Lesson 5: Introduction to functional programming: IPMF 2021

THE TERM-REWRITING MODEL OF COMPUTATION

Here we consider basic concepts that describe the two main abstract models of computation we use. (A valid "objection" can be raised: is the theoretical content of this lesson appropriate for beginners? Answer: if you understand this content, it will be easier for you to study Gofer in parallel with C, Java, etc.)

One main difference between the two rests on the meaning of the equality symbol "=". In the model that we are studying, it gives rise to a fact: equality by definition. In the other, it causes the action of assignment.

(Note: This Lesson does not use the interpreter, but there is an exercise marked €: for you to use it.)

Gofer, the name of our programming language, is an acronym for "good for equational reasoning". Here we discuss the meaning of the term equational reasoning, and briefly explore how equational reasoning is the process that drives computation in functional programming.

Equational reasoning is the name for a basic idea that we are all familiar with. Simply stated, it is what we do in basic algebra. If α and β are "syntactically well-formed sequences of symbols", we understand the equation $\alpha=\beta$ to mean that α and β have the same value e.g., 2+2=4, or taking a more substantive example, `sin 2x = 2 sin x cos x`. Equation reasoning proceeds as follows: whenever we see one of α or β in a larger expression, we may replace it by the other, without changing the value of the larger expression.

In middle and high school, we manipulate expressions in this manner in order to simplify them, till they reach an *irreducible form*, i.e., cannot be simplified any more. Schoolchildren call it the "answer." Essentially, this is what the Gofer interpreter does. Specifically, it always replaces the left hand side of an equation by the right hand side. (Note: Sometimes, while writing proofs related to Gofer, we humans may replace the right hand side of an equation by its left hand side.) The exact manner in which the interpreter proceeds is given by an example below.

Story: Imagine a ninth-standard student in the process of performing calculations. Imagine this student has the capacity to perform the basic arithmetic operations, but that he simply does not have the abstraction capability to comprehend how algebraic formulas arise and how they are derived. (We are all familiar with this type of student; unfortunately, it seems that each class has a few such students.)

Suppose the teacher poses the problem of finding the interest on a certain principal P, where the interest rate is R % and the time for which the loan is taken is N years. If the numbers are relatively small and simple, it is easy to mentally calculate the interest and the total amount due. But not for our hypothetical student—he has to memorize the relevant formulas (though substitution within formulas and arithmetic computations are within his capabilities.) Suppose, for a simple example, the principal is Rs 2000, the interest rate is 15% per annum and the time period is 3 years. The teacher systematically writes the solution on the blackboard, as given in the column on the left:

Classroom exercise	Gofer script (initial letter is always lower case)
SI = P*N*R/100	`si = p*n*(r/100.0)`
Amt = P + SI	`amt = p + si`
P = Rs 2000	`P = 2000.0`
N=3 years	`n = 3.0`
R= 15% pa	`r = 15.0`

Then our student writes the equation for simple interest, and rewrites it substituting values for P, N and R, thus obtaining SI = 2000 * 3 * 15 /100. After performing the two multiplications and one division, the student obtains the answer that SI = Rs 900 and that the amount Amt = P + SI = 2000 + 900 = Rs 2900.

The column on the right is the same as that on the left, except for changes required by Gofer: the variable names start with lower-case letters (a requirement of Gofer syntax), all the numeric constants are floating-point (recall that with integer operands, Gofer performs integer division where any remainder is lost, Gofer cannot perform binary operations on an integer and a floating-point) and moreover, parentheses are introduced in the formula for simple interest (because in Gofer, using a `*` and a `/` together leads to ambiguity).

€: These equations are communicated to the Gofer interpreter by writing them in a script and then loading the script into the interpreter. (The previous Lesson, Lesson 4, has details of how to do this.)

The Gofer interpreter performs its calculations with equational reasoning just as we do in school mathematics. If we enter the variable name `amt` after the Gofer prompt, then the interpreter internally replaces the name `amt` with `p + si`, the right hand side of the equation for `amt`, and next, `si` with `p*n*(r/100.0)`, the right hand side of the equation for `si`. (Gofer is, arguably, even slower than our student: in rewriting, only one variable name is replaced at a time by its right hand side, or only one arithmetic operation is performed.) Next, each of the variables in this expression is replaced by its value, i.e., the right hand side of the corresponding equation (one at a time), then the two multiplication and one division operations are performed (one at a time), thus yielding the answer that the value of `si` is `900.0`. In this way, the Gofer interpreter performs calculations just as a maths student would. The only difference is that integers and floating point numbers cannot be mixed in Gofer operations. (Initially, this appears to be a limitation, but we shall find that it is turned into a powerful tool for checking the correctness of types, and hence the correctness of our thinking.)

At each step the term or expression being evaluated is rewritten, either by replacing one variable name with its definition on the right hand side of an equation, or by replacing an operator and its operand(s) with the value resulting after performing the operation. This process comes to a halt when no more such rewriting steps can take place. The resulting term is said to be in irreducible form, and is the “answer”, the value of the expression to be evaluated. This is why the the abstract computer that the Gofer interpreter implements is called the *term-rewriting model* of computation. In contrast, practically all computers in use today employ the *state-change model of computation*.

The question arises: in the expression `p + si`, does the interpreter replace `p` by its value first, or `si` by its expression first? It does not matter in Gofer, just as it does not in arithmetic. Either way, the answer is the same. (There is an important theorem in the mathematics underlying Gofer, that no matter which way any arbitrary expression is reduced in this way, the result is the same.) Given this fact, the exact order in which the interpreter performs the operations is not relevant for us.

If the sequence of equations above is placed in a program written in a language like C or Python, then the system finds an error: the variable `p` is undefined in the statement for `si`. If we move the statement `p = 1500.0` before the equation for `si`, then the error is changed slightly: `n` is undefined. If we move the statement n = 3.0 to just after the equation for p, then the error is: r is undefined. If all three statements, for `p`, `n` and `r`, are moved before that for `si`, then indeed the correct value of si is determined.

What is happening in C or Python is that “`si = p*n*r/100`” is *not an equation with which we can employ equational reasoning*. That is why we have called it a *statement* in the preceding paragraph, not an *equation*. Warning: The paragraph that follows has an explanation that is brief to the extent of being incomplete! Much later, we will be expanding upon the concepts introduced below.

In languages like C, the = symbol does not stand for *equality*, as it does in high school mathematics. Instead, it stands for an *action* called assignment. The meaning of a statement with the “=” symbol is: calculate the value of the expression to the right hand side of the “=”, and assign it to the name on the left hand side of the “=”. (The details depend on the language: in C, assignment means: *store* it in the memory location associated with the name on the left hand side, whereas in Python it means, effectively, draw an arrow from the name to this newly computed value.) This is not at all the way in which the Gofer interpreter behaves. In both C and Python, something in memory (either the memory location associated with the name, or the arrow from the name to its value) changes. The technical term for "content of memory" is *state*, and hence the abstract computer performing calculations in languages like C or Python is called the *state change model* of computation.

The behaviour of the term-rewriting and state-change models of computation gives rise to two corresponding programming styles: the *declarative* and *imperative programming paradigms*. (Terminology: A *paradigm* is a fundamental way of thinking or viewing the world.)

The origin of these two terms is in the syntax of sentences in natural languages: a declarative sentence is merely an announcement or narration, whereas an imperative statement is a command to perform an action.

Examples: (declarative statement) "The ground is wet."
(imperative stmt) "Pour some water on the ground," or, "make the ground wet."

The fundamental difference in these two statements is essentially the same as the different interpretations of forms like `p=2000` and `a = p + si` in the two models of computation. In the term-rewriting model, the forms are treated as equations, announcements of facts: `p` is `2000` and `a` is the sum of `p` and `si`, while in the state-change model, they are *instructions* to assign the values of expressions on the right hand side of the = symbol to the names on the left hand side. In other words, the = sign refers to the narration of a fact in the term-rewriting model, whereas in the state-change model it signifies an instruction to perform the action of assignment. For these reasons, the programming styles for the two models of computation are called *declarative* and *imperative*.

The *functional programming* paradigm is the dominant style of declarative programming. In functional programming, the declarations (announcements of fact) are in the form of definitions of mathematical functions, so that in a given instance, the value of a function can be computed for a given argument. How the function is evaluated is not our concern: there are values—argument values and function values—but no "mutable memory" in which they are stored. Given an argument value, the function value is the same every time. In particular the function value does not depend on the value of some other name. This behaviour is called *referential transparency*. This is the style of thinking and programming that we shall be studying first, using the Gofer programming platform.

We will then study imperative programming.

Date 06-01-2020 edited 01-01-2021, 06-01-2021