Initial Value Computation

July 29, 2011

1 Definitions

In this draft we want to explain the notion of domain. The domain of a variable is the set of values that a variable can take. The domain of all the variables in a query expression can easily be created recursively if some rules will be 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 $\vec{x}$ is a vector representing the variables 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 will start with the definition of $\mathsf{dom}_{\vec{x}}(R(\vec{y}))$:

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

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 the vector of names of each element of the tuple $\vec{c}$.

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

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

The domain of a comparison is infinite. In fact using implication operator in the above definition allows us to extend the $\vec{x}$ to whatever vector we want. 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 delimited by the schema of the expression. In such cases the dom is infinite as the other variables can take any value. 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:

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

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 will be the number of 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

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

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 $R \cdot S$

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. We need a way to compute the arity of an AGCA expression. Since we substitute the subexpressions with maps, we can easily consider the maps without input variables as some relations. 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 expression

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.

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

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

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 ::- $q{\cdot}q \mid q{+}q \mid q\theta t \mid t \leftarrow q \mid$ *constant* $\mid$ *variable*. However constants and variables can be eliminated from the computation because relations are of interest.

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

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

$$arity(v_1\ \theta\ v_2, t =< \cdots, v_1, \cdots, v_2, \cdots >) = \begin{cases} 0, & \text{if } v_1\theta v_2 \text{ is false} \\ 1, & \text{if } v_1\theta v_2 \text{ is true} \end{cases}$$

$$arity(v \leftarrow q, t) = \begin{cases} 0, \text{ if } \forall\vec{x} : \mathsf{dom}_{\vec{x}}(q) \text{ is empty} \\ 1, \text{ otherwise} \end{cases}$$

3 Graph representation of maps

For computing the domains of input variables of maps we can use graph modeling. In such a way that, given the expression we can compute the domain of each expression with this modeling in $O(m \cdot p)$ where m is the number of occurrences of all input variables and p is the cost of evaluating the domains according to the mentioned rules.

In this section firstly we talk about the the graph modeling(dependency graph), then we will give an algorithm for computing this dependency graph from the parse tree of expressions. From the parse tree of an expression we can construct a directed graph $G(V, E)$. For each map $m[.][.]$ there is a vertex in V and edges represent the variable. We have an edge between m_1 and m_2 with label a, if and only if a is in the input variables of m_2 and output variables of m_1 and there is an information flow from m_1 to m_2.

Obviously this graph does not have any cycle. To prove it, suppose it has a cycle with at least two vertices, call the first two vertices as m_1, m_2. In the cycle there is an edge between m_1 and m_2. Since this is a cycle then there is a path between m_2 and m_1 also. But it is not possible because having an edge between m_1 and m_2 means that m_2 occurs after m_1 and according to information flow rule we can not have a path again between m_2 and m_1.

In this graph vertices without any incoming edge are always the relations. If we evaluate the nodes with a topological traversal we can guarantee that the input variables of each maps is computed in order. The cost of domain computations is $O(m \cdot p)$, where m is the number of edges in the graph and p is the cost of using any of mentioned rules for computing the domain of subexpressions as we had in the aforementioned way.

We can construct the graph from the given parse tree in $O(m)$ where m is the number of edges. We start from the root of the parse tree and recursively compute the subgraph of the left child and right child, then we add the edges between the maps with some input variables on the right child to the appropriate vertices on the left child's graph. Since we consider each edge just one time in the algorithm, the overall order is $O(m)$.

4 Domain computation

We are trying to compute the domains for each variable that appears in a query. This computation will be 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 algorithm will need as inputs the root of the tree and a structure that must be previously defined and will return a structure that will contain the vector of variables and the domains for each variable defined in the vector. The structure will look like:

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

Algorithm 1 Computing the domains

Input: the root of the tree **node**, a structure **s** that will be null
Output: a structure **s1** that will contain the vector of variables $\vec{x}$ and the domains of each variable $\textbf{dom}_{\vec{x}}(query)$

```
1:  if node has children then
2:      if + is between node.Left and node.Right then
3:          struct $s1 \leftarrow$ computeDomain(node.Left,struct s)
4:          struct $s2 \leftarrow$ computeDomain(node.Right,sturct s)
5:          struct $s3.\vec{x} \leftarrow s1.\vec{x} \cup s2.\vec{x}$ {this will compute the vector for all the
            variables, both from the right and left node}
6:          struct $s3.dom \leftarrow \mathsf{dom}_{s3.\vec{x}}(node.Left + node.Right)$
7:          return s3
8:      else
9:          struct $s1 \leftarrow$ computeDomain(node.Left,struct s)
10:         struct $s2 \leftarrow$ computeDomain(node.Right,struct s1)
11:         return $s2$
12:     end if
13: else
14:     if struct s=null then
15:         $s1.\vec{x} \leftarrow$ all the variables of the leaf
16:         $s1.dom \leftarrow \mathsf{dom}_{s1.\vec{x}}$(relation from the leaf)
17:     else
18:         $s1.\vec{x} \leftarrow s.\vec{x} \cup$ all the variables of the leaf
19:         $s1.dom$ will be the new domain, computing the domain of the leaf
            relation, but also taking into account the structure $s$
20:         return $s1$
21:     end if
22:     return $s1$
23: end if
```

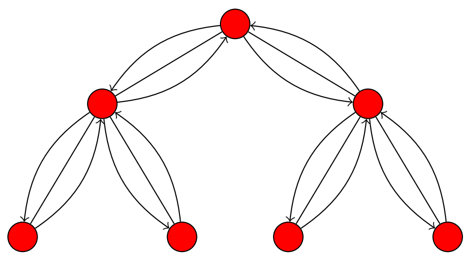

Figure 1: Tree traversal