Knowledge Graph Language (KGL)

September 30, 2015

Team

<table>
<thead>
<tr>
<th>Name</th>
<th>UNI</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bingyan Hu</td>
<td>bh2447</td>
<td>Project Manager</td>
</tr>
<tr>
<td>Cheng Huang</td>
<td>ch2994</td>
<td>Language Guru</td>
</tr>
<tr>
<td>Ruoxin Jiang</td>
<td>rj2394</td>
<td>System Architect</td>
</tr>
<tr>
<td>Nicholas Mariconda</td>
<td>nm2812</td>
<td>Verification &amp; Validation</td>
</tr>
</tbody>
</table>

Motivation

Almost everything in the world is connected together through some complex web of relationships. As such, building, expressing and traversing graphs is one of the most essential applications of computer science. However, it is common knowledge that implementing graphs in traditional languages is no trivial task. Many past projects have addressed this problem by designing graph-based languages that make building graphs easier. However, such projects were limited by having single, fixed relationships between nodes. Often, algorithms that operate on real-world data – such as machine learning and information retrieval algorithms – are too obfuscated to be represented by a graph with one-dimensional relationships.

Proposed Uses

Knowledge Graph Language (KGL) is a domain-specific graphing language that supports multiple user-defined relationships between nodes. Edges, nodes, and graphs are built-in types of the language; however, two nodes can be connected by multiple edges, with each edge being identified by a unidirectional, user-defined relationship. KGL reaps many of the benefits of a graphing domain-specific language – users can build, express and traverse complex graphs succinctly – while also providing a means for users to query their graphs directly. This is the main thrust of the language – by providing the users with a mechanism for defining their own relationships between nodes, they can extract a more robust collection of data through graph queries.
Syntax

Data Types

- **Primitive Types**
  - `int` an integer
  - `float` a floating point number
  - `boolean` data type that has only two values: true and false
  - `char` a character
  - `string` plain text encoded in ASCII

- **Graph Related Types**
  - `Node` a node in a graph, storing data such as name, an array of outgoing edges, an array of incoming edges and other attributes of the node
  - `Edge` a directed edge in a graph, storing data such as label, source node, target node and attributes of the edge
  - `Graph` a directed, multi-relational graph, representing a collection of nodes and edges

- **Other Types**
  - `array` an ordered sequence that can change in size
  - `attribute` a key/value map, e.g. `{key1: value1, key2: value2}`

Operators

- **Basic Operators**
  - `>`, `<`, `<=`, `>=`, `==`, `!=` comparison operators for basic types
  - `!`, `&&`, `||` logical NOT, AND, OR for boolean type
  - `*`, `/`, `%`, `+`, `-` arithmetic operators for int and float
  - `=` assignment operator

- **Graph-related Operators**
  - `(object).(member)` member access, e.g. `graph.allEdges()`
  - `==`, `!` `==` `/!` return true if two nodes/edges are identical/ not identical
  - `node1 -(label)- > node2` defines an arc from node1 to node2
  - `node1 < -(label)- > node2` defines an undirected edge between node1 and node2

Control Flow

- `;` end of a statement
- `#` start of a single line comment, e.g. `# Comment here.`
- `/# ... */` start/end of a block comment
- `while` while loop, e.g. `while (loop invariant) { loop body }`
- `for` for loop, e.g. `for (loop invariants) { loop body }`
Build-in functions

• Graph

- `allEdges()` return all edges in the graph
- `allNodes()` return all nodes in the graph
- `addNode(name, attributes)` create a new node in the graph
- `addEdge(source, target, label, attributes)` add a new edge in the graph
- `deleteNode(node)` delete the given node from the graph
- `deleteEdge(edge)` delete the given edge from the graph
- `getNode(name)` get the node with the given name
- `countNodes()` return the total number of nodes
- `countEdges()` return the total number of edges
- `print(graph)` print all the nodes and edges in this graph using `print(node)` and `print(edge)` described below

• Edge

- `getSourceNode()` return the source node
- `getTargetNode()` return the target node
- `getLabel()` get the label of this edge
- `getAttributes()` return all attributes of the edge
- `getAttribute(key)` get the value of the given key from the edge’s attributes
- `addAttributes(attributes)` add new attributes (a key/value map) to the edge
- `setLabel(label)` set the new label of this edge
- `print(edge)` prints the source and targets nodes using `print(node)` along with the attributes

• Node

- `getName()` return the name of this node
- `getOutgoingEdges()` get all outgoing edges
- `getOutgoingEdges(label)` get all outgoing edges with the given label
- `getNeighbors()` get all neighboring nodes
- `getNeighbors(attribute)` get all neighboring nodes with the given attribute
- `getAttributes()` get all attributes of the node
- `getAttribute(key)` get the value of the given key from the node’s attributes
- `addAttributes(attributes)` add new attributes (a key/value map) to the node
- `print(node)` prints all attributes of the node

• Attribute

- `getKeys()` return all keys in the attribute
- `getValue(key)` return the value of the given key in the attribute
- `insert(key, value)` insert an key/value pair in the attribute
- `delete(key)` delete a key/value pair from the attribute
- `size()` return the number of key/value pairs in the attribute
- `clear()` clear all key/value pairs in the attribute
Program Structure

- Function

```cpp
func ret_type fname(type1 arg1, type2 arg2, ...)
```

function declaration

```cpp
ret_type fname(type1, arg1, type2, arg2, ...) {
    declarations
    statements
    return value
}
```

function definition

```cpp
main() {
    declarations
    statements
}
```

main routine

- Graph

```cpp
Graph g = {
    a --(knows)--> b;
    b --(knows)--> c;
    a --(likes)--> d;
    c --(likes)--> d;
}
```

graph definition (1)

```cpp
Graph g;
g.addNode("a", {}); g.addNode("b", {});
g.addEdge("knows", "a", "b", {});
```

graph definition (2)

Sample Code

```cpp
func Node [] findAll2ndConnections(Graph g, Node start) {
    Node [] firstConnections = start.getOutgoingEdges("knows").getTargetNode();
    Node [] secondConnections;
    for (Node n in firstConnections.getOutgoingEdges("knows").getTargetNode()) {
        if (! n in firstConnections && n != start) {
            secondConnections = secondConnections + n;
        }
    }
    return secondConnections;
}
```

```cpp
main() {
}
```
Graph g = {
    a --(knows)--> b; a --(knows)--> c; b --(knows)--> c;
    b --(knows)--> d; b --(knows)--> f;
    c --(knows)--> e; c --(knows)--> f;
}
Node [] connections = findAll2ndConnections(g, g.getNode("a"));
print connections