DD2481 - Principles of Programming Languages
Lab 2

Andreas Lindner

11 April 2018

1 Goal

The goal of this second lab is the extension of lab 1 with the untyped lambda calculus using the call-by-value evaluation strategy seen in the lecture. We provide an extended code template to enable parsing of the extended syntax. Additionally, this code template contains a simple test infrastructure for you to extend.

2 Syntax and Semantics

The syntax and small-step evaluation rules extending the ones from lab 1 follow.

Syntax extensions:

$$
\begin{array}{rcll}
t & ::= & & \textit{terms}: \\
 & | & \ldots & (\textit{see lab 1}) \\
 & | & \lambda x.\ t & \textit{abstraction} \\
 & | & t\ t & \textit{application}
\end{array}
$$

$$
\begin{array}{rcll}
v & ::= & & \textit{values}: \\
 & | & \ldots & (\textit{see lab 1}) \\
 & | & \lambda x.\ t & \textit{abstraction value}
\end{array}
$$

Evaluation extensions:

$$\text{E-App1}\quad \frac{t_1 \mid \mu \to t_1' \mid \mu'}{t_1\ t_2 \mid \mu \to t_1'\ t_2 \mid \mu'} \qquad \text{E-App2}\quad \frac{t_2 \mid \mu \to t_2' \mid \mu'}{v_1\ t_2 \mid \mu \to v_1\ t_2' \mid \mu'}$$

$$\text{E-AppAbs}\quad (\lambda x.\ t_{12})\ v_2 \mid \mu \to [x \mapsto v_2]t_{12} \mid \mu$$

3 Task

As in lab 1, the interpreter should be based directly on the small-step evaluation rules. This means that a step is defined exactly by these rules and nothing else. Make sure that you can argue where and how your code ties to these rules. This applies also to the rules you implemented for lab 1.

In order to convince yourself that you implemented the rules correctly you will develop a small test suite containing test programs and expected outputs. The test framework is described in Section 6. By implementing at least a function for integer multiplication and a function for calculating the factorial of an integer, you show that you worked with the language we are building.

4 Parsing and AST extension

Notice the syntax difference for input programs and the abstract syntax and semantics description above. A "λ" above corresponds to a "\" character in our interpreter input files.

Notice also that you might require more parenthesis that you would expect. In a nutshell, we assign binding strengths to the language constructs. Embedding constructs of less binding strength in constructs with higher binding strength always requires parentheses. Thus, it might help to just add some parentheses if you think your input should be able to parse but parsing fails. The binding strengths in ascending order are as follows:

1. Sequence-operator
2. Ifthenelse, iszero, assignment, lambda abstraction
3. Plus
4. Function application

The extended parts of the AST, which are also provided to you, are implemented in Scala as follows (file "parser/AST.scala"):

```
trait AST extends Positional
...
case class LamExp(id: String, e: AST) extends AST
case class AppExp(e1: AST, e2: AST) extends AST
```

5 Implementation

Your implementation will be in the `step` method of the `Interpreter` singleton object like in lab 1. It is located in the same file. Please refer to lab 1 for the template description.

Compare the two templates using a textual or graphical diff tool of your choice to review all the changes we made. Some, like the changes in the parser, are less relevant. Specifically, take a closer look to the changes in the `eval` method of the `Interpreter` singleton object. Corresponding comments explain the relation of the code to the interpretation of the semantic rules listed above.

A new boolean flag `printSteps` you can set in the code allows you to enable printing of intermediate evaluation steps. Thereby, you can check whether the evaluation steps computed in scala correspond to the small-step semantics provided above.

To get started, merge the new skeleton with your existing code. For implementing the evaluation rules for lab 2, it might be helpful to start by implementing a function for determining the set of free variables in an expressions. Another function you should implement is actually carrying out the substitution in an expression. Notice that a substitution might cause a variable to get captured. This is an error case in which you should use the function `errVarCap` we defined for you together with two test cases. The set of free variables is useful for identifying the case of a captured variable during substitution and it can be simply computed as an immutable list. For this purpose the template already contains the two following method stubs:

```
/* function for determining the
   free variables of an expression */
def freevars (x: AST) : List[String] =
    throw new Exception("implement me")

/* function for carrying out a substitution */
def subst (x: String, s: AST, t: AST):AST =
    throw new Exception("implement me")
```

The course book "Types and Programming Languages" might be a useful reference for this task, among others the Definition 5.3.5 [Substitution]. The third rule of the substitution definition in this book clarifies the usage of the freevars function.

6 Development and testing

The scala build tool `sbt` offers options for automatically rerunning the program once code changes. This also works for other files once they are specified in the "build.sbt" file using the `watchSources` option.

Using the template directly for example, you may develop interpreter input code in a file "src/input.sint" and then use the command `~run src/test.sint` (simply prepending a `~`). Whenever you change scala source code or this input file, your program will be compiled an rerun.

Once you finished developing a test input code, you want to add it to the test framework. For this purpose, take a look at the classes found in the "src/test/scala/simpret" directory. The corresponding input files reside in the directories "src/test/sint/lab1" and "src/test/sint/lab2". Test cases you develop for the rules introduced in lab 1, should be added to MainTestLab1. Similarly, test cases involving function abstraction should be added to MainTestLab2. You may want to get inspired by the examples already found there.

Notice that we expect you to generate a decent coverage, which is required when convincing somebody of the correctness by testing. Remember to also add test cases to test when the evaluation is supposed to get stuck. All the available test cases defined in this directory can be run by simply issuing the command `test` in the `sbt` shell.

7 Example

Since our language is relatively restricted when it comes to integers, subtraction does not lend itself as a pedagogical example. Therefore, we focus on implementing a simple sum expression which calculates the square of an integer greater than zero.

$$n^2 = \sum_{n}^{k=1} 2k - 1$$

We can implement this as the following imperative loop-based pseudocode using only constructs available or simple to represent in our language:

```
n := n - 1;
k := 1;
sum := 0;

while !(iszero n)
  sum = sum + k + k - 1
  k := k + 1;
  n := n - 1;
```

The variable `sum` will contain the result after execution and this implementation is feasible to represent in our language because of the following. Firstly, decrements can be directly written as addition with the integer literal -1. Secondly, the loop can be represented by a recursive function, which can be represented as a lambda expression using the fix-point operator as we have seen in the lecture. Finally, the boolean negation can be represented using a simple conditional expression.

An alternative to translating it using the steps above is to simply rewrite it in a functional recursive manner:

```
fun square n =
  if iszero (n - 1) then
    1
  else
    k + k - 1 + square (n - 1)
```

Now we translate this to our simple language and we start by defining the inputs. In this case it is only the variable `n` and we set it to 3 to compute the corresponding square value later on.

```
n := 3;
```

Since we need recursion, we define the fix-point operator in our language and assign it to a variable for reusability.

```
fix := \f. (\x. f (\y. x x y)) (\x. f (\y. x x y));
```

The helper function representing our translated square computation can be represented in our language as follows:

```
g := \sq. \n. if iszero (n + -1) then
                  1
                else
                  n + n + -1 + sq (n + -1);
```

Applying `fix` to `g` leaves us with the square function we are interested in.

```
square := fix g;
```

And finally a line to evaluate the square function with the previously defined input allows us to evaluate the whole program in our interpreter - assuming that the evaluation rules have been implemented correctly.

```
square n
```

This example program with three different input values can be found in the test case input directory for lab 2. Notice, this code only works for $n > 0$.

8 Multiplication and factorial

In the last section you saw how to implement a recursive function in our toy language. Now you can go ahead and implement functions for computing a multiplication and afterwards factorial.

Since we are only provided with addition and a zero check on integers, we can make some simplifications for our experiments. For the factorial function it is enough to consider positive integers for example. Similarly, the multiplication function only needs to work on positive integers. For convenience, your multiplication function may be called times.

If you feel challenged, you could additionally try to implement a function for comparing integers and a subtraction function for example. This might require thought about the integer representation in `scala` and word length.

Good luck!

9 Changelog

Version	Contribution	Author	Contact
0.1	Derived from lab 1	Philipp Haller	phaller@kth.se
0.2	Development for VT2018	Andreas Lindner	andili@kth.se
1.0	Development for VT2018	Andreas Lindner	andili@kth.se