PLOG Proposal: A programming language for graph operations

wab2125@columbia.edu
COMS 4115 Summer session - Project proposal

June 9, 2016

Introduction:

Graphs are familiar structures used throughout mathematics and computer science– structures containing elements commonly known as nodes and edges. In practice, a particular node typically refers to a particular (e.g. physical or conceptual) entity, and an edge might refer to a particular kind of relationship existing between two nodes. Properties are also frequently assigned to nodes and edges (e.g. as "keys" and their associated values). The resulting structure is a property graph, which can be modeled and operated on with PLOG.

PLOG, the Programming Language for Operations de Graphes, was designed as an approach to modeling with property graphs. It allows users (of the language) to perform graph queries and updates concisely with pattern-matching, and to enforce "rules" upon graphs, but still allows users to specify "lower-level" algorithms. It's intended for simple applications that wish to model and interact with data as property graphs, e.g. for (establishing flow rates in) models of computer networks, (establishing relationships and inferring patterns of) social networks, et cetera.

The remainder of this document describes aspects of the PLOG language in some detail. It should be noted that the constructed compiler for PLOG will target the Python language.

Parts of PLOG:

PLOG programs are composed of 2 main segments of code: 1) type and global object definition, and 2) the main procedure.

1) Type and global object definition: This segment of code is essentially the global scope of the program. All types—i.e. edge types and node types—are defined globally. Additionally, functions (if applicable) must be defined in this global scope; nested functions are not permitted. Optionally, graph objects may be defined in this scope.

2) Main procedure: The "main" procedure is where program execution takes place. Throughout the main procedure, global types and objects may be referenced and accessed, functions may be called, and local graph objects (i.e. graphs, nodes, edges) may be created, referenced, updated, and deleted. Additionally, variables of primitive types (e.g. int, float) can be used.

—Primitive types:

PLOG supports the following common types—with their usual meanings—"natively":

- bool, char, int, float, string

And additionally supports lists and dictionaries ("dicts"; key-value stores). Lists can either be empty ('[]') or contain elements of exactly 1 type. Dictionaries map keys—all of 1 type—to their corresponding values.

—**Graph element types:**

node: Node types can be created. A node type is defined by a name and an associated pattern (discussed below, in "Pattern-matching"). Node types allow queries and updates to graph objects to be stated concisely. For example, the following code defines a "grandparent" node type, assuming a "parent" edge is already defined (where "X parent-> Y" means: "X's parent is Y"):

```
node grandparent = z in x parent-> y parent-> z
```

In the above example, a "grandparent" is defined as any node which is the parent of a parent. We will see soon how such node (and edge) types can be used.

edge: Similarly, new edge types can be created. An edge type is given by its name, e.g.:

```
edge parent, ancestor
```

declares two new edge types: "parent" and "ancestor". (As stated before, all types are defined globally.)

—**Graph and graph element instances:**

Node and edge types may be defined globally, but instances of node and edge types must exist within a graph. A graph object is declared with the "graph" keyword, and may be initialized with nodes and edges within brackets, as below:

```
graph myGraph { node Jimmy }
```

which initializes "myGraph" to contain a single node, "Jimmy".

Nodes can explicitly be added to graphs by declaring them within the brackets following a graph name. For example, adding two more nodes to the above graph can be accomplished with

```
myGraph { node Craig, Jennifer }
```

which updates "myGraph" to now contain the 3 declared nodes.

Edges must be declared "between" two node names (the tail node and the head node). For example, given the "parent" edge declared earlier,

```
myGraph { Jimmy parent-> Craig, Jennifer }
```

adds two edges to myGraph: one "parent" edge from Jimmy to Craig (tail to head), and another "parent" edge from Jimmy to Jennifer (tail to head).

Nodes and edges can be deleted with the "del" operator, as in:

```
myGraph { node NewGuy; Jimmy ancestor-> NewGuy; del Jimmy ancestor-> NewGuy }
```

The deletion of a node also deletes all of the edges directly connected to it.

—**Type inference (basically, none of it):**

Type inference is essentially not carried out by a PLOG compiler; variable types must be explicitly stated upon declaration. In particular, lists and dictionaries must also be declared with the element type they contain (for lists) or the key type they contain (for dicts).

—**Control structures and loops:**

PLOG supports common 'if/else' branching and 'for/while' looping. Both 'if' and 'while' statements require an expression evaluating to a boolean value (as is typical), while the 'for' loop makes use of the 'in' keyword to loop over variables of a specified type within a given graph. Note: it's not required that the specified type be the only type of object within the graph; the loop will iterate over only objects of the specified type within the graph. For example:

```
for node n in myGraph { n.iteratedOver = true }
```

—**Properties:**

Properties can be explicitly added to nodes, e.g. as in:

```
myGraph { Jimmy.age = 10; Craig: age = 46, gender='M' }
```

Dot ('.') notation allows the setting/getting of individual properties by their name, and colon notation allows you to specify multiple properties (their names and values) at once. (Note: separate statements can be executed on the same line if separated by a semi-colon; otherwise, a newline indicates the end of a statement.)

Edges can also possess properties. Assuming the "loves" relation was defined,

```
myGraph { Craig <-l/loves-> Jennifer where l.howMuch = ``a lot'' }
```

declares two new edges: "Craig loves-> Jennifer" and "Jennifer loves-> Craig", and assigns the property "howMuch" to have the value "a lot" for both edges.

—**Pattern-matching:**

PLOG supports pattern-matching for queries and updates performed on graph objects. Querying a graph object—as given below—with a pattern will essentially return a "(sub)graph view" of the original graph object. Effectively, this returned (sub)graph view is a graph object, but only with references to the pattern-matched elements of the original graph. This querying/pattern-matching is evoked by:

```
graph mySubgraphView = graphName( pattern )
```

where 'pattern' is the pattern to match against the graph (i.e. its elements). For example:

```
graph mySubgraphView = myGraph( e in x e/parent-> y )  // Graph of only parent edges
for edge e in mySubgraphView { e.asOf = ``2016-06-07'' }
```

—**"Update rules":**

"Update rules" can be defined at the global level, which can effectively constitute a kind of "extended definition" of certain types. They can be thought of as conditions that are always checked and rules that are enforced. For example, in addition to the "edge parent, ancestor" declaration described above, the following rules can be placed in the global scope:

```
if x parent-> y  { x ancestor-> y }
if x ancestor-> y and y ancestor-> z  { x ancestor-> z }
```

This is taken to mean: any time there is a node 'x' whose parent is 'y', the edge "x ancestor-> y" is made sure to exist: if it doesn't already exist in the graph of 'x' and 'y', it is created. The second rule essentially enforces the transitive nature of the intended "ancestor" relationship.

—**Functions:**

Functions may be defined at the global scope. Defining a function consists of: 1) the 'func' keyword, 2) the function name, 3) (optionally) declaring the function's input variable(s) and their types (enclosed in parantheses), 4) (optionally) declaring the function's return type(s), and 5) the function body, enclosed with braces.

Two example function definitions are given:

```
func renameAllHumans( graph g, string newName )
{
    for node n in g( x.genusSpecies = ``Homo sapiens'' ) { n.name = newName }
}

func getDummyIntsAndString() return int list, string list
{
    return [0,1,2,3,4], [``Zero'', ``One'', ``Two'', ``Three'', ``Four'']
}
```

Example Application:

```
rel distance

graph Neighborhood {
   node Me, Jimmy, Sally, Bobby, Tiffany
   Me d/distance-> Jimmy, Sally where d.value = 6
   Me d/distance-> Bobby where d.value = 2
   Bobby, Sally d/distance-> Tiffany where d.value = 1 }

func DAlg ( graph mygraph, node start, node end ) return int dict, node dict
{
   node list Q = list( for node n in mygraph )  // list() creates a list from 'for'

   int dict distances; node dict previous
   for node n in Q {
      distances[n] = INF  // ``Infinity'', a defined constant
      previous[n] = NIL   /* ``Null'', a defined constant */  }
   distances[start] = 0   // More dict setting

   while length(Q) > 0 {  // length() returns list length
```

```
      node closest
      int min_dist = INF
      for node n in Q {
         if distances[n] < min_dist {
            min_dist = distances[n]
            closest = n } }
      if closest == end { break } // break exits ‘‘nearest’’ loop
      Q = remove(closest, Q)      // remove() returns a list minus the element

      for node out in closest.out_nodes {
         int out_dist = d.value in closest d/distanceTo out
         int new_dist = distances[closest] + out_dist
         if new_dist < distances[out_dist] {
            distances[out] = new_dist
            previous[out] = closest } }
   }
   return distances, previous
}

func main {
   int dict distances, node dict previous   // Next line for purposes of this document..
      = DAlg( Neighborhood, Neighborhood(Me), Neighborhood(Tiffany) )
   node list S = []
   node n = Neighborhood(Tiffany)
   while previous[n] != NIL {
      S = prepend(n, S)                // prepend() adds to the beginning of a list
      n = previous[ n ] }
   S = prepend(n, S)

   for node n in S { print( string(n) + ’ ’ ) }  // print() prints
}
```