Acknowledgements: This work is supported in part by the National Science Foundation under grant IIS-9811525.

References

[1] R. Agrawal, N. Gehani, and J. Srinivasan. OdeView: The Graphical Interface to Ode. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ*, pp 34–43, May 1990.

[2] N. Balkir, E. Sukan, G. Ozsoyoglu, and Z. Ozsoyoglu. VISUAL: A Graphical Icon-Based Query Language. In *Proceedings of the Twelfth International Conference on Data Engineering, 1996, New Orleans, Louisiana*, pp 524–532, February 1996.

[3] M. J. Carey, L. M. Haas, V. Maganty, and J. H. Williams. PESTO : An Integrated Query/Browser for Object Databases. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pp 203–214. Morgan Kaufmann, 1996.

[4] M. Chavda and P. T. Wood. Towards an ODMG-Compliant Visual Object Query Language. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pp 456–465. Morgan Kaufmann, 1997.

[5] I. F. Cruz. DOODLE: A Visual Language for Object-Oriented Databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2-5, 1992*, pp 71–80, 1992.

[6] I. F. Cruz, M. Averbuch, W. T. Lucas, M. Radzyminski, and K. Zhang. Delaunay: A Database Visualization System. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pp 510–513, 1997.

[7] L. Fegaras. *λ-DB: An Object-Oriented Database Management System.* User manuals and source code are available at `http://lambda.uta.edu/lambda-DB/manual/`.

[8] L. Fegaras and D. Maier. Towards an Effective Calculus for Object Query Languages. *ACM SIGMOD International Conference on Management of Data, San Jose, California*, pp 47–58, May 1995.

[9] S. Kaushik and E. A. Rundensteiner. SVIQUEL: A Spatial Visual Query and Exploration Language. In *Conference and Workshop on Database and Expert Systems Applications (DEXA'98), Vienna, Austria*, pp 290–299, August 1998.

[10] D. Papadias and T. K. Sellis. A Pictorial Query-by-Example Language. *Journal of Visual Languages and Computing, Special Issue on Visual Query Systems*, 6:53–72, 1995.

[11] R. Cattell *et al. The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997.

[12] K. V. Vadaparty, Y. A. Aslandogan, and G. Özsoyoglu. Towards a Unified Visual Database Access. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pp 357–366. ACM Press, 1993.

4 Limitations

Even though our framework can capture most OQL queries, there are some serious limitations that we hope to address in a future research. The most important limitations are listed below:

1. record construction in our framework is very limited, namely, there is no choice on attribute names and record attributes are constructed only in the order listed in the templates;
2. our group-by construct allows *having* conditions (i.e., conditions after group-by) for non grouped-by values, and regular conditions (before group-by) for grouped-by values only;
3. the OQL code in the head and cond part of a template is not type-checked by our system (this can be fixed by incorporating an OQL compiler into our system);
4. there is no support for constructing constant collections, disjunctive predicates, unions, and array operations (e.g. the nth element of a list);
5. finally, there is no visual way of calling methods and predefined operations (such as element).

In addition, our visual queries can only capture OQL queries whose *from* clauses have domains that are path expressions, not complex queries. We do not consider this as a serious limitation because earlier work [8] has shown that most queries with this type of nesting can be normalized into queries over path expressions.

5 Related Work

Our work is influenced by the PESTO project [3]. PESTO provides an integrated user-friendly interface for both query formulation and data browsing. Even though the PESTO query language is less expressive than ours, since it does not support query nesting, integrating data browsing with query formulation is highly desirable because data can provide a very useful feedback to the query formulation process itself. Another project with similar goals as ours is the QUIVER project [4], which uses data-flow diagrams to represent functions and query blocks. Even though the QUIVER visual query language supports query nesting, it does not support group-by, order-by, and universal quantification. The VQL query language [12] uses a restricted universal quantifier to capture many advanced OODB query features but is based on Datalog and unification, which requires the explicit use of domain variables to relate data in different parts of the visual query. We believe that a query language based on the functional model, such as VOODOO, is more appropriate and intuitive than one based on the logic model when the intention is to capture a functional-style query language, such as OQL. The same can be said for the DOODLE visual query language [5, 6], which is based on F-logic and uses variables to perform joins and existential quantifications. One novel idea used in DOODLE, which we would like to explore in a future research, is that its visualization constructs are user-defined instead of hard-wired. This allows a better visualization of queries in specialized application domains. There is also related work on query formulation under spatial constraints [2, 10, 9]. The focus of these languages is to express spatial queries by positioning spatial objects on a plane in a way that reflects the spatial relationship of these objects in the database.

6 Conclusion

We have presented a simple and effective visual language to express ODMG OQL queries. The language is expressive enough to allow most types of query nesting, aggregation, universal and existential quantifications, group-by, and sorting. It is uniform, which makes it easy to learn. Most queries can be constructed visually using few clicks on some buttons and by typing the constant values of conditions. Our language is strongly typed in the sense that queries constructed in our system are always type correct. In addition, there is sufficient type information displayed by the system that guides every stage of the query construction. The implementation of our system required less than 1000 lines of Tcl/Tk source code. Currently, our system is self-contained and generates OQL text only but there are plans to connect it to the λ-DB OODB [7] developed at the University of Texas at Arlington.

where X denotes a new variable name (that has not appeared before). The make_query function is recursive: it generates one query for each template accumulator:

make_query(path, p, $[a_1, a_2, \ldots, a_n]$, env) =
let $(a, hs, is, ps) \leftarrow$ make_query(path, X, $[a_2, \ldots, a_n]$, $env[p/X]$)
in if a_1 = "elem" **then** $(a_1, hs, is \cup \{X \textbf{ in } p\}, ps)$
else $(a_1$, make_oql(a, hs, is, ps), $\{X \textbf{ in } p\}$, $\{\})$

where X denotes a new variable name and $env[p/X]$ extends the environment, env, with the binding from X to p. This function considers all the accumulators in a template in sequence and for each accumulator it generates a nested query whose *from* clause is X **in** p. The only exception to the rule is for the elem accumulator. This accumulator requires the nested query to be flatten, so it simply appends the X **in** p *from* clause to the query associated with the previous accumulator. The only component of the make_query function that remains to be expressed is the one that handles templates with no accumulators:

```
make_query(path,p,[],env) =
{   if not template(path) then return ("", {p}, { }, { })
    heads ← { }; iterations ← { }; predicates ← { }
    foreach A ∈ scope(path) {
        v ← compile_path(path.A, env)
        (a, hs, is, ps) ← make_query(path.A, v, accumulators(path.A), env)
        if print(path.A)
        then if a = "elem"
            then {
                heads ← heads ∪ hs
                iterations ← iterations ∪ is
                predicates ← predicates ∪ ps
            } else heads ← heads ∪ { A : make_oql(a, hs, is, ps) }
        else if template(path.A) and dtype(path.A)="bool"
            then if a = "elem"
            then {
                iterations ← iterations ∪ is
                predicates ← predicates ∪ ps ∪ hs
            } else predicates ← predicates ∪ { make_oql(a, hs, is, ps) }
        foreach path': join(path,path') ≠ "" {
            predicates ← predicates ∪ { v "join(path,path')" compile_path(path',env) }
        }
    }
    if head(path) ≠ "" then heads ← heads ∪ { head: head(path) }
    if predicate(path) ≠ "" then predicates ← predicates ∪ { predicate(path) }
    return ("",heads,iterations,predicates)
}
```

where compile_path is defined as follows:

compile_path(path,env) =
if path = path1.path2 **and** (X,path1)$\in$ env **then** compile_path(X.path2,env) **else** path

that is, it uses the environment, env, to replace some path variables. The make_query algorithm considers every attribute A in the scope of the template and constructs the components of the tuple (acc,heads,iterations,predicates) by calling the make_query function recursively and by merging the results. For example, if the P-button of A is pushed (i.e. print(path.A) is true) then the result of calling the make_query function recursively for A is transformed to an OQL query and is appended to the head list. Here too there is special code to handle templates with an "elem" accumulator.

The original type of a path, otype(path), is derived directly from the schema:

$$\begin{array}{lcl}
\text{otype(root)} & = & \text{a struct that contains all class extents} \\
\text{otype(path.}A\text{)} & = & \textbf{let } \text{path'} \leftarrow \text{scope(path)}[A] \\
 & & \textbf{in if } \text{otype(path'.}A\text{)} = T_1\langle\cdots\langle T_n\langle\text{struct}(\ldots, A : T, \ldots)\rangle\rangle\rangle,\ n \geq 0 \textbf{ then } T \\
 & & \quad \textbf{else if } \text{otype(path'.}A\text{)} = T_1\langle\cdots\langle T_n\langle\text{class_name}\rangle\rangle\rangle,\ n \geq 0 \\
 & & \quad \textbf{then } \text{the type of attribute } A \text{ in class class_name}
\end{array}$$

where each T_i is a collection type constructor, such as a set. The gtype of a path is equal to otype, except when that template associated with this path is in a group-by mode and its G button has not been pushed; in the latter case it becomes a set:

$$\begin{array}{lcl}
\text{gtype(root)} & = & \text{otype(root)} \\
\text{gtype(path)} & = & \textbf{let } \text{path'} \leftarrow \text{scope(path)}[A] \\
 & & \textbf{in if } \exists B \in \text{scope(path')} :\ \text{group(path'.}B\text{)} \\
 & & \quad \textbf{then if } \text{group(path'.}A\text{)} \textbf{ then } \text{otype(path')} \textbf{ else } \text{set}\langle\text{otype(path')}\rangle \\
 & & \quad \textbf{else } \text{otype(path')}
\end{array}$$

The derived type of a path depends on which attributes are P-checked as well as on the template accumulators:

$$\begin{array}{lcl}
\text{dtype(path)} & = & \textbf{if not } \text{template(path)} \textbf{ then } \text{gtype(path)} \\
 & & \textbf{else let } L \leftarrow \{\ A : \text{dtype(path.}A\text{)} \mid A \in \text{scope(path)},\ \text{print(path.}A\text{)}\ \}, \\
 & & \qquad S \leftarrow \textbf{if } \text{head(path)} \neq \text{“”} \textbf{ then } L \cup \{\ \text{head} : \text{type_of(head(path))}\ \} \textbf{ else } L, \\
 & & \qquad tp \leftarrow \textbf{if } S = \emptyset \textbf{ then if } \text{condition(path)} = \text{“”} \textbf{ then } \text{gtype(path)} \textbf{ else bool} \\
 & & \qquad\quad \textbf{else if } S = \{X : T\} \textbf{ then } T \textbf{ else make a struct with } S \text{ components} \\
 & & \quad \textbf{in } f(\text{accumulators(path)}, tp)
\end{array}$$

where function f considers the template accumulators in sequence to derive the mapped type:

$$\begin{array}{lcl}
f([\,], tp) & = & tp \\
f([a_1, \ldots, a_n], tp) & = & \textbf{if } a_1 \in \{\text{“}sum\text{”}, \text{“}max\text{”}, \text{“}min\text{”}\} \textbf{ then if } f([a_2, \ldots, a_n], tp) = \text{int} \textbf{ then int else error} \\
 & & \textbf{else if } a_1 \in \{\text{“}and\text{”}, \text{“}or\text{”}\} \textbf{ then if } f([a_2, \ldots, a_n], tp) = \text{bool} \textbf{ then bool else error} \\
 & & \textbf{else if } a_1 \in \{\text{“}elem\text{”}\} \textbf{ then } f([a_2, \ldots, a_n], tp) \\
 & & \textbf{else } a_1\langle f([a_2, \ldots, a_n], tp)\rangle
\end{array}$$

We are now ready to present the algorithm that translates visual queries into OQL. To simplify the presentation of the algorithm, we will not consider group-by and sorting. The algorithm consists of two components:

- make_query(path,p,accumulators,env) returns a tuple (acc,heads,iterations,predicates) that specifies an OQL query for the template associated with path, where acc is an accumulator, head is a list of expressions that appear in the *select* clause of an OQL query (the projection part), iterations is a list of strings of the form "var **in** domain" that appear in the *from* clause of an OQL query, and predicates is a list of boolean predicates. Path expression p designates the place to retrieve the template values, the list accumulators contains the template accumulators, and env is an environment (a binding list) that contains the bindings from X to p for every X **in** p in the *from* clause of the query.

- make_oql(acc,heads,iterations,predicates) generates an OQL query from its parameters (which are returned by the make_query function and are described above).

The definition of the make_oql function is straightforward:

```
make_oql(acc,heads,iterations,predicates) =
let q ← "select heads from iterations where predicates"
in if acc="or" then "exists X in q: X"
   else if acc="and" then "for all X in q: X"
   else if acc ∈ {"sum","max","min"} then "acc(q)"
   else if acc="set" then "select distinct heads from iterations where predicates" else q
```

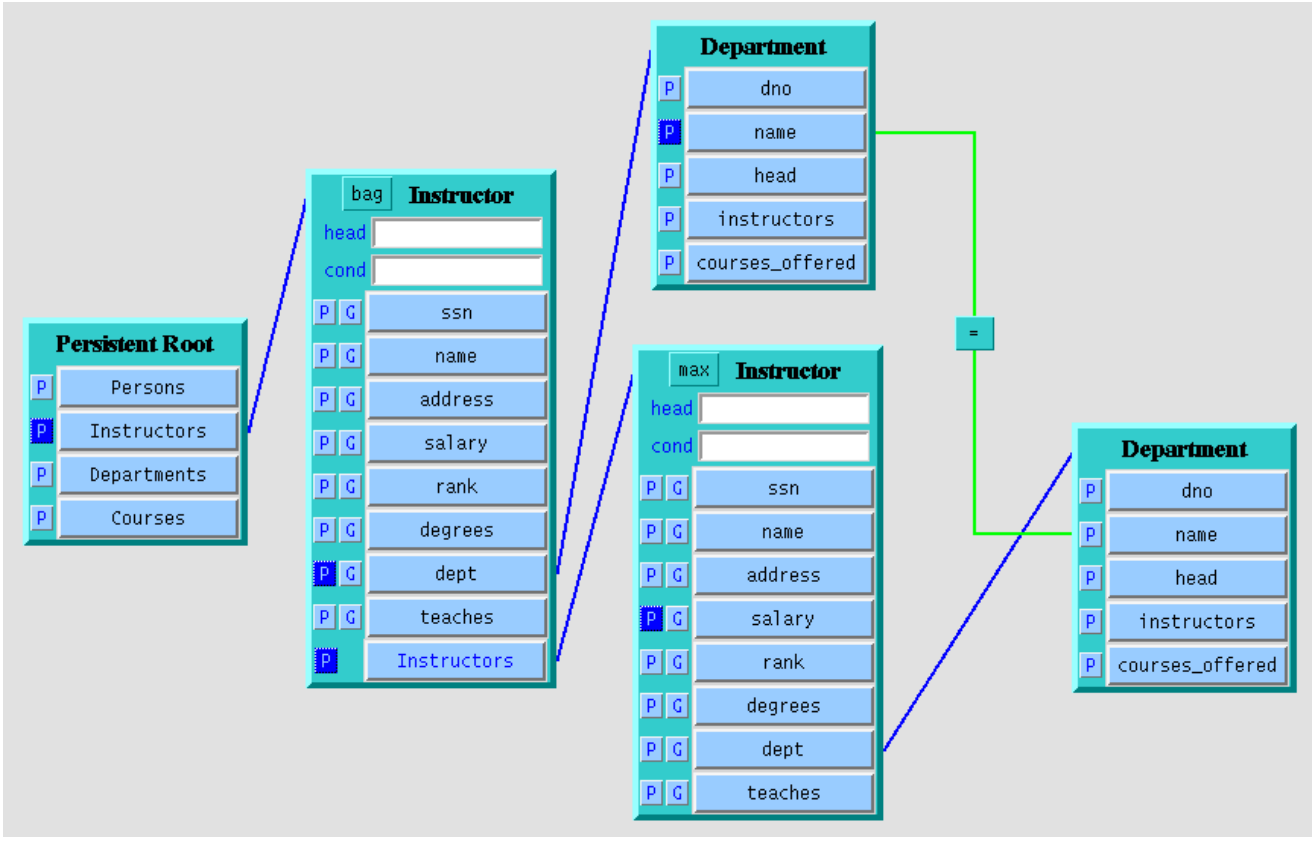

Figure 8: For each instructor print the instructor's department name and the maximum salary of all instructors working in the same department

We will use the following data structures to represent the information displayed in a query diagram:

template(path)	boolean that indicates whether the path template is displayed on the canvas
otype(path)	the original type of the attribute
gtype(path)	the original type of the attribute after group-by
dtype(path)	the derived type of the attribute
scope(path)	the template environment: a list of name-to-path bindings
print(path)	boolean that indicates whether the P button is pushed
group(path)	boolean that indicates whether the G button is pushed (true, if G does not exist)
sort(path)	boolean that indicates whether the S button is pushed
head(path)	the expression written in the head of the template
condition(path)	the boolean expression written in the cond part of the template
accumulators(path)	a list of strings from "set", "bag", "max" etc, one string for each template accumulator
join(path1,path2)	the string "" (no join), or "=", "<=" etc to indicate a join between the two paths

The scope(path) contains bindings for all attributes that appear inside the template for path, including those copied from ancestor templates. The binding of an attribute is the path of the template that owns this attribute before it was copied. For example, scope(root.Instructors) in Figure 8 contains a binding from the Instructors attribute name to the path root. In addition, the join(path1,path2) was stored in such a way that path2 satisfies the previously mentioned restriction for joins: the path from path2 to the closest common ancestor of path1 and path2 must not contain any template with an accumulator. For example, in Figure 8 we have join(root.Instructors.Instructors.dept.name,root.Instructors.dept.name)="=", since the closest common ancestor is the leftmost Instructor template.

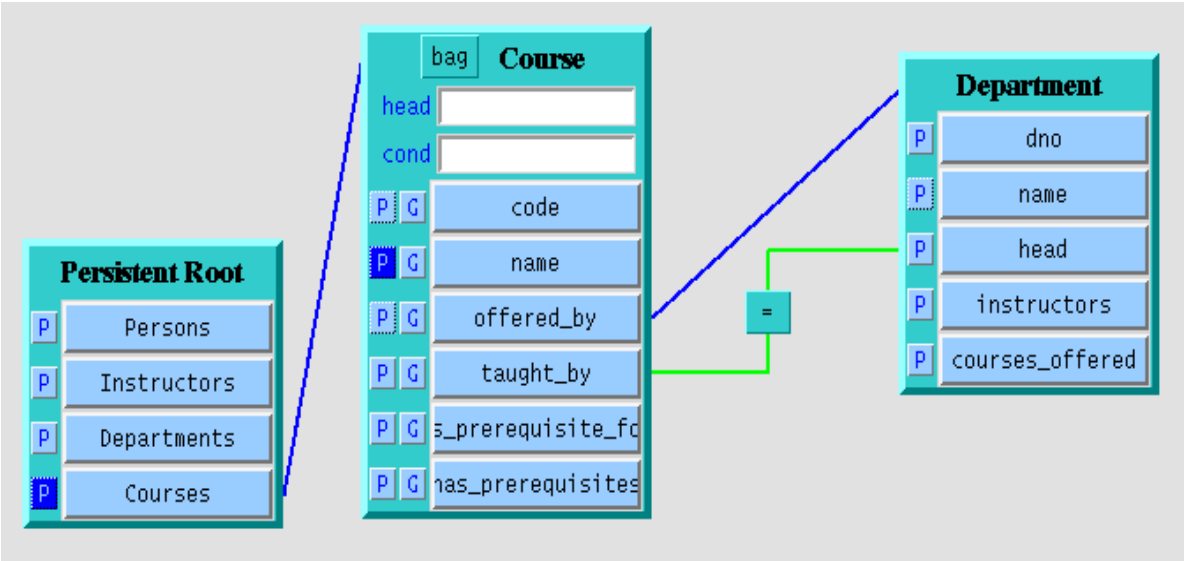

Figure 7: Find all courses taught by the heads of the offering departments

The last query displayed in Figure 8, is an alternative way of performing a group-by using nested queries. It represents the OQL query:

```
select dept: e.dept.name,
       Instructors: max(select s.salary
                        from s in Instructors
                        where e.name = s.name)
from e in Instructors
```

which is equivalent to:

```
select dept: e.dept.name, Instructors: max(e.salary)
from e in Instructors
group by e.name
```

The new thing here is that we copied the Instructors attribute from the Persistent Root template into the Instructor template. This was done by clicking the middle mouse button on the Instructors button of Persistent Root and dragging it and release it on top of the Instructor template. We can copy any attribute in a template into any template on the canvas as long as the destination template is a descendent (or it is the template itself) of the source template. This reflects the variable scoping of OQL where variable names in outer queries are visible in inner queries but are not visible in sibling queries. Note that when an attribute is copied, its template is not copied. Note also that if we click the middle mouse over an attribute and release it immediately, this attribute is copied into the same template. That way we can map an attribute in different ways by copying it and by attaching different templates to it. We can copy the same attribute multiple times; the system renames each duplicate attribute name, say ssn, by appending a new number, e.g. ssn_1.

3 Translating Visual Queries into OQL

In this section, we present an algorithm that translates visual queries into OQL queries. The data structures used by this algorithm are expressed in terms of paths. A path is associated with both a template button and the template attached to this button (if the template exists). The path of the root template is *root*. If there is an attribute with name A in a template with path *path*, then the path of the button for A as well as the path of the template attached to this button is *path.A*. For example, the path of the Department template in Figure 8 is *root.Instructors.Instructors.dept*.

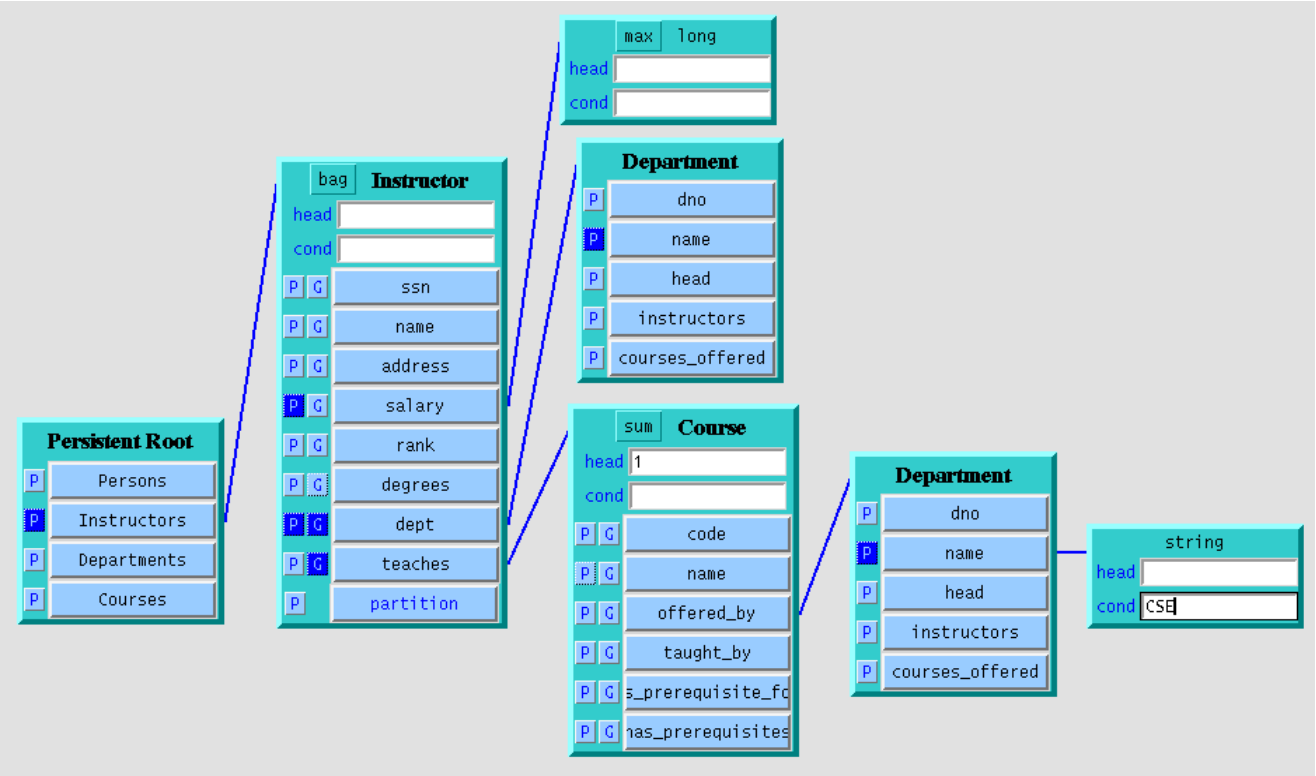

Figure 6: Group all instructors by their department name and by the number of CSE courses they teach. For each group, return the department name and the maximum salary of its instructors

The G buttons appear only on templates that correspond to collection original types (i.e. templates that have at least one accumulator). They indicate the attributes to group by. When at least one G button is pushed, then the template goes to a group-by mode. In that mode, the original attribute type of each button that has not been G-checked becomes a set of the original type. The original types of the G-checked buttons remain the same. In our query, the original type of dept remains Department, for teaches remains set⟨Course⟩, while for ssn changes to set⟨long⟩, for degrees changes to set⟨set⟨string⟩⟩, etc. Each non G-checked button corresponds to a group of values of the corresponding attribute constructed when we group by the G-checked attributes. There is also a new attribute added to the template, called *partition*, of type set⟨Instructor⟩, that contains the groups. It corresponds to the partition variable of the OQL group-by construct and is used for more complex aggregations after the group by.

Arbitrary join predicates between template attributes can be constructed using join lines on the canvas. The query in Figure 7, for example, finds all courses taught by the heads of the offering departments, and has the following OQL form:

```
select c.name
  from c in Courses
 where c.taught_by = c.offered_by.head
```

The join line in Figure 7 is constructed by clicking one of the participating buttons (Course.taught_by or Department.head) and dragging it and release it on top of the other button. Joins are permitted if at least one path to the closest common ancestor from the two participating attributes does not include a template with an accumulator (this restriction is justified in the formal description of the framework, given in Section 3). The join line is red in the beginning and becomes green only when the participating attributes have the same derived type and satisfy the restriction mentioned above. Our system does not permit invalid joins. The equal sign in the middle of the join line is a menu that allows the selection of a comparison relation.

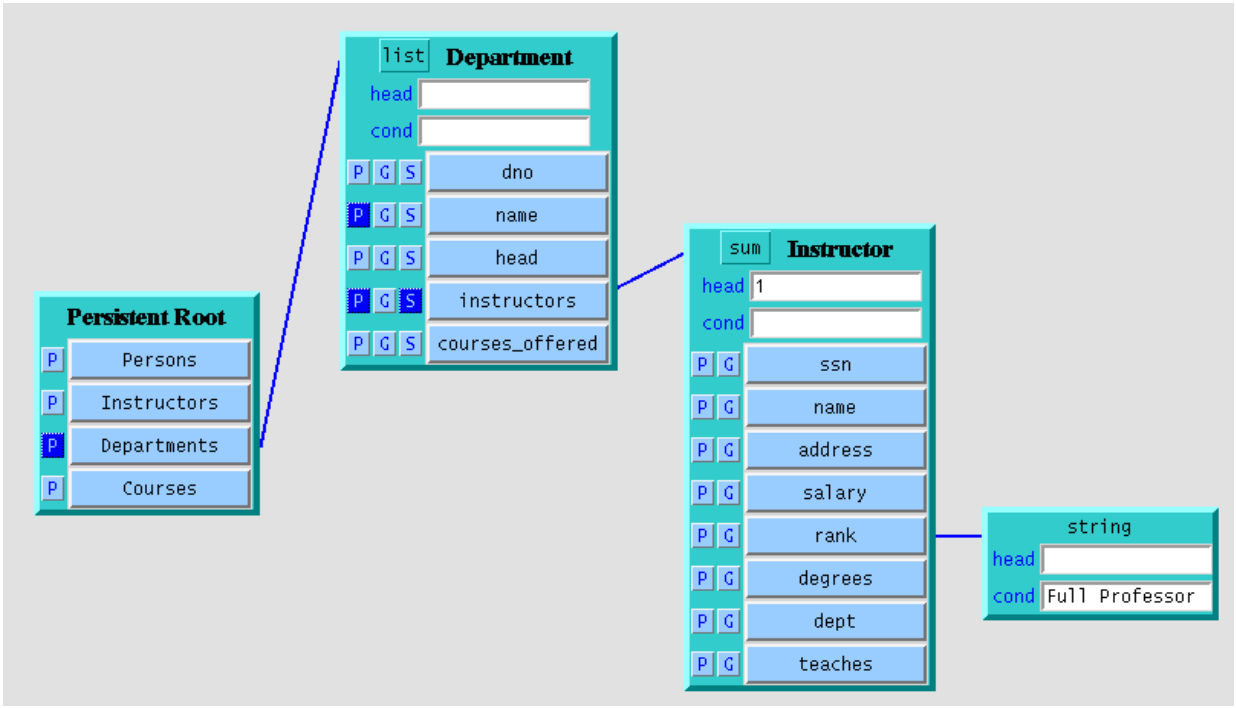

Figure 5: Print all departments sorted by the number of full professors

If we select "list" as an accumulator, then a new button S is displayed next to each attribute in the template (which disappears when the accumulator changes to something else). If an S button is pushed, then the corresponding attribute is chosen for sorting. For example, the query in Figure 5 sorts departments by the number of full professors. It corresponds to the OQL query:

```
select name: d.name,
       instructors: count(select * from e in d.instructors
                          where e.rank = "Full Professor")
 from d in Departments
order by count(select * from e in d.instructors
               where e.rank = "Full Professor")
```

Here each Instructor is mapped into the integer one. Thus, the Department instructors attribute is mapped to the number of the instructors in the Department since all these ones are added together by the accumulator "sum".

The group-by construct is syntactic sugar in OQL; it can be easily (and better) expressed using nested queries. The same is true for our query language. Even though the group-by construct is non-functional and often confusing, we decided to supported it directly in our framework in order to be compliant with legacy query languages. We will present later an alternative way of performing a group-by. For example, the query in Figure 6 groups all instructors by their department name and by the number of CSE courses they teach. It corresponds to the OQL query:

```
select salary: max(e.salary), dept: e.dept.name
 from e in Instructors
group by e.dept.name,
         count(select *
               from c in e.teaches
               where c.offered_by.name = "CSE")
```

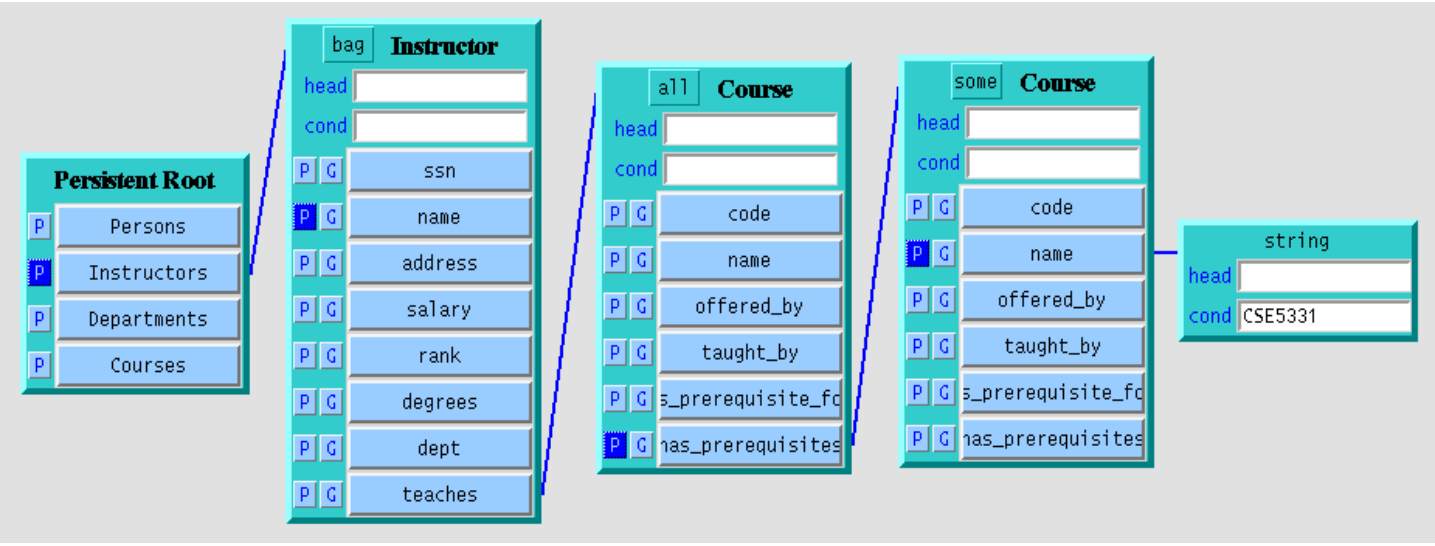

Figure 4: Print the name of an instructor if *every* course this instructor teaches has CSE5331 as a prerequisite

A more challenging example of a query construction is the query in Figure 3. It finds the names and addresses of all instructors in the CSE department who earn more than 100K. This query can be expressed in OQL as follows:

```
select name: e.name, address: e.address
from d in Departments,
     e in d.instructors
where d.name = "CSE"
and e.salary > 100000
```

The new thing in this query is the accumulator "elem" in the Instructor template, which stands for a single collection element. It basically captures the nested relational algebra operator, unnest. For each Department d with a set of instructors $\{e_1, e_2, \ldots, e_n\}$, it associates d with each instructor e_i. The derived type of the Instructor template is struct(name: string, address: struct(street: string, zipcode: string)), which is also the derived type of the instructors attribute in Department. Thus, the result type of the query is:

bag⟨struct(name: string, address: struct(street: string, zipcode: string))⟩

There are many other types of accumulators. For example, the query in Figure 4 uses the accumulators "all" and "some" to express universal and existential quantifications. This query can be expressed in OQL as follows:

```
select e.name
from e in Instructors
where for all c in e.teaches:
      exists d in c.has_prerequisites: d.name = "CSE5331"
```

The selected attribute of the rightmost Course template has a boolean derived type, and this template uses "some" to accumulate the boolean values. When function "some" is applied on a set of booleans, it returns true if at least one boolean is true. That is, if there is at least one course with name CSE5331 then the mapped value of this template will be true. Similarly, the only attribute in the left Course template has a boolean derived type and the "all" accumulator indicates that if all these booleans are true then the mapped value of this template will be true too. Other kinds of accumulators include aggregations, such as summation, product, maximum, minimum, and average, and lists specified by order-by (described next). For example, if we change the accumulator of Instructor to "sum" (summation) in Figure 4 and we P-check salary instead of name, then the mapped value of this template will become an integer, namely the sum of the salaries of all instructors if all the courses these instructors teach have CSE5331 as a prerequisite.

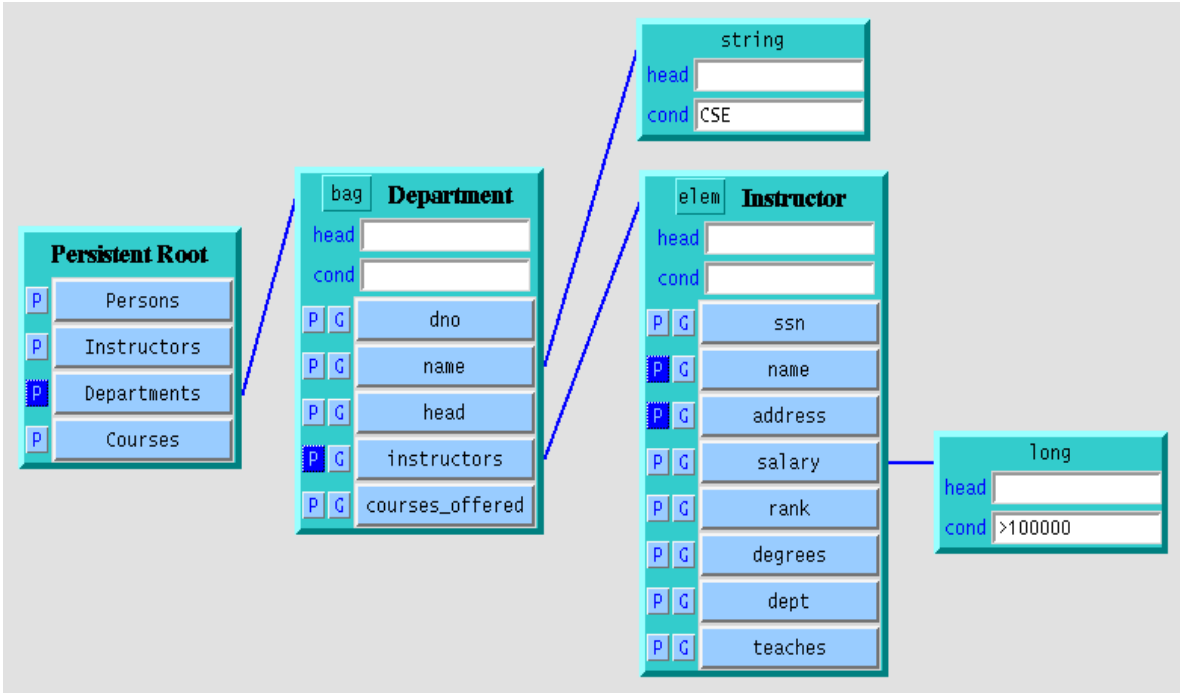

Figure 3: Find the names and addresses of all instructors in the CSE department who earn more than 100K

The cond entry window in a template indicates a condition to be satisfied by the mapped value of this template. The syntax of the condition is similar to that of the Microsoft Access Query-by-Example interface. For example, if the mapped value is an integer, then one valid condition is: >100 and <200, which indicates that the integer value must be between 100 and 200. In our example, the string template indicates that the string must be equal to "Smith". If no attribute is selected, and the head is empty, but the cond is not empty, then the template is mapped into a boolean, which is the result of the condition. If the condition is empty too, the derived type is equal to the original type. For example, the string template is mapped into a boolean, which indicates that the Instructor name has now a boolean attribute type. Since the Instructor name is P-checked, the Instructor template is mapped into a boolean too. If the derived type of an attribute is a boolean, then this boolean is used as a condition to be satisfied by the template. This means that, since the Department head attribute is of type boolean, it is used as a condition, namely in d.head.name="Smith" for every department d.

To summarize, it took us five button clicks and the typing of the word "Smith" to construct our first query. In fact most OQL queries can be constructed using a small number of button clicks and the unavoidable typing of some constant values.

Type information is very important when constructing queries in our framework. To simplify this process, our interface displays all the necessary type information and, more importantly, it restricts any type inconsistent query construction. For example, if the mouse cursor is positioned on top of a button, a small label is displayed on the canvas that contains the derived type of the attribute. A similar label is displayed when the mouse cursor is positioned on top of the template header, which contains the derived type of the button associated with the template. Guiding labels with usage help are also displayed when the mouse cursor is on top of labels and buttons. There is also support for tree editing functionality that allows us to display queries nicely. For example, if we click and drag the template from its header using the left mouse button, we can move the template into a new position on the canvas, dragging its connected lines along with it. If we do the same with the middle mouse button, we drag the template along with all its descendent templates. If we click a template header using the right mouse button, then this template, all its descendent templates, and all their connecting lines are removed from the canvas.

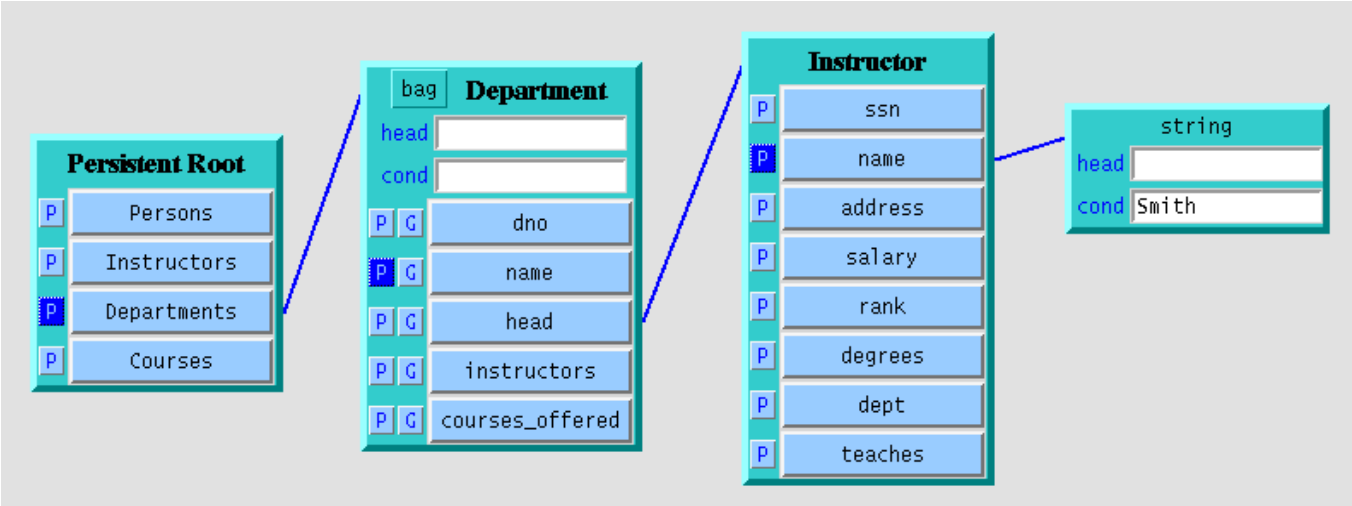

Figure 2: Find the name of the department whose head is Smith

the persistent root template contains four attributes: Persons, Instructors, Departments, and Courses, which represent the class extents of our University database.

Initially, when its button has not been pushed, the type of an attribute is the type defined in the database schema. This is called the *original attribute type* of the button, to be distinguished from the *derived attribute type*, which depends on the types of the children templates of this button, as we will see next. For example, the original type of the Department attribute in the Persistent Root template is set⟨Department⟩. When a button is pushed, it creates a template that reflects the structure of its original attribute type. For example, if we push the Departments button we create the Department template, which is displayed next to the Persistent Root template and is connected to the Departments button by a line. This new template contains the attributes of the class Department. When the head button in the Department template is pushed, an Instructor template is created (since the original type of the head is Instructor). Finally, when the name button in the Instructor template is pushed, another template for the string type is constructed (since the original type of the Instructor name is string). This template is similar to the other templates but has no attributes.

A template, which is generated by pushing a button, specifies a function that maps the original attribute type of the button into some domain. The derived attribute type of the button is equal to this domain type, which may be different than the original attribute type. The buttons and menus inside a template are used in specifying this mapped value of the template. The P check-button at the left of an attribute button indicates whether this attribute should be considered in the mapped value (when P is on) or not (when P is off). (The G button is for grouping-by and will be explained later.) The entry window below the template header labeled *head* indicates an explicit value to consider in the mapped value. It can be a constant value or even a complex expression (such as a method call) that may use the values of the template attributes. The mapped value of the template is an OQL struct that contains all the P-checked attributes in the template plus the head, if the head is not empty. If this struct contains one element only, the struct is dropped and the mapped value becomes the element itself.

At the top left side of the Department template there is a menu, called an *accumulator*, which indicates how to accumulate the selected attributes to form the mapped value. For each collection type in the original attribute type, there is one accumulator next to the template name that indicates how to handle this collection. The Instructor and string templates in our example do not have accumulators because their original types were not collections. The Department template though has one accumulator since the original type was set⟨Department⟩. The Department accumulator indicates that we form a bag out of the selected values. Thus the mapped value of the Department template has the type bag⟨string⟩, which also indicates that the derived attribute type of the button Departments in Persistent Root is also bag⟨string⟩, which in turn is the type of the query result.

```
class Person ( extent Persons key ssn )
{   attribute long ssn;
    attribute string name;
    attribute struct( street:string, zipcode: string ) address;
};
class Instructor extends Person ( extent Instructors )
{   attribute long salary;
    attribute string rank;
    attribute set⟨string⟩ degrees;
    relationship Department dept inverse Department::instructors;
    relationship set⟨Course⟩ teaches inverse Course::taught_by;
};
class Department ( extent Departments keys dno, name )
{   attribute long dno;
    attribute string name;
    attribute Instructor head;
    relationship set⟨Instructor⟩ instructors inverse Instructor::dept;
    relationship set⟨Course⟩ courses_offered inverse Course::offered_by;
};
class Course ( extent Courses keys code, name )
{   attribute string code;
    attribute string name;
    relationship Department offered_by inverse Department::courses_offered;
    relationship Instructor taught_by inverse Instructor::teaches;
    relationship set⟨Course⟩ is_prerequisite_for inverse Course::has_prerequisites;
    relationship set⟨Course⟩ has_prerequisites inverse Course::is_prerequisite_for;
};
```

Figure 1: The ODL Schema of the University Database

queries in terms of comprehensions but, instead, to translate them directly into OQL, because in an earlier work [8] we have provided a formal semantics of OQL in terms of comprehensions. Finally, in Section 4 we list the limitations of our language and in Section 5 we compare our work with existing query formulation tools.

2 Examples of Queries

In this section we present eight queries to show the basic features of our visual query language. We use the ODL schema in Figure 1 for our examples, which describes a University database.

Our query formulation interface consists of a drawing canvas that contains a number of rectangles, called templates, connected by lines. For example, the query in Figure 2 finds the name of the department whose head is Smith. This query can be expressed in OQL as follows:

```
select name: d.name
from d in Departments
where d.head.name = "Smith"
```

Templates are organized in a tree form. The root of the tree is the persistent root, which is an aggregation (an ODL struct) of all persistent objects in the database, including all the class extents. When VOODOO starts, the only template displayed on the canvas is the persistent root template. The large buttons inside some templates represent template attributes. They create new templates when they are pushed. For example,

A Strongly Typed Visual Query Language for OODBs

Leonidas Fegaras

Department of Computer Science and Engineering
The University of Texas at Arlington
416 Yates Street, P.O. Box 19015
Arlington, TX 76019-19015
email: *fegaras@cse.uta.edu*

1 Introduction

Query and data visualization are key components of many commercial relational database systems, such as Microsoft Access and Paradox. Object-oriented databases offer excellent opportunities for visual data browsing because they provide a direct and natural mapping of real-world objects and relationships into equivalent constructs. Even though there is already a sizable number of graphical user interfaces for object browsing [1], which include commercial products, such as O2Look and O2Tools of O2, there is little work on visual query formulation in OODBs. Some researchers believe that, since OODB queries can be arbitrarily complex, this complexity must be reflected in the query formulation tool itself. We believe, and we hope to show in this paper, that this is not necessarily true. If a query language is well-designed and uniform, as it is the case for ODMG OQL [11], it would require very few visual constructs for query formulation, since it would map similar features into similar visual constructs.

This paper presents a simple and effective visual query formulation tool for OQL, called VOODOO (which stands for a Visual Object-Oriented Database language for Odmg Oql). Our design goal was to develop a visual language that can capture most OQL queries and at the same time to be uniform and very simple to learn and use. Since query nesting is an important feature of any OODB query language, we designed our language in such a way that the composition of complex queries from simple ones is natural and intuitive.

A VOODOO query has a tree form that reflects the structure of the database schema, where each class or type reference in the schema is expanded, potentially leading to an infinite tree. The root of the tree is the persistent root and each tree node represents a class or a structure in the database schema. The tree nodes, which are called *templates*, are expanded on demand, so that only the minimal information is displayed. A query is formulated by mapping templates into values in some domain using a number of buttons and menus inside the template layout. The value of a template depends on the values of its children templates in the query tree as well as on the values of its buttons and menus. The structure of the database, which is reflected in the query tree, determines the methods for accessing the database to retrieve data (i.e. it specifies which data paths to follow and which collections to traverse) and it is specified from the root to the tree leaves. The actual data manipulation and the formulation of the query result is specified from the tree leaves to the root by mapping each template to some value, which in turn is passed to the parent of the template to form the parent's value. The template mappings as well as the composition of the template values are formed in a pure functional way. Since new value domains are formed in each template, type information about these mapped values is very important to the user. There is sufficient type information displayed by the system that guides every stage of the query construction. In addition, VOODOO uses type information to reject invalid queries, so that queries constructed in our system are always type correct.

The remainder of this paper describes the basic features of VOODOO using a number of visual queries (Section 2). Section 3 presents a type inference algorithm to typecheck visual queries and a formal framework for compiling visual queries into OQL queries. Even though our work has a formal basis (a VOODOO template is nothing but a monoid comprehension [8]), we decided not to express the semantics of VOODOO