Designing an Interpreter for a dynamic language for graph algorithms

Sandeep Dasgupta*

October 28, 2015

*Electronic address: sdasgup3@illinois.edu

1 Motivation

As graphical models are increasingly being used in various fields like biological and physical sciences , organizational structures, social networking, there is a need to represent the various graphical models in an efficient way. This involves (but not limited to) the following

- Designing a language which provide an easy interface to the programmer to program those models. An easy interface involves:
  - Expressive power of the language representing those models.
  - Intuitive extensibility of the language.
  - Ability of the language to provide exploratory programming, where the user may experiment with different ideas before coming to a conclusive one.
- The language implementation should be efficient in the following sense.
  - the underlying data structure need to be carefully chosen so that the test program runs reasonably as compared to existing static compilers.
  - the implementation need to be scalable w.r.t the memory requirement.

To meet one of the above goals like exploratory programming we decided to work an a dynamically typed language so that the user do not have to worry much about declaring types and focus on his/her experiments.

To meet this we introduces GRI (Graph Interpreter), which is an Interpreter to for a dynamic language well suited to represent graphs and apply various algorithms on it. The dynamic language has the following features:

- As "nodes" and "edges" are the backbone of any graph structure, we have them as first-class objects. Even though the user will not necessarily be accessing individual nodes, but the option is available to them.
- Users have the option to specify attributes on nodes and edges and later use those attributes for computing results. For example: to find all the red colored nodes (where color is marked as an attribute of node) connected to a specific node.
- The language supports operators to specify relationships between the nodes/edges or groups of them. For example, the user can create sub graphs like

$$department\_1 = (N1, N2, N3) \tag{1}$$

$$managers = (N3, N4, N5) \tag{2}$$

  and use operators like $\cap$ to find nodes "department 1 employees who are managers as well". Also the language supports operators to find edges between groups of nodes. This will answer interesting queries like:

$$managers \rightarrow (department1 - managers)$$

  could mean "the number of email conversation between managers to department_1 non managers"

  Most of these ideas behind choosing the operators are borrowed from [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.
- Also borrowed from [2], is the option of saving state of the graph. This seems a useful service provided to do exploratory programming, as the user might be interested to check-out the last saved state (or a state with a any tag) or to undo all the experiments upto a particular state.

2 GRI Interpreter

Proposed language does not allow the user to specify the types and a variable may point to objects of different types during run time(and hence a dynamically types language). In this scenario we could do a static compilation after a static type inference to generate sub optimal machine code, whose efficiency depends on how well the types are inferred.

As opposed to that, we are trying to build an interpreter which will interpret the AST spit out by the parser. If time permits, we are planning to convert the the AST into LLVM byte code to be either interpreted or Jitted by the LLVM tool-chain.

3 Language

The syntax of the language is an oversimplified version of C. Also planning to provide a rich set of built in functions like *deleteVertex(graph, vertex), dfs_iterators(vertex), bfs_iterators(vertex)* which will facilitate writing complex algorithms. Some of the ideas of the language design will be borrowed from [1].

The language will distinguish between user variables and variables representing say graph objects. The later will be treated immutable and an error will be issued if the user tried to modify this. For example, a graph object g, representing a graph, cannot to modified (for example, by assigning g = 5;). In that sense the "dynamically typed" nature of the language will be restricted to the user variables.

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.