Second Task

Programming Languages

LVA 185.208
2023 S
TU Wien

What to do

Develop an interpreter for a functional language as specified below in a dynamically typed language like Python or Ruby.

The Language

A program is an expression that gets evaluated when executing the program. The following language elements shall be supported:

- integers
- structured data (like lists or records containing integers and other structured data)
- functions (e.g., lambda abstractions as in the lambda calculus)
- named entities (as a means to specify names for above language elements and use names to adress them)
- a small set of predefined operations, e.g., functions named `plus`, `minus`, `mult`, `div` and `cond` (conditional execution)

It is recommended to use the lambda calculus as well as Lisp or another simple, interpreted language as a guideline, but to implement only the smallest possible subset of it.

Please feel free to design your own language, but keep the syntax and semantics simple. It is recommended not to use expensive compiler technologies for developing a lexer, parser, AST, etc., although it is not forbidden to do so.

An Example Language

Here is the description of a language as an example. If you want, you can implement this language, but your own language is preferred. This is the syntax in EBNF:

```
 <expr> ::= <apply>
          | <name> '->' <expr>

<apply> ::= <basic>
          | <apply> <basic>

<basic> ::= <integer>
          | <name>
          | '(' <expr> ')'
          | '{' [<pairs>] '}'
```

```
<pairs> ::= <name> = <expr>
          | <pairs> ',' <name> = <expr>
```

Basic tokens are integer literals and names, together with some symbols and parentheses as syntactic elements. Looking at the next (one or two) tokens shall be sufficient to determine the next element syntactically. Most of the syntax shall be self-explanatory. An expression `x->`... represents a function (lambda abstraction) with `x` as formal parameter and ... as function body, e.g., `x->mult x x` is a function returning the square of `x`. Expressions can be nested as in `x->y->add(mult x x)y` or equivalently `x->(y->add(mult x x)y)`. In this way nested functions, each with one parameter, can be used to specify functions with several parameters (Currying). Round parentheses group syntactic elements. A function application is written as a function followed by its argument. For example, `(x->mult x x) 2` applies the above function to 2, thereby replacing the formal parameter by 2, resulting in `mult 2 2` which should be evaluated to 4. By nesting function applications we can provide several parameters to functions, e.g., `(x->y->add(mult x x)y) 2 3` is equivalent to `((x->(y->add(mult x x)y))2)3`, and the evaluation will, step by step, result in

```
(x -> y -> add (mult x x) y) 2 3  ⟶
(y -> add (mult 2 2) y) 3  ⟶
add (mult 2 2) 3  ⟶
add 4 3  ⟶
7
```

Parameters can be of any kind, i.e., integers, functions, and records.

Expressions like `{d=x->mult x x, v=d 2}` represent records of named fields, `{}` is the empty record. In this example, `d` is a named function and `v` stands for the result of applying `d` to `2`. Pairs within braces are evaluated from left to right. At the time when `v` is evaluated, the value of `d` is already known and can be used. In general, lazy evaluation is used. Hence, when constructing a record, only names are resolved while constructing an internal representation, that is, names are replaced with references to corresponding values or executable expressions, but referenced expressions are not executed immediately. So the value of `v` is just `d 2`, not 4. A record not applied to some other expression stands just for itself. Therefore, executing the program `{a=x->y->add(mult x x)y, b=a 2, c=b 3}` has just `{a=x->y->add(mult x x)y, b=a 2, c=b 3}` as result. In a similar way, a function not applied to an argument stands just for itself. For example, executing a program `x->y->add(mult x x)y` results in `x->y->add(mult x x)y`. However, each function applied to an argument outside of records is reduced as far as possible, the outermost function first. For example, if `(x->y->add(mult~x~x)y) 3` ist the whole program, it will be evaluated to `y->add 9 y`.

Records specify environments of named expressions when applied to arguments (i.e., a record followed by an expression). For example, `{a=x->y->add(mult x x)y, b=a 2, c=b 3}minus(b 5)c` is evaluated to 2: `minus(b 5)c` stands for

```
minus ((x -> y -> add (mult x x) y) 2 5)
      ((x -> y -> add (mult x x) y) 2 3)
```

which is reduced to `minus 9 7` and finally 2. Hence, `{...}e` has approximately the same meaning as `let ... in e` in usual functional programming languages. However, `{...}` can be used as arguments and results of functions or as named entities in other records, something that cannot be done with `let ...` without `in e`. Records are building blocks of larger data structures. An expression `{...}e` can also be seen as the execution of `e` in a program `{...}`. Predefined operations can be regarded as existing in each environment.

We assume that the predefined operation `cond c a b` returns the evaluated argument `b` if the evaluated argument `c` is 0 or the empty record `{}`, otherwise it returns the evaluated argument `a`; the unused argument is not evaluated. The predefined operations `plus x y` and `minus x y` have usual semantics. Here is a larger example:

```
{
    list = c -> f -> x ->
        cond (c x)
             { val = x, nxt = list c f (f x) }
             {}
    ,
    reduce = f -> x -> lst ->
        cond lst
             (f (reduce f x (lst nxt)) (lst val))
             x
    ,
    range = a -> b ->
        list (x -> minus b x) (x -> plus 1 x) a
    ,
    sum = lst ->
        reduce (x -> y -> plus x y) 0 lst
}
sum (range 3 6)
```

The outermost function in the program gets executed first, this is the application of `sum` to `range 3 6` in the specified environment. Because of lazy evaluation always the outermost executable function gets executed, constructing the result step by step. The functions `range` and `list` will together produce a data structure corresponding to `{val=3,nxt={val=4,nxt={val=5,nxt={}}}}`, but this data structure will never become visible in this form because the functions `sum` and `reduce` use up all parts of the data structure while it is still

under construction. The result of the computation shall be `12`, this is the sum of 0, 3, 4 and 5. As a detail: `lst nxt` in `reduce` is the field `nxt` in the record provided as parameter `lst` because the application of `lst` to `nxt` sets the environment and thereby hides every other meaning of `nxt` that existed before; accordingly for `lst val`.