appropriate example. It calculates the factorial of 4 and its answer has over 400 frames.

References

[1] Lazy Evaluation and Delimited Control – Ronald Garcia (Carnegie Mellon University), Andrew Lumsdaine (Indiana University), Amr Sabry (Indiana University)