Programming Languages
A Journey into Abstraction and Composition
2nd Exercise: Functions, Substitution, First-Order and Higher-Order Functions

Programming Group
guido.salvaneschi@uzh.ch

University of Zurich – June 28th, 2022

Taxonomy of Functions

Question 1:

What is the difference between first-order functions and higher-order functions? Give an example for a first-order function and for a higher-order function.

A first-order function is a function that takes a non-function as argument (e.g. an Int) and returns a non-function. Examples in Scala: `+`, `String.length`.

A higher-order function is a function that takes a (first-class) function as argument or returns a (first-class) function. Examples in Scala: `map`, `filter`, `fold` on `List`.

Question 2:

What does it mean that a language has first-class functions? Does a language with first-class functions support higher-order functions?

A language that supports first-class functions treats functions as values (first-class members). A first-class function is a function expression that is itself an expression in the language and that can, for example, be passed to another function as argument. That means that a language that supports first-class functions also supports higher-order functions.

Example for a first-class function in Scala: `val f = (x : Int)=>x + 5`

Question 3:

Are there first-class/higher-order functions in F1LAE? Justify your answer to this question by pointing out relevant parts in the F1LAE interpreter code.

F1LAE contains neither first-class functions, nor higher-order functions: In the F1LAE language definition, `FunDef` does not extend F1LAE and hence a function definition is not by itself a valid expression in the language. Thus, it is also not possible to define a function in F1LAE that takes another function as argument or returns a function.

Immediate Substitution

The first interpreter for F1LAE that we saw used immediate substitution for evaluating let-expressions. Let us look at some example expressions to understand what that means.

Task 4:

Define the following mathematical formula "as is" in F1LAE, i.e., not using let-expressions:

$$f(x) + g(y), \text{ where } f(x) = x + c \text{ and } g(x) = x + 1 + c$$

File

```
1  val fundefs = Map(
2      "f" -> FunDef("x", Add(Id("x"), Id("c"))),
3      "g" -> FunDef("x", Add(Id("x"), Add(Num(1), Id("c"))))
4  )
5  val expr = Add(App("f", Id("x")), App("g", Id("y")))
```

Task 5:

What happens if you directly execute this expression? What do you have to do in order to execute the expression with, e.g., $x = 5, y = 10, c = 2$? Can you make the expression executable without modifying your original expressions for $f(x)$ and $g(x)$? If not, what is the problem?

Direct execution of `expr` with `fundefs` results in an exception ("found unbound id "x"""). In order to execute `expr` with concrete values, we have to bind the identifiers using Let-expressions. For example, we may try:

File

```
1  interp(
2      Let("x", Num(5), Let("y", Num(10), Let("c", Num(2), expr))),
3      fundefs
4  )
```

However, in our F1LAE interpreter with immediate substitution, this results in an exception "found unbound id "c"". In order to make expr executable with these values, we would have to bind c directly within the definitions of the functions (twice, once in every function body!).

Immediate Substitution vs. Deferred Substitution

Question 6: Environments

For the language F1LAE, we introduced environments as an alternative for substitutions. Why did we do this?

There are multiple answers to this question:

1. To reduce the complexity of the interpreter. If we have many let-expressions in our program, executing immediate substitution is really slow, because the interpreter has to traverse parts of the program multiple times in order to apply substitution.

2. If we want to evaluate programs with dynamic scoping, we have to be able to know about the substitution for a variable like c after fetching the function body from the function definition map.

Task 7: Substitution vs. Environments

Discuss the advantages and disadvantages of substitution versus environments.

Multiple answers are possible here:

1. One could say that an interpreter with substitution is more intuitive because it is clear what happens and which exact expressions will be evaluated in each step. Also, when using immediate substitution, we don't need an extra data-structure (which could potentially get very large).
2. However, as we have seen before, environments make our interpreter a lot more efficient in the presence of programs with a lot of let-expressions.

Task 8:

Come up with more F1LAE examples that notably behave differently when run with an interpreter that uses substitution versus environments.

The examples are all along the lines: If you have a function definition with a free identifier that you try to bind *outside* of the function definition via a let, this will throw an error during substitution, but evaluate when using environments and dynamic scoping (assuming that the program expression does not contain any other errors!).

First-order Functions

Task 9: Implementing `If0`

Extend the language F1LAE with an `If0` expression:

```
case class If0(test: F1LAE, thenBody: F1LAE, elseBody: F1LAE)extends F1LAE
```

The new expression type should have the following semantics: If `test` is 0, the `then` part will be evaluated. Otherwise, the `else` part will be evaluated.

Use the template provided in `first_order_functions/If0.scala`.

Task 10: Fibonacci

Use the extended F1LAE language with `If0` you implemented in the previous task to implement a function that computes the n-th Fibonacci number recursively.

Use the template provided in `first_order_functions/Fibonacci.scala`.

Task 11: Multiple Arguments

In the substitution-based interpreter for F1LAE, modify the language such that it allows functions with an arbitrary number of arguments. This means every function definition and every function application should hold a list of function arguments.

Use the template provided `first_order_functions/MultipleArguments.scala`.

Higher-Order Functions

Task 12: Arithmetics

Use the template provided in `higher_order_functions/Arithmetics.scala`:

a) Write a recursive function `sum` that sums up the elements of a list of numbers.

b) Write a recursive function `prod` that builds the product of the elements in list of numbers.

c) Can you recognize similarities between `sum` and `prod`? Write a higher-order function `fold` that abstracts over these similarities.

d) Reimplement `sum` and `prod` in terms of `fold`.

Task 13: Trees

Use the template provided in `higher_order_functions/Trees.scala`:

a) Write a recursive function `sumTree` that sums up the elements of a tree of numbers.

b) Write a recursive function `collectTree` that collects all elements stored in the leaves of a tree.

c) Can you recognize similarities between `sumTree` and `collectTree`? Write a higher-order function `foldTree` that abstracts over these similarities.

d) Reimplement `sumTree` and `collectTree` in terms of `foldTree`.

Arithmetic Expressions

Programs

Task 14: Programs

Use the template provided in `arithmetic_expressions/Programs.scala`:

a) Write a function `progSize` that counts the number of AST nodes in a program.

b) Write a function `freeVars` that collects the free (unbound) variables of a program.

c) Can you recognize similarities between `progSize` and `freeVars`? Write a higher-order function `foldProg` that abstracts over these similarities.

d) Reimplement `progSize` and `freeVars` in terms of `foldProg`.

Program Equivalence

Take a look at `arithmetic_expressions/ProgramEquivalence.scala` to answer the following questions.

Question 15: Laziness

Is this interpreter eager or lazy?

It is a lazy interpreter, because expressions (`namedExpr`) are substituted without evaluating.

Question 16:

Would you say that the programs `program1`, `program2`, and `program3` are equal? What types of equivalence can you think of?

Programs.scala

```
1  val program1 = Let("x", Num(2), Let("y", Num(3), Add(Id("x"), Id("y"))))
2  val program2 = Let("x", Num(2), Let("y2", Num(3), Add(Id("x"), Id("y2"))))
3  val program3 = Num(5)
```

Behavioral Equivalence When two programs evaluate to the same result then they are equal. In that sense, all three programs are equal. This equality definition blanks out the computation time that is needed for evaluating an expression.

Exact Equivalence Two programs are equal if they match exactly. None of the three provided programs are exactly equal to the other programs. This type of equality is often to restrictive to be useful.

Alpha Equivalence Two programs are equal when they match up to renaming of bound variables. Alpha equivalence is an important equivalence in the field of programming languages. But the importance of alpha equivalence is not just so that we can ignore the choice of bound variable names. We will see in that in the next task.

Question 17:

Would you say that the programs `program1b` and `program2b` are equal? What is the problem?

ProgramsB.scala

```
1  val program1b = Let("x", Id("y"), Let("y", Num(3), Add(Id("x"), Id("y"))))
2  val program2b = Let("x", Id("y"), Let("y2", Num(3), Add(Id("x"), Id("y2"))))
```

These two programs evaluate differently with our current interpreter implementation. `program1b` evaluates to 6, and `program2b` does not evaluate (unbound identifier).

The problem is that in the first case we are allowed to substitute an expression (`Id("y")`) that has `y` as a free variable for `x` in a place where `y` is bound.