Programming Languages
A Journey into Abstraction and Composition
3rd Exercise: First-Class Functions, Scoping and Closures

Programming Group
guido.salvaneschi@uzh.ch

University of Zurich – June 28th, 2022

First-class Functions

The following exercises are about extending the FLAE interpreter and understanding the difference between first-class functions and first-order functions. Definitions of the base languages used in these assignments are in the `first_class_functions.definitions` package.

Task 1: Let

We do not need the `let` construct when we have functions as values. Taking this into consideration, define a preprocessor for FLAE such that `let` is replaced with a combination of `App` and `Fun`.

Use the template provided in `first_class_functions/Let.scala`. Your code has to be in functional style, i.e., do not use mutable `var`s and imperative constructs.

Task 2: Multiple Arguments

Multi-argument functions are another feature that can be implemented in a preprocessor. We call FLAE with multiple-arguments MFLAE. Define a preprocessor where the support for multiple-argument functions is replaced with currying and single-argument functions.

Use the template provided in `first_class_functions/MultipleArguments.scala`. Your code has to be in functional style, i.e., do not use mutable `var`s and imperative constructs.

Task 3: First-Order vs. First-Class Scala

Implement two functions in Scala that make use of first-class functions and could not be written nicely with first-order functions only, and two test cases *for each* function.

Use the templates provided in `first_class_functions/{ScalaFns,ScalaFnsTest}.scala`.

Task 4: First-Order vs. First-Class MFAE

Implement two functions in the MFAE language that make use of first-class functions and could not be written nicely with first-order functions only, and two test cases *for each* function.

To define MFAE functions use Scala's `val` definitions, like in the example already given in the project template. Note that the example does not use first-class functions. You may use examples similar to the ones in the previous task.

Use the templates provided in `first_class_functions/{MFAEFns,MFAEFnsTest}.scala`.

Scoping

Question 5: Dynamic vs. Static Scoping

What is the difference between dynamic scoping and static (lexical) scoping?

In a language with dynamic scoping, the scope of an identifier's binding is the entire remainder of the execution during which that binding is in effect. In a language with static scoping, the scope of an identifier's binding is a syntactically delimited region. In the interpreter code, we can see the difference when interpreting `App`-nodes: To interpret a function's body, we ignore the previous environment and only bind the arguments.

Question 6: "Static"

What does "static" stand for in static scoping?

In static scoping, we can decide *statically*, i.e., by looking at the source text of an expression without executing the program, whether an identifier is bound or not.

Question 7: Dynamic Scoping

What are potential problems with dynamic scoping?

In general, dynamic scoping can result in unexpected behavior: Since free identifiers can be dynamically bound during program execution, it is not immediately clear before execution how or whether a free identifier in a function body is bound. If the programmer forgets to bind an identifier, the interpreter will dynamically fail.

Closures

Consider the interpreters for FLAE in the following questions.

Question 8: Definition

In the interpreter with environments we introduced closures. What is a closure? What do we need them for?

A closure is a function definition combined with the environment at function-definition time.

We need closures for to implement static scoping in a language with first-class/higher-order functions: The closure "remembers" exactly the bindings for free identifiers used within the function definition that was valid when the function was defined.

Question 9: Closures and Substitution

Why did we not need closures in the interpreter with substitution?

The interpreter with substitution immediately replaces all bindings in function definitions within let-bodies. Hence, any binding is "automatically remembered"—it is directly inlined into the function

definition.

Question 10:

Why did we not need closures in the F1LAE interpreter with environments?

In the F1LAE interpreter, functions are not first-class and hence cannot be defined within a program expression. Instead, functions are predefined in an external environment given to the interpreter for evaluating a program expression. Therefore, when implementing static scoping, it is simply not possible to bind free identifiers in function definitions (neither when using substitution, nor when using environments). Hence, there is no need for closures.

When implementing dynamic scoping, the environment would always dynamically contain any binding that we need during evaluation, just as in the dynamically scoped interpreter for FLAE, where you don't need closures anyway.