Progress Report: Designing an Interpreter for a dynamic language for graph algorithms

Sandeep Dasgupta*

November 15, 2015

*Electronic address: sdasgup3@illinois.edu

1 Problem Statement

In this project we are planning to implement a dynamically typed language to represent graphs and apply various computations on them.

2 Language Syntax

The syntax of the language is an oversimplified version of C, but without mention of any types. The operations on incompatible types will be error-ed out while interpreting.

2.1 Progress

- We have implemented the tokenizer and syntax analyzer using flex and bison.
- We are supporting syntax like #include("filename") and #define("PI", "3.14") while doing a single pass of parsing (i.e. Preprocessing of these constructs are done while parsing). This is achieved by using flex's internal stack to manage multiple buffers.
- The grammar rules are borrowed from `http://www.quut.com/c/ANSI-C-grammar-y.html`. The rules are compiled by bison tool to generate the C parser.
- We are able to generate the AST corresponding to test-cases confirming to the grammar rules. Our AST is basically a list of function definitions. Each function definition object contains name of the function, a set of formal arguments and a list of body statements. These body statements could be an assignment, loop-statement, function call, etc. The leafs of the AST could be an identifier, int, float, true, false, null, string, vertex, edge or graph.
- Some of the key features of the parser is as follows:
  - Support of C statements like *if then*, *if then else*, *while*, *for*.
  - Support of *break*, *continue* within loop-body and *return* in function-body. As we are representing both loop-body and function-body as compound statements (i.e. anything between "{" & "}"), so we do not have to distinguish these two cases. But we will error-out if break is used inside non-loop body. The detailed semantics of executing a *break*, *continue* and *return* will be discussed in the interpreter runtime section.
  - Supporting graph as first class object *valuegraph*. The syntax to declare a graph is *g = graph();* which will be represented in AST as an assignment-node with left-node containing an identifier and right node as a function call. Now this function call corresponds to a built in function that returns a *valuegraph* (which is of one the leaf nodes of AST) on execution.
  - Supporting vertices and edges as first class objects which contains a map to add properties. This feature is useful in various graph algorithms like in dfs traversal we may use a vertex property "visited" to keep track of vertices already explored.

3 Language Semantics

The language semantics will be same as that of C as we are using a subset of it.

4 Interpreter Runtime

The following are the key features of the runtime:

- The runtime starts with searching for function definition *function main(argv)* and then creates a function call out of it and execute it. While creating the function call it uses the command-line parameters as the actual parameters of the function call.
- The execution of a function call involves finding the corresponding function definition, checking if the number of formal and actuals are equal and then pushing a call stack frame (which contains the mapping between the formal and actual values passed to them) in a global call stack. After that, the function is executed w.r.t the current context(i.e. the top of the call stack).
- The execution of the function involves executing a list of statements. The statements may add further mappings in the current call stack frame. Whenever a name (identifier) is refereed, the mapping in the current context need to be consulted to get the actual value of it.
- The semantics of *break*, *continue* or *return* is supported using the try-catch mechanism of C++.

  For example, while interpreting a *loop-body*, whenever a *break*is encountered, a corresponding *break-object* is thrown, which is caught in a place outside the entire loop execution in order to implement the semantics of break.

  Similarly, while executing a *loop-body*, whenever a *continue* is encountered, a *continue-object* is thrown, which is caught outside the *loop-body* execution so as to skip the current iteration and continue with the *loop-incr* execution (in case of for loop) and *loop-condition-expr* execution.

  And finally, while executing a *function-body*, which is a list of statements, when any one of those statements is a *return*, a *return-object* is thrown, which is caught outside of the loop which is going over that list and in this way the semantics of return is maintained.
- Occurrence of *break* and *continue* within non-loop body triggers an error. While interpreting a node-block (which is a set statements within "{" and "}"), whenever the runtime finds a *break* or *continue* it throws a object. Now if this object is caught inside a non-loop block then error is reported.
- Division by zero and operations on incompatible types are runtime errors.

5 Future Work

The following are the future work.

- To support additional operators to specify relationships between the nodes/edges or their groups. The operator that we are planning to support are the described next with their semantics.
  - v1←v2: Select all the directed edges from vertices v1 to v2.
  - v1→v2: Select all the directed edges from vertices v2 to v1.
  - v1↔v2: Select all the directed edges between vertices v1 and v1.
  - v1?v2: Select all the edges (directed or undirected) between vertices v1 and v1.
  - S1∩S2: Selects the intersection between the set of vertices/edges.

Most of these ideas behind choosing the operators are borrowed from Guess [2]. The intuition behind these operators is that the users do not have to remember longer commands. Also its very intuitive to build up complex operations using the simpler ones.

- To support saving of state and retrieving it back using routines like saveSateFromFile & loadStaeFromFile. This idea is borrowed from Guess [2] and this seems a useful service provided to do exploratory programming, as the user might be interested to checkout the last saved state (or a state with a any tag) or to undo all the experiments down to a particular state.

5.1 Evaluation Strategy

The baseline of our evaluation will be the open source Graphal [1] system. Graphal [1] is an interpreter of a programming language that is mainly oriented to graph algorithms. There is a command line interpreter and a graphical integrated development environment. The IDE contains text editor for programmers, compilation and script output, advanced debugger and visualization window. The evaluation will be between command line version of Graphal with our implementation. The evaluation will cover the following two aspects of our implementation:

- The convenience and intuitive extensibility provided by our programming model.
  - We will be implementing a couple of well know graph algorithms in both baseline and our implementation and use number of dynamic instructions interpreted as a measure to show the conciseness of our representation.
- Performance
  - As we are planning to keep the compiled version of frequently used graph routines (like dfs_iterators(vertex), bfs_iterators(vertex)), we are expecting to achieve better performance in terms of runtime.

References

[1] *Graphal: Graph algorithms interpreter.*

[2] E. Adar, *Guess: A language and interface for graph exploration*, in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '06, New York, NY, USA, 2006, ACM, pp. 791–800.