Architecture of the QSemanticDB implementation

Rehno Lindeque

May 16, 2010

Abstract

The architecture an OpenSemanticDB reference implementation is described.

1 Introduction

1.1 Terminology

- *Semantic symbol / symbol* – The conceptual representation of a semantic value. For example a could be a symbol and so could $a.b$
- *Semantic id / id* – The internal representation of some semantic symbol.
- *Fully qualified id* – An id that represents some semantic symbol inside some context.
- *Unqualified id* – An id that represents some semantic symbol disregarding the context it is found in. (I.e. $a.(b.c)$ is qualified, but c or even $b.c$ is unqualified with reference to a).
- *Relation* – A declaration in the form $a \rightarrow b$ which is represented by the existence of the query $a.b$.
- *Unqualified relation* – A pair of id's $(domainId, codomainId)$ such that $codomainId$ is unqualified with respect to $domainId$.

1.2 Data structures

The following basic indexes are needed.

- SplitRelations: $qualifiedCodomain \rightarrow (domain, unqualifiedCodomain)$
- UnifyRelations: $(domain, unqualifiedCodomain) \rightarrow qualifiedCodomain$
- DomainQCodomains: $domain \rightarrow qualifiedCodomain$
- DomainUCodomains: $domain \rightarrow unqualifiedCodomain$

1.3 Special relations

In order to represent certain operations and syntactic sugar, certain special structures must be reserved.

1.3.1 Anonymous symbols

An anonymous symbol is syntactic sugar that allows a user to select from a set without binding it to its own symbol. Hence the database must assign automatically generate an anonymous symbol for the set. Furthermore, anonymous sets can be nested inside other queries and sets, so the evaluator needs to ignore these symbols when they are not used in a query.

An anonymous set is structured as follows:

$anonymous \rightarrow HIDDEN \rightarrow \{x\ y\ z\}$

where *anonymous* is an automatically generated id and $HIDDEN$ is a reserved id used to prevent the evaluator from evaluating *anonymous* or any of its codomains (aside from the query that uses the symbol).

Hence, a query such as `result = { { x y z }.z }` , will be represented as:

$result \rightarrow \{anonymous \rightarrow HIDDEN \rightarrow \{x\ y\ z\}\} \xrightarrow[speculative]{} x$

1.4 Scheduler

In order to perform queries with undetermined outcomes a scheduler is needed. This allows the interpreter to perform queries ahead of the evaluation steps that returns the string of symbols to a client program.

1.4.1 Operation

The scheduler works in the following manner:

- First, when the interpreter recieves its first symbol it is pushed on to the schedule.
- The evaluator pops the first available symbol from the schedule. If this is the last symbol in the first branch in the schedule, then the codomain of the symbol is pushed onto this branch.
  To clarify: There will be the root node and several branches from the root. The evaluator always pops from the first branch from the root. Every branch may also have sub-branches. Once the first branch has been popped from the schedule, it is replaced with its sub-branches. If the first branch has no sub-branches and only

one symbol remains, then this symbol is evaluated to add new symbols/branches to the first branch before being popped. However, if this symbol has no codomains (I.e. it is an atom) then the first branch will be removed entirely to be replaced by the next branch which now becomes the "first" branch.

- If the symbol is a query then its first concrete domain is pushed onto the schedule. The query is evaluated until the codomain is either found or the string ends. Then the whole thing is either commited or rolled back.

- todo...

1.4.2 Pseudo-code

```
proc Eval()
    endproc
```

1.4.3 Parallel evaluation

The design of the scheduler is intended to make parallel execution easier later on: Currently only the "first" root branch gets evaluated until it is completed. However, it is permissible to evaluate every root branch concurrently (in their own environment) until they are completed. If a branch is replaced by its sub-branches, more threads may be generated to take advantage of the added parallelism. Note that this is only theoretic ideas however. It is unlikely that the interpreter itself will be parallelized since its primary purpose is to eventually generate compiled bytecode. Queries that cannot be resolved in "compile-time" will probably make use of similar scheduling ideas however.