Programming Paradigms (Neapolis University Pafos) Assignment 1

February 28, 2025

Set-up: For this assignment, you will complete and extend implementation of an interpreter for a small "language" for two-dimensional geometry objects. An implementation in Python is mostly completed for you. You should choose any other functional programming language (Haskell, Lisp, Scala, SML, Racket, ...) and rewrite the code in FP-style. Try to heavily use functions and pattern-matching. The Python implemented is structured with subclasses and methods, including some mind-bending double dispatch and other dynamic dispatch to stick with an OOP style even where your instructor thinks the functional style is easier to understand.

Use ‘assignment1.py‘ and ‘assignment1tests.py‘ files for reference.

Language Semantics

Our "language" has five kinds of values and four other kinds of expressions. The representation of expressions depends on the metalanguage (Python or your FP language), with this same semantics:

- A NoPoints represents the empty set of two-dimensional points.
- A Point represents a two-dimensional point with an x-coordinate and a y-coordinate. Both coordinates are floating-point numbers.
- A Line is a non-vertical infinite line in the plane, represented by a slope and an intercept (as in $y = mx + b$ where m is the slope and b is the intercept), both floating-point numbers.
- A VerticalLine is an infinite vertical line in the plane, represented by its x-coordinate.
- A LineSegment is a (finite) line segment, represented by the x - and y-coordinates of its endpoints (so four total floating-point numbers).
- An Intersect expression is not a value. It has two subexpressions. The semantics is to evaluate the subexpressions (in the same environment) and then return the value that is the intersection (in the geometric sense) of the two subresults. For example, the intersection of two lines could be one of:
- NoPoints, if the lines are parallel
- a Point, if the lines intersect
- a Line, if the lines have the same slope and intercept (see the note below about what "the same" means for floating-point numbers)
- A Let expression is not a value. It is like let-expressions in other languages we have studied or will stydy: The first subexpression is evaluated and the result bound to a variable that is added to the environment for evaluating the second subexpression.

- A Var expression is not a value. It is for using variables in the environment: We look up a string in the environment to get a geometric value.
- A Shift expression is not a value. It has a deltaX (a floating-point number), a delta Y (a floating-point number), and a subexpression. The semantics is to evaluate the subexpression and then shift the result by deltax (in the x-direction; positive is "to the right") and deltaY (in the y-direction; positive is "up"). More specifically, shifting for each form of value is as follows:
- NoPoints remains NoPoints.
- A Point representing (x, y) becomes a Point representing $(x + \text{delta}\,X, y + \text{delta}\,Y)$.
- A Line with slope m and intercept b becomes a Line with slope m and an intercept of $b + \text{delta}\,Y - m \cdot \text{deltaX}$.
- A VerticalLine becomes a VerticalLine shifted by deltaX; the delta Y is irrelevant.
- A LineSegment has its endpoints shift by deltaX and delta Y.

Note on Floating-Point Numbers

Because arithmetic with floating-point numbers can introduce small rounding errors, it is rarely appropriate to use equality to decide if two floating-point numbers are "the same." Instead, the provided code uses a helper function/method to decide if two floating-point numbers are "real close" (for our purposes, within .00001) and all your code should follow this approach as needed. For example, two points are the same if their x-coordinates are within .00001 and their y-coordinates are within .00001.

Expression Preprocessing

To simplify the interpreter, we first preprocess expressions. Preprocessing takes an expression and produces a new, equivalent expression with the following invariants:

- No LineSegment anywhere in the expression has endpoints that are the same as (i.e., real close to) each other. Such a line-segment should be replaced with the appropriate Point. For example, LineSegment(3.2,4.1,3.2,4.1) should be replaced with Point(3.2,4.1).
- Every LineSegment has its first endpoint (the first two real values) to the left (lower x-value) of the second endpoint. If the x-coordinates of the two endpoints are the same (real close), then the LineSegment has its first endpoint below (lower y-value) the second endpoint. For any LineSegment not meeting this requirement, replace it with a LineSegment with the same endpoints reordered.

The FP Code

You have to choose your favorite FP language and rewrite the interpreter. Our sample solution has about 220 lines of SML code. As always, line counts are just a rough guide.

Try to organize your code around a datatype-definition for expressions, functions for the different operations, and pattern-matching to identify different cases. The interpreter ‘eval_prog‘ uses a helper function intersect with cases for every combination of geometric value (so with 5 kinds of values there are 25 cases though some are handled together via pattern-matching). The surprisingly complicated part is the algorithm for intersecting two line segments.

The Python Code:

Most of the Python solution is given to you. All you have to do is look through it. We have defined classes for each kind of expression in our language, as well as appropriate superclasses. We have implemented parts of each class and left comments with what you need to do to understand the implementation.

Notice the Python code is organized around classes where each class has methods for various operations. All kinds of expressions need methods for preprocessing and evaluation. They are subclasses of ‘GeometryExpression‘ just like all constructors should be part of the ‘geom_exp‘ datatype (though the GeometryExpression class turns out not to be so useful). The value subclasses also need methods for shifting and intersection and they subclass ‘GeometryValue‘ so that some shared methods can be inherited (in analogy with some uses of wildcard patterns and helper functions in FP code).

Advice for Approaching the Assignment

- Understand the high-level structure of the code and how the Python file is structured in different ways before diving into the details.
- Approach the questions in order even though there is some flexibility.
- Because almost all the Python code is given to you, for much of the FP implementation, you can port the corresponding part of the Python solution. Doing so makes your job much easier (e.g., you need not re-figure out facts about geometry). Porting existing code to a new language is a useful and realistic skill to develop. It also helps teach the similarities and differences between languages.
- Be sure to test each line of your FP code. All languages and projects require testing. For example, Python will not even tell you statically if you misspell a method name or instance variable.

The Problems (Finally)

Note: we reserve the opportunity for reviewers to deduct points for incomprehensible code or a carelessly designed project.

1. **(20 points)** Implement an FP function ‘preprocess_prog‘ of type ‘geom_exp‘ → ‘geom_exp‘ to implement expression preprocessing as defined above. The idea is that evaluating program ‘e‘ would be done with ‘eval_prog‘ (‘preprocess_prog e, []‘) where the ‘[]‘ is the empty list for the empty environment.
2. **(20 points)** Add shift expressions as defined above to the FP implementation by adding the constructor Shift of real ∗ real ∗ ‘geom_exp‘ to the definition of ‘geom_exp‘ and adding appropriate branches to ‘eval_prog‘ and ‘preprocess_prog‘. (The first real is deltaX and the second is deltaY.)
3. **(30 points)** Complete the FP implementation. Do not modify the code given to you. Remember all the different cases in Python will appear somewhere in the FP solution, just arranged very differently.
4. **(30 points)** We strongly encourage you to provide tests for your functional programming solution.
5. (Lack Of) Challenge Problem: the most educational challenge problem is not something we can reasonably auto-grade or peer assess, so we encourage doing it even though it will not count toward your grade: Make a third version of your solution in a statically typed OOP language like Java or C#. Follow the structure of the Python solution, with no use of type casts or features like Java’s instanceof. You will need to have abstract methods and abstract classes that you then subclass. (Naturally, you can also enjoy the challenge of implementing your solution in any other programming language you like.)