Nicole Maines, Richard Harker, Shayne Hayes, Son Huynh

Dr. Karim Sobh

CMPS 109

17 March, 2017

SRI Report (v3)

Build Instructions:

(Note: These instructions apply to both the SRI Client and the SRI Server)

To build -- “make SRI”

To run -- “SRI”

(If running the client and server on the same machine, use 127.0.0.1 to connect to the server)

To delete .o files and executable -- “make clean”

To run valgrind -- “make check”

Design Decisions:

Central to our program is our Group class, which is a data structure that stores pointers to Names. We decided to use pointers for names rather than strings in order to reduce memory usage, since it is quite likely that the same name will be repeated several times across the Knowledge Base and Rule Base. Groups are able to be compared to other groups in several manners; most importantly, function match() compares the non-parameter entries of two Groups to determine if their non-parameter entries are equal. This method is at the heart of our queries. To find groups that fill in the parameter blanks of this query group, all we have to do is call the match() function of other groups. It’s a very flexible system that allows for any sort of filtering!

Originally our Group class contained a vector of Names and a vector of corresponding positions. However, we replaced this approach with the parameter placeholders for several reasons. First, parameter placeholders are temporary and are only ever used during inferences. Second, simply using a placeholder to fill a positional gap is much more space efficient than storing an integer for every name in the group. Third, using placeholders is more intuitive to use and makes our code much more readable than before. Finally, filtering parameters during inferences (like Inference Fact1($X,$X)) would not be possible in our previous approach to Group.

Group also contains a method central to OR parameter substitution: reorder(). Reorder will return a new group that contains the members of another group, but reordered according to two vectors of ints. For instance, if we have a group {a,b,c} and we call reorder({0,1,2},{1,2,0}), a new group {b,c,a} will be returned. Because the parameters of the predicates of OR are independent and each predicate is guaranteed to contain all of the the input parameters, we may simply call reorder() on our query group to rearrange the group, pass this group to the query() function of the predicate, and then call reorder() again on the returned results to rearrange the groups back to the input format. Pretty slick, eh? I suppose the alternative to this would be to perform an unfiltered query on each predicate and only then filter the results and construct groups to return out of certain columns of names. However, this approach is much less efficient since many more groups will be returned and compared to one another.

This brings us to the parameter substitution rules themselves. In rule we store a vector of ints for each predicate, with the parameter input being stored in params[0]. Each parameter has a corresponding int value in the 2D vector. For example, Rule1($X,$Y,$Z):- OR Rule2($Z,$Y,$A)

Rule3($B,$Y,$X,$C) is stored as {{0,1,2},{2,1,3},{4,1,0,5}}. We store the parameter substitution pattern this way for several reasons. First, these vectors of ints may be passed directly to reorder(). If reorder encounters a number in its second vector that is not in the first vector, it will add a parameter placeholder to the group it returns, which is exactly what we want for querying. Second, storing ints uses less memory than storing strings. Third, this representation is easy to read and use. *Technically* this limits our number of parameters to 4,294,967,295 parameters, but come on. (If we wanted to support more parameters for some really bizarre reason, all we would need to do is change int to string. Problem solved!)

During logical AND inferences, query() calls a function called queryStep() that will recursively build a list of groups to return. It does this one group at a time rather than one predicate at a time, building what can be thought of as a tree of solutions to the AND rule. Our original design for AND was iterative rather than recursive -- we would store the results of each predicate and build a tree of indices that would define the various permutations that would need to be constructed and returned. However, this approach proved to be far too difficult to implement correctly and only worked under the assumption that each predicate relies only on the previous predicate's results (rather than all of the previous predicates' results).

Our multithreading of the OR and AND operators is fairly simple -- we use the provided Thread class to process all predicates in parallel for OR by creating a new Thread for each predicate, and pipeline the predicates for AND by creating a new Thread for each group. We considered multithreading some of the functions in Fact or Group (such as equals), but thought it overkill. After all, only so many threads can be processed at once in the first place.