Initial Value Computation

August 2, 2011

1 Definitions

In this draft we want to explain the notion of domains. The domain of a variable is the set of values that it can take. The domain of all the variables in a query expression can easily be computed recursively if some rules are respected. We will use through out the entire paper the notation of $\mathsf{dom}_{\vec{x}}(q)$ for the domain of a set of variables, where q is the given query and $\vec{x}$ is a vector representing the variables(not necessarily present in the expression q). Taking into account the expressions from the AGCA (Aggregate Calculus[**?**]) the definition of an expression will be:

$$q \text{ ::- } q \cdot q | q + q | v \leftarrow q | v_1 \theta v_2 | R(\vec{y}) | \text{constant} | \text{variable}$$

We start with the definition of $\mathsf{dom}_{\vec{x}}(R(\vec{y}))$:

$$\mathsf{dom}_{\vec{x}}(R(\vec{y})) = \left\{ \vec{c} \,\middle|\, (\prod_{i:x_i \in \vec{y}} (x_i \leftarrow c_i) \cdot R(\vec{y})) \neq 0 \right\}$$

where $\vec{x} =< x_1, x_2, x_3, \cdots, x_i, \cdots >$, $\vec{c} =< c_1, c_2, c_3, \cdots, c_i, \cdots >$. $\vec{x}$ defines the schema of $\vec{c}$, specifically, $\vec{x}$ is a vector of names of each element of the tuple $\vec{c}$. It is not necessary that $\vec{x}$ has the same schema as the given expression. Thus, we can evaluate $\mathsf{dom}_{\vec{x}}$ for a broader range of $\vec{x}$ and it is not restricted by the schema of the input query expression. In such cases the dom is infinite as the not presenting variables in the query can take any value.

For the comparison operator $(v_1 \theta v_2)$, where v_1 and v_2 are variables, we can compute the domain as follows:

$$\mathsf{dom}_{\vec{x}}(v_1 \theta v_2) = \left\{ \vec{c} \,\middle|\, \Big((\prod_{i} (x_i \leftarrow c_i)) (v_1 \theta v_2) \Big) \neq 0 \right\}$$

The domain of a comparison may be infinite. For the join operator we can write:

$$\mathsf{dom}_{\vec{x}}(q_1 \cdot q_2) = \{\vec{c}|\vec{c} \in \mathsf{dom}_{\vec{x}}(q_1) \wedge \vec{c} \in \mathsf{dom}_{\vec{x}}(q_2)\}$$

while for the union operator the domain definition is very similar:

$$\mathsf{dom}_{\vec{x}}(q_1 + q_2) = \{\vec{c}|\vec{c} \in \mathsf{dom}_{\vec{x}}(q_1) \vee \vec{c} \in \mathsf{dom}_{\vec{x}}(q_2)\}$$

$$\mathsf{dom}_{\vec{x}}(constant) = \left\{\vec{c}\right\}$$

$$\mathsf{dom}_{\vec{x}}(variable) = \left\{\vec{c}\right\}$$

Finally, we can give a formalism for expressing the domain of a variable that will participate in an assignment operation:

$$\mathsf{dom}_{\vec{x}}(v \leftarrow q_1) = \left\{\vec{c}\middle|\vec{c} \in \mathsf{dom}_{\vec{x}}(q_1) \wedge \left(\forall i : (x_i = v) \Rightarrow \left((q_1 \cdot \prod_{j : x_j \neq v} (x_j = c_j)) = c_i\right)\right)\right\}$$

In fact using implication operator in the above definition allows us to extend the $\vec{x}$ to whatever vector we want, as we already said the schema of $\vec{x}$ is not necessarily the same as the schema of q.

Having the definitions for the domains, we will try to give some insight regarding to the notion of arity. We will start with an example relation:

$$q = R(a, b) \cdot S(b, c)$$

the schema for q will have three variables (a, b, c). The arity of a tuple is the number of its occurrences in the relation, in other words the order of multiplicity of that tuple. The arity of a tuple will increase or decrease if insertions or respectively deletions will be made to a relation. For example:

R	a	b	arity
	<1	2>	1
	<1	3>	2
	<3	4>	1

Table 1: Relation R

S	b	c	arity
	<1	1>	1
	<2	2>	2
	<3	5>	2

Table 2: Relation S

$R \cdot S$	a	b	c	arity
	<1	2	2>	2
	<1	3	5>	4
	<3	4	$*>$	0
	$<*$	1	1>	0

Table 3: Relation $R \cdot S$

We need to maintain the maps for certain values as long as the arity of a tuple is greater then 0. If the arity of the tuple drops to 0 then the tuple will not be taken into consideration and therefore it can be eliminated from the domains of the maps.

We need to store the arities inside each map and also way to compute the arity of an AGCA expression. Since we substitute the subexpressions with maps, we can easily consider the relations as the maps without input variables. Also a map with input variables can be seen as a relation with a group-by clause. Input variable of a map bind some variables. Thus if we compute the map values by all different combinations of these variables, we will look up into these values and return the appropriate value according to the input variables.

2 Computing the arities of the AGCA expressions

We define the function *arity* which will be used to compute the arity of a tuple in a certain relation. The function will be defined on the relation and the tuple for which the multiplicity order is desired to be computed.

$$arity(\text{Relation } q, \text{Tuple } t) = \text{multiplicity order of tuple } t \text{ in the relation } q$$

$$arity(q,t) = \pi_t(q)$$

where $\pi_t(q)$ means the projection of relation q for the tuple t. This function can be used for the computation of the arity of the expressions from the AGCA: q ::- $q{\cdot}q$ | $q{+}q$ | $q\theta t$ | $t \leftarrow q$ | constant | variable. However constants and variables can be eliminated from the computation because relations are of interest.

$$arity(q_1 \cdot q_2, t) = \sum_{\{t_1\} \bowtie \{t_2\} = t} arity(q_1, t_1) * arity(q_2, t_2)$$

$$arity(q_1 + q_2, t) = arity(q_1, t) + arity(q_2, t), \text{ where Schema}(q_1) = \text{Schema}(q_2)$$

$$arity(v_1 \ \theta \ v_2, t) = \begin{cases} 1, & \text{if } v_1 \theta v_2 \text{ is true and } v_1 \in t \wedge v_2 \in t \\ 0, & \text{otherwise is false} \end{cases}$$

$$arity(v \leftarrow q, t) = \begin{cases} 0, & \text{if } t \setminus \{v\} \notin \mathsf{dom}_{Schema(t)}(q) \\ 1, & \text{otherwise} \end{cases}$$

3 Domain computation

We are trying to compute the domains for each variable that appears in queries. This computation is performed on the query decomposition tree. We can traverse the tree in post order, first visiting the leaves that are represented by some relations and afterwards visiting the parent nodes and combining the relations of the children nodes. Using this technique we are trying to compute the domains, but also the vector $\vec{x}$ of all the variables defined in that query.

The intuition is simple. MMM ...

The algorithm will need as inputs the root of the tree and a structure that must be previously defined and will return a structure that contains the vector of variables(x) and the domains for each variable defined in the vector(dom). The structure will look like:

```
struct{
x: the vector of all the variables
dom: the domains of all the variables
}
```

In 1 we gave the function which compute this structure for the whole tree. We call it on the root and with an empty structure for s, computeDomain($root$, NIL).

Algorithm 1 computeDomains($node$,s)

Input: $node$ as the root of the tree to be traversed, and s as the structure
Output: a structure $result$ that will contain the vector of variables $\vec{x}$ and the domains of each variable $\mathsf{dom}_{\vec{x}}(query)$

```
1:  if $node.type =$ "+" then
2:      $s_1 \leftarrow$ computeDomain($node.left, s$)
3:      $s_2 \leftarrow$ computeDomain($node.right, s$)
4:      $s_3.\vec{x} \leftarrow s_1.\vec{x} \cup s_2.\vec{x}$ // this will compute the vector for all the variables,
        both from the right and left node
5:      $s_3.dom \leftarrow \mathsf{dom}_{s_3.\vec{x}}(node.left + node.right)$
6:      return $s_3$
7:  else if $node.type =$ "*" then
8:      $s_1 \leftarrow$ computeDomain($node.left, s$)
9:      return computeDomain($node.right, s_1$)
10: else
11:     if $s =$ NIL then
12:         $s_1.\vec{x} \leftarrow$ all the variables of the $node$
13:         $s_1.dom \leftarrow \mathsf{dom}_{s_1.\vec{x}}(node)$
14:     else
15:         $s_1.\vec{x} \leftarrow s.\vec{x} \cup \{$all the variables of the $node\}$
16:         $s_1.dom \leftarrow \mathsf{dom}_{s_1.\vec{x}}(s.dom * \mathsf{dom}_{s_1.\vec{x}}(node))$
17:     end if
18:     return $s_1$
19: end if
```

The order of Algorithm 1 is $O(n \cdot P)$ where n is the number of nodes in the query decomposition tree and P is the time needed for any rule of dom's computation, according to previous section.

In Algorithm 1 we suppose that there are three types of nodes. Union nodes, join nodes and null nodes. Null nodes do not have any children. A union node or a join node always has two children so line 10 will be executed when the node is a leaf. Type field refers to the type of nodes. Also, since any node represents a query expression then we can obtain its domain with dom operator as was mentioned in the previous section.

The “if” in line 11 is used to initialize structure s for the first time(happens in the leftmost node of the tree).

Theorem 1. *Algorithm 1 compute the domains and the variables list of the given tree in $O(n \cdot P)$ correctly.*

Proof. $\square$

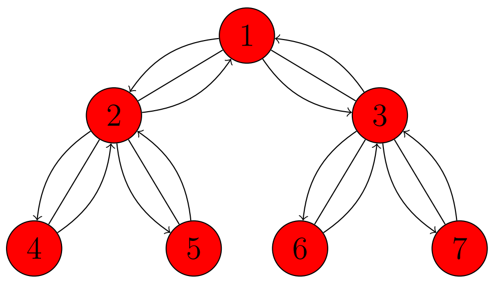

Figure 1: Tree traversal