Automatic Incrementalization of Object-Set Queries in Object-Oriented Programs *

Tom Rothamel and Yanhong A. Liu

Computer Science Department
Stony Brook University
rothamel,liu@cs.sunysb.edu

Abstract

High-level query constructs help greatly improve the clarity of programs and the productivity of programmers, and are being introduced to increasingly more languages. However, the use of high-level queries can come at a cost to program efficiency, because these queries are expensive to compute and may be computed repeatedly on slightly changed inputs. For efficient computation in practical applications, a powerful method is needed to incrementally maintain query results with respect to updates to query parameters.

This paper describes a general and powerful method for automatically generating incremental implementations of high-level queries over objects and sets in object-oriented programs, where a query may contain arbitrary set enumerators, field selectors, and additional conditions. The method can handle any update to object fields and addition and removal of set elements, and generate coordinated maintenance code and invocation mechanisms to ensure that query results are computed correctly and efficiently. Our implementation and experimental results for example queries and updates in role-based access control, electronic health record policy, etc. confirm the effectiveness of the method.

1. Introduction

High-level query constructs in programming languages help greatly improve the clarity of programs and the productivity of programmers. These query constructs, such as SETL set formers [38], Python comprehensions [34], Boo generator expressions [9], LINQ query operators for C# [22], and JQL for Java [43], allow queries over sets and objects to be written in a concise and abstract manner. This allows the programmer to focus on what is computed, rather than how to compute efficiently.

However, the use of high-level queries can come at a cost to program efficiency, because they are expensive to compute and may be computed repeatedly on slightly changed inputs. To improve efficiency, the results of the queries need to be stored and incrementally maintained when values the queries depend on are updated. This can lead to a drastic and often asymptotic improvement in program running time, especially in programs where queries occur more often than changes.

Currently, this incremental maintenance is mostly performed by hand. Rather than writing a clear high-level query, a programmer is forced to write code that maintains the result of a query in response to updates to values the query result depends on. This code can be complex and error-prone. Even worse, because the updates may be spread throughout a program, the code that incrementally maintains the query result may also be scattered all over the program, making the program difficult to understand and maintain. If an update occurs without the corresponding incremental maintenance, the correctness of the query is compromised.

This leads to a conflict between clear high-level queries and efficient incremental maintenance. Programmers are forced to choose between clear yet slow implementations and efficient yet complex implementations of the same queries. This typically results in performance-critical code being hard to maintain, and less important code being unnecessarily slow. Automatic incrementalization helps resolve this conflict, by transforming clear high-level queries into efficient implementations.

This paper describes a general and powerful method for automatically generating incremental implementations of high-level queries over objects and sets in object-oriented programs, where a query may contain arbitrary set enumerators, field selectors, and additional conditions. The method can handle any update to object fields and addition and removal of set elements, and generate coordinated mainte-

* This work was supported in part by ONR under grant N00014-04-1-0722 and NSF under grants CCR-0306399, CCR-0311512, and CCF-0613913.

nance code and invocation mechanisms to ensure that query results are computed correctly and efficiently. We also describe implementation and experimental results for example queries and updates in role-based access control, electronic health record policy, etc. that confirm the effectiveness of the method.

Our method is practical for several reasons. First, it processes one query at a time and handles any update to the values that the query depends on, using only local analysis, not whole-program analysis, so it scales to programs of large sizes and applies to program modules that may depend on libraries and plug-ins. Second, the method does not change the representations of sets and objects, so incrementalized modules can interact with the remaining modules. Third, the method incurs overhead only on objects that an object-set query depends on. Finally, our method is automatic, allowing it to be applied without user interaction.

There is a large amount of prior work on incrementalization of programs and on incremental query evaluation, as discussed in Section 10. However, there is no existing method that can automatically incrementalize queries over sets and objects in object-oriented programs without changing representations of objects in the programs.

2. Language

Our method applies to any language that supports the following query and update constructs.

Queries. We consider queries of the following form, called *object-set comprehensions*, or *comprehensions* for short.

$$
\begin{aligned}
\mathit{comprehension} &::= \mathit{parameter}^{+}\texttt{->}\\
&\quad\ \texttt{\{}\,\mathit{result\_exp}\,\texttt{:}\,\mathit{enumerator}^{+}\ \mathit{condition}^{*}\texttt{\}}\\
\mathit{enumerator} &::= \mathit{enumeration\_var}\ \texttt{in}\ \mathit{selector}\\
\mathit{selector} &::= \mathit{variable} \mid \mathit{selector}.\mathit{field}\\
\mathit{parameter} &::= \mathit{variable}\\
\mathit{enumeration\_var} &::= \mathit{variable}\\
\mathit{result\_exp} &::= \mathit{expression}\\
\mathit{condition} &::= \mathit{expression}
\end{aligned}
$$

Intuitively, given values of parameters, a comprehension returns the set of values of the result expression for all values of the variables that satisfy the enumerator and condition clauses. We can see that a comprehension may contain arbitrary object field selections and set element enumerations. We require that every variable in a comprehension appear as either a parameter or an enumeration variable. We also require that the result expression and conditions be functions of the values of the variables in the comprehension, i.e., they can be any expressions whose values depend only on the values of the variables in the comprehension and that have no side-effect.

Precisely, the result of evaluating a query can be given in terms of the set of all possible variable assignments, called the *assignment set*, of the query. A *variable assignment* maps each variable in the query to a value. A variable assignment is in the *assignment set* of the query iff (1) each parameter of the query is assigned the given value of that parameter and (2) each enumerator and condition clause in the query is satisfied when evaluated under the variable assignment. The *result set* of the query is the set formed by evaluating the result expression under each variable assignment in the assignment set.

A variable may appear as both a parameter and an enumeration variable. We use *unconstrained parameters*, *constrained parameters*, and *local variables* to refer to parameters that are not enumeration variables, parameters that are also enumeration variables, and enumeration variables that are not parameters, respectively. An enumerator whose enumeration variable is also a parameter is equivalent to a set membership test; we do not treat such tests as conditions, because we can handle them uniformly together with other enumerators for more efficient incremental computation.

Changes. We decompose all changes to query parameters into the following three kinds, and incrementally maintain the query result under all changes of these kinds:

- adding an element to a set.
- removing an element from a set.
- assigning a value to a field of an object.

Running example. We use the following query as a running example:

$$
\begin{aligned}
&wifi\ \texttt{->}\\
&\texttt{\{}\ ap.ssid\ \texttt{:}\ ap\ \texttt{in}\ wifi.scan, ap.strength > wifi.threshold\\
&\texttt{\}}
\end{aligned}
$$

This query is performed on a wifi object, referenced by the variable $wifi$. The object has two fields: a signal strength threshold ($threshold$) and a set of access point objects ($scan$). Each access point object has two fields: a station id ($ssid$) and a signal strength ($strength$). The query has one parameter, $wifi$, the result expression, $ap.ssid$, an enumerator, ap `in` $wifi.scan$, and a condition clause, $ap.strength > wifi.threshold$. The result set of this query contains the $ssid$ field of all aps in $wifi.scan$ such that $ap.strength$ is greater than $wifi.threshold$.

The result of this query can be affected by many kinds of changes: adding to or removing from $wifi.scan$, and assigning to $ap.ssid$, $ap.strength$, $wifi.threshold$, and $wifi.scan$. Furthermore, these changes can be spread anywhere in many components of the program.

Language for generated code. We use a standard object-oriented language that supports operations on sets, maps, and tuples and where all values are considered to be objects.

Figure 1 describes the operations we use on sets, maps, and tuples. All these operations take constant time, assuming that hashing is used in the implementation of sets, including the key sets and image sets of maps. Sets and maps are empty when first created. We only create tuples of a constant length. An image set of a map is a reference to the set, and the set is updated when the map is updated.

s`.empty()`	set s to the empty set
s`.add(`x`)`	add element x to s
s`.remove(`x`)`	remove element x from s
x `in` s	return whether x is an element of s
x `not in` s	return whether x is not an element of s
m`.add(`x`,`y`)`	add mapping from x to y to map m
m`.remove(`x`,`y`)`	remove mapping from x to y from m
m`.img(`x`)`	return image set of key x under m
(x_1,...,x_k)	create a tuple with elements x_1,...,x_k

Figure 1. Operations on sets, maps, and tuples.

We use x == y to denote object identity comparison. It returns true if and only if two values refer to the same object. It is a constant-time operation.

For ease of presentation, in the generated intermediate code, we use special for-loops of the form `for` (x_1,...,x_k) `in` s`:`, where s is a set of tuples of length k, and each x_i may already be bound to some value before the loop. Such a loop iterates over the elements of s that *match the pattern* (x_1,...,x_k)—elements where each component corresponding to a bound component of the pattern equals the value of the corresponding variable in the pattern. This can be done in time proportional to the number of matched elements, by maintaining a map from values of bound components to values of unbound components, looking up the bound components in the map in constant time, and iterating only over the unbound components, as described in [36].

Other than the above, we use standard statements for assignment (v = e), sequencing ($stmt1$ $stmt2$), branching (`if` c`:` $stmt$), and looping (`for` v `in` s`:` $stmt$). We use indentation to indicate scoping.

3. Overview of the method

Our method processes one query at a time and handles any update to the values that the query depends on.

To compute the result of a query efficiently in the presence of all possible changes to the parameters of the query, we maintain the result incrementally with respect to the changes. To do this, we first note that the result of a query can be computed from scratch straightforwardly in two steps. Step 1 computes the assignment set: it creates all possible variable assignments allowed by the enumerators and puts each such variable assignment that also satisfies the conditions in the assignment set. Step 2 computes the result set: it iterates through the assignment set and puts the result of evaluating the result expression under each variable assignment into the result set.

There are five main ideas for efficient incremental computation. (1) We can compute the precise change to the assignment set after any change to the query parameters, called *differential assignment set*, denoted *DAsgnSet*. (2) We can maintain a query result efficiently using *DAsgnSet* by keeping a reference count with each element in a result set. (3) We can return query results efficiently for possibly arbitrary parameter values by maintaining a map from combinations of parameter values to query results. (4) To avoid maintaining query results for all possible values of unconstrained parameters, which would be impractical, we keep combinations of values of unconstrained parameters that have been queried on. (5) To efficiently retrieve objects from field values, and sets from members, as needed for efficient incremental computation, we can maintain inverse maps.

The differential assignment set, *DAsgnSet*. The differential assignment set, *DAsgnSet*, is a set of variable assignments that would be added to or removed from the assignment set by an update. Generating code to compute it efficiently under all possible changes is at the core of our method. We use *DAsgnSet* in maintaining the result set efficiently, avoiding maintaining the entire assignment set. Note that, in general, we can not compute the change to the result set efficiently without computing *DAsgnSet*, because multiple variable assignments may lead to the same value in the result set.

Reference counts for elements in a result set. Exactly because multiple variable assignments may lead to the same value in the result set, to determine whether a value should be in the result set when the assignment set is updated, we must keep a count of the number of variable assignments that produce that value. Addition and removal operations to a result set maintain the reference counts, and do the actual addition and removal of an element only if its reference count changes from 0 to 1 and 1 to 0, respectively. This is important for us to update a result set correctly and efficiently.

Result map, *R*. Instead of creating a new result set for each query instance, i.e., a query with a combination of parameter values, we maintain a map, *R*, called the *result map*, that maps combinations of parameter values to the result sets of the query. We can retrieve the result set of a query from *R*, based the parameter values, using a constant-time access operation. This yields a result set that changes as the result map does, which we call a *live* result set. In many cases, this is acceptable, as the set is used transiently and then discarded. Where necessary, we make a copy of the result set, at cost linear in the size of the result, and

return that. Techniques exist for determining where copying is necessary [14].

Values of Unconstrained Parameters, *UncParamsVals*. We can not maintain query results for all possible values of unconstrained parameters, because we do not know what the values might be. So we keep combinations of values of unconstrained parameters that have been queried on, as a set of tuples, denoted *UncParamsVals*, and only maintain query results for these values of the unconstrained parameters. Note that for each tuple in *UncParamsVals*, we maintain query result for all possible values of constrained parameters, because they are determined by values of unconstrained parameters. So we look up the query result in constant time if the values of unconstrained parameters are found in *UncParamsVals*.

Inverse maps, inv_m and inv_f's. We also maintain the following inverse maps. The map inv_m, where m stands for member, maps an object to the sets that contain it; it is the inverse of the usual mapping from a set to its members. The maps inv_f, one for each field f, maps an object to the objects referring to it through field f; it is the inverse of field selection.

These sets and maps are manipulated by the generated code. *DAsgnSet*, *R*, *UncParamsVals*, and the inverse maps are all stored in variables that are unique to a comprehension; these variables are bound to a single object in all the maintenance code generated for a comprehension, but are bound to different objects in code generated from other comprehensions.

4. Generating code for computing the differential assignment set

We need to generate code to compute the differential assignment set, *DAsgnSet*, for each possible change to the data used by the query. With the current representation of queries, called the *object-domain* representation, changes include assigning new objects to all chains of selected fields, and adding and removing elements of all sets, in the query. To make it much easier to enumerate all possible changes and generate code, we translate the query into a pair-domain representation, enumerate changes and generate code in the pair domain, and then translate the code back to the object domain.

4.1 Translating to the pair domain

The *pair domain* uses sets of pairs to represent the field-value relations and the set-membership relation, though these sets do not exist in the final generated code. This allows us to consider only the addition and removal of pairs as changes. The translation also replaces each field selection with a fresh variable, so every object that the query depends on is referred to by a pair-domain variable. This makes it easy to enumerate all possible changes that can affect the result of a query.

Precisely, we use the following sets in the pair domain:

- For each field f, a set, $field_f$, is used to relate any object with the value of the field f of the object:

$$(o, v) \in field_f \iff v == o.f.$$

- A single set, $member$, is used to relate any set with any member of the set:

$$(s, o) \in member \iff o \in s.$$

Note that $field_f$ is used for same-named fields of different objects in the pair domain, just like $.f$ i is used to access same named fields of different objects in the original object domain.

We translate a comprehension into the pair domain by applying the following two rules repeatedly until they do not apply:

- For each variable o and field f, replace all occurrences of the field selection $o.f$ with a fresh variable, say v, and add a new enumerator (o,v) `in` $field_f$.
- Replace each enumerator v `in` s, where v and s are variables, with a new enumerator (s,v) `in` $member$.

This eliminates all fields and sets in the object domain.

For the wifi query, this yields:

$wifi$ ->
{ ap_ssid : (ap, ap_ssid) `in` $field_{ssid}$,
($wifi$, $wifi\_scan$) `in` $field_{scan}$,
($wifi\_scan$, ap) `in` $member$,
(ap, $ap\_strength$) `in` $field_{strength}$,
($wifi$, $wifi\_threshold$) `in` $field_{threshold}$,
$ap\_strength > wifi\_threshold$ }

This replaces all 4 fields and 1 set in the object domain with 5 sets in the pair domain and increases the number of variables used from 2 to 6.

4.2 Generating code for all possible changes

Recall we need to generate code to compute *DAsgnSet* for each possible change to the data used by the query. Now we do this in the pair-domain.

First, we explain that each change we must handle corresponds to an element addition and/or removal based on an enumerator in the pair-domain comprehension. It is obvious, from the translation, that each occurrence of a field selector, and each retrieval of a set element, corresponds to an enumerator. So, each assignment to a field of an object and each element addition or removal that can affect the query result corresponds to an enumerator—each is indeed changing the object referred to by the left variable in the enumerator. In particular, we have the following:

- An enumerator of the form (o,v) `in` $field_f$ means that if the field f of an object, say o_0, that o refers to is assigned a value v_2, where the value of field before the change is v_1, then the corresponding changes we must handle are removing the pair (o_0,v_1) from $field_f$ followed by adding the pair (o_0,v_2) to $field_f$.
- An enumerator of the form (s,o) `in` $member$ means that if an element, say o_0, is added to a set, say s_0, that s refers to, then the corresponding change we must handle is adding (s_0,o_0) to $member$, symmetrically for removing an element.

Next, for each enumerator, we generate a block of code for computing *DAsgnSet* for either adding an element or removing an element from the set being enumerated. The same block of code is used for both element addition and removal because, for each enumerator, the variable assignments added to the assignment set when an object is added to the set being enumerated are the same as the variable assignments removed from the assignment set when the object is removed from the set.

Note that the generated code refers to the element added or removed. Thus, for removal, the generated code must be run before the removal, and for addition, the generated code must be run after the addition.

Generating code here has two steps. Step 1 creates the clauses that compute the assignment set; this is independent of the enumerator considered. Step 2 determines a nesting order of these clauses that minimizes the cost of executing all clauses, based on enumerator considered.

Generating clauses. We generate one clause for each enumerator and each condition in the pair-domain comprehension. For each enumerator (x,y) `in` s, a for-clause of the following form is generated:

`for` (x,y) `in` s:

For each condition c, an `if`-clause is generated:

`if` c:

We also generate a single for-clause that ensures the values of the unconstrained parameters are in *UncParamsVals*:

`for` unc_params `in` *UncParamsVals*:

where unc_params is a tuple of the unconstrained parameters of the comprehension.

For the wifi query, consider the change that adds ap to $wifi\_scan$. The following clauses are created:

`for` (ap, ap_ssid) `in` $field_{ssid}$:
`for` ($wifi$, $wifi\_scan$) `in` $field_{scan}$:
`for` ($wifi\_scan$, ap) `in` $member$:
`for` (ap, $ap\_strength$) `in` $field_{strength}$:
`for` ($wifi$, $wifi\_threshold$) `in` $field_{threshold}$:
`if` $ap\_strength > wifi\_threshold$:
`for` ($wifi$) `in` *UncParamsVals*:

Nesting clauses. The basic idea of choosing a nesting order is to use the bound values of the variables in the given change to maximize lookups, which take constant time, and to minimize iterations, which have a linear factor. The high-level effect is to minimize the amount of work in incremental computation caused by a change. Doing this exploits the fact that bound variables in a special for-statement use lookups to reduce the amount of iteration needed. While an optimal ordering could be produced, using precise set sizes in the enumerators and costs of the conditions, by a powerful combinatorial optimization algorithm, we have found that using only the distinction between constant time and linear time (in any set size), based on boundness of the variables, with a simple greedy algorithm, works well in practice.

The greedy strategy picks a clause that has the minimum asymptotic running time to execute next, given the set of variables bound so far. The set of bound variables initially contains the variables that appear in the change. This set is used to analyze each clause, using the rules below, to determine if a clause is *runnable*, and if so, what the asymptotic running time is. A runnable clause with the lowest asymptotic running time is chosen, and added to the nesting order. The variables bound by that clause are added to the set of bound variables, and the process is repeated until all clauses are added to the order. Rules for analyzing the clauses are as follows:

- A special for-loop that iterates over a $field_f$ set is runnable if at least one variable in the pattern is bound; this avoids iterating over every object in the program. This for-loop takes constant time if the first variable in the pattern is bound, because it is a field selection, and linear time otherwise.
- A special for-loop that iterates over the $member$ set is runnable if at least one variable in the pattern is bound; this avoids iterating over every set in the program. This for-loop takes constant time if both variables in the pattern are bound, because it is a set membership test, and linear time otherwise.
- The special for-loop that iterates over *UncParamsVals* is always runnable. It takes constant time if all variables in the pattern, i.e., all unconstrained parameters of the query, are bound, and linear-time otherwise.
- A `if`-clause is runnable if all of the variables in it are bound. All `if`-clauses are considered to take constant time by default.

A nesting order will always be computed, ensured by the clause that iterates through *UncParamsVals*. This is because the definition of object-set comprehensions ensures that ev-

```
after adding ap to a set that wifi_scan refers to:

  for (ap, ap_ssid) in field_ssid:
    for (ap, ap_strength) in field_strength:
      for (wifi_scan, ap) in member:
        for (wifi, wifi_scan) in field_scan:
          for (wifi, wifi_threshold) in field_threshold:
            if ap_strength > wifi_threshold:
              for (wifi) in UncParamsVals:
                DAsgnSet.add(var_asgn())
```

Figure 2. Generated pair-domain code for computing *DAsgnSet* for the running example.

pair-domain construct	for (*x*,*y*) in $field_f$: *block*	for (*x*,*y*) in *member*: *block*
x bound *y* bound	if *y* == *x*.*f*: *block*	if *y* in *x*: *block*
x bound *y* unbound	*y* = *x*.*f* *block*	for *y* in *x*: *block*
x unbound *y* bound	for *x* in inv_f.img(*y*): *block*	for *x* in inv_m.img(*y*): *block*

Table 1. Rules for translating back to the object domain.

ery variable is reachable, through a path containing selection and enumeration, from at least one unconstrained parameter. As each selection and enumeration corresponds to a pair-domain clause, there is a path of pair-domain clauses from the unconstrained parameter to the variable. When the unconstrained parameters become bound by the statement iterating over *UncParamsVals*, all for-loops will become runnable, allowing all variables to be bound. This then ensures that every `if`-statement is runnable, allowing every statement to be placed in the nesting order.

Once all variables are bound to some values, these variables and values are used to create a variable assignment, which is then added to *DAsgnSet*. Let $var\_asgn()$ be a function that creates a variable assignment for these variables using their bound values. Figure 2 shows the generated code in the pair domain for computing *DAsgnSet* under one update that affects the result of the wifi query.

4.3 Translating back to the object domain

This translation eliminates field and member sets in the pair domain, and replaces special for-loops with standard statements. The translation uses the rules in Table 1. It gives code for special for-loops over $field_f$ sets and over the $member$ set, and for all three possible combinations of boundness of the two variables in the pattern—recall that we do not have the case when both variables are unbound.

- When both variables are bound, loops over $field_f$ sets are field-value tests, and loops over the $member$ set are membership tests.
- When the first argument is bound but the second is not, loops over $field_f$ sets are field selections, and loops over the $member$ set are element retrievals.
- When the second variable is bound but the first is not, inverse maps are used for reverse retrievals for both field selections and element retrievals.

In the third case, we also generate code for incrementally maintaining the inverse maps, as given in Table 2; it is easy to see that these maps takes constant time to maintain and has a constant-factor space overhead.

Note that computing *DAsgnSet* for a change uses these inverse maps. Thus, for element addition, inverse maps must be updated before computing *DAsgnSet*, and for element removal, the inverse maps must be updated after computing *DAsgnSet*.

for (*x*,*y*) in $field_f$: *block*	for (*x*,*y*) in *member*: *block*
when an object is first referred to by x: inv_f.add(*x*.*f*, *x*)	*when an object is first referred to by x*: for *y* in *x*: inv_m.add(*y*, *x*)
before assignments to x.*f*: inv_f.remove(*x*.*f*, *x*)	*before x*.remove(*y*): inv_m.remove(*y*, *x*)
after assignments to x.*f* inv_f.add(*x*.*f*, *x*)	*after x*.add(*y*): inv_m.add(*y*, *x*)

Table 2. Rules for generating code for maintaining inverse maps.

The special for-loop over *UncParamsVals* is implemented similarly. If some variables in the pattern are not bound, we maintain a map from values of bound variables to values of unbound variables. These maps are incrementally updated when *UncParamsVals* changes. This allows each matched element to be retrieved in constant time, and the entire for-loop to take linear time in the number of matched elements.

For the wifi query and the update we considered, the code in Figure 3 is generated.

```
after adding ap to a set that wifi_scan refers to:

  ap_ssid = ap.ssid
  ap_strength = ap.strength
  if ap in wifi_scan:
    for wifi in inv_scan.img(wifi_scan):
      wifi_threshold = wifi.threshold
      if ap_strength > wifi_threshold:
        if (wifi) in UncParamsVals:
          DAsgnSet.add(var_asgn())
```

Figure 3. Generated object-domain code for computing *DAsgnSet* for the running example.

5. Generating Code for Maintaining the Result Map

Once *DAsgnSet* is computed, it is used to update the result map *R*. For each block of code that computes *DAsgnSet*, we generate another block of code that updates *R*.

For maintenance after assigning a value to a field or adding an object to a set, we generate the code below, where $params(a)$ takes a variable assignment a and returns a tuple containing the values of the parameters of the query, and $eval(e,a)$ evaluates expression e under the variable assignment a. *DAsgnSet* is reset to empty after it is used for updating R, thus is empty and takes no space when we are not executing maintenance code.

```
for a in DAsgnSet:
  R.add(params(a), eval(result_exp, a))
DAsgnSet.empty()
```

For maintenance before assigning a value to a field or removing an object from a set, the generated code is the same except with `add` replaced with `remove`.

The result map maintenance code must be run after all code for computing *DAsgnSet* for a given change has been run. Aliasing could cause problems otherwise. For example, in the query below, suppose $p.S$ and $p.R$ are aliased to the same set, say called SR. When an object, say o, is added to set SR, two updates occur: one adds an object to a set that $p.S$ refers to, and the other adds an object to a set that $p.R$ refers to. The *DAsgnSet* for both updates will contain the variable assignment $\{x \mapsto o, y \mapsto o\}$.

```
p -> {x : x in p.S,  y in p.R,  x == y }
```

By running the result map maintenance code after all code for computing *DAsgnSet*, we ensure that each variable assignment causes a result mapping to be added to the result map once, and only once. This maintains the invariant that the reference count of a mapping in the result map equals the number of variable assignments in the assignment set projecting onto that mapping.

6. Organizing Maintenance Code

Three kinds of maintenance code have been generated: (1) code that maintains inverse maps, (2) code that computes *DAsgnSet*, and (3) code that maintains *R*. They must be run in response to updates to objects, including set objects.

We organize maintenance code based on the pair-domain variables. This is for two reasons, from Section 4: (1) each object that the query result depends on is referred to by a pair-domain variable, and (2) each block of maintenance code generated is for an update to an object that a pair-domain variable refers to, or when the object is first referred to by the variable. We put together, conceptually, all maintenance code that handles updates to the object referred to by a pair-domain variable v, and we call it *an obligation* any object referred to by v must fulfill; we use v as the id of the obligation. Note that an object may have multiple obligations, because it may be referred to by more than one pair-domain variable, a.k.a. aliasing.

Recall that, among the three kinds of maintenance, *R* must be maintained after *DAsgnSet* is computed, and for addition, inverse maps must be maintained before *DAsgnSet* is computed and *R* is maintained, and all three must be done after the addition, while for removal, inverse maps must be maintained after *DAsgnSet* is computed and *R* is maintained, and all three must be done before the removal. When an object has multiple obligations, we run the maintenance code of the same kind from all obligations, before running maintenance code of another kind, for the same reasons as before. Note that *R* is only updated once, by the first block of maintenance code for *R*, because *DAsgnSet* is reset to empty at the end of it, and later blocks of code have no effect.

Obligations are assigned to objects using the function $assign\_obligation$. It takes an object o and an obligation 'v' as arguments. It does nothing if o is already assigned obligation 'v'. Otherwise, it (1) runs any maintenance that needs to be run when o is first referred to by variable v, and (2) registers the maintenance code corresponding to v, separately for addition and removal of course, with o, so that it is called when addition and/or removal occurs. Maintenance code for (1) includes code for updating inverse maps, as in Section 4.3, and assigning obligations to other object, as described below. Implementation for (2) depends on the host language, which we will describe for Python, in Section 9; similar ideas apply to other languages.

Two mechanisms are used to assign obligations to objects. Obligations are assigned to unconstrained parameters by the query execution code discussed in the next section. Obligations are assigned to enumeration variables, i.e., constrained parameters and local variables, by maintenance code associated with other obligations.

Assigning obligations to enumeration variables. The code that assigns obligations to enumeration variables is generated following a reachability-based approach. We start by initializing a set, called the set of supported variables, to the unconstrained parameters of the comprehension. We then search for a pair-domain enumeration of the form:

(x,y) `in` s

where x is in the set of supported variables, and y is not. This clause is then used to create obligation assignment code, as given below, and y is added to the set of supported variables. This process repeats until all variables are added to the set of supported variables. This process will always complete because all variables are reachable from the unconstrained parameters.

The obligation assignment code generated depends on the set s in the enumeration. If s is a $field_f$ set, we generate the following code:

when obligation 'x' is assigned to an object referred to by x:
  $assign\_obligation(x.f, 'y')$

when an object with obligation 'x' has field f assigned:
  $assign\_obligation(x.f, 'y')$

If it is the set $member$, we generate:

when obligation 'x' is assigned to an object referred to by x:
  `for` i `in` x:
    $assign\_obligation(i, 'y')$

when an object with obligation 'x' has element i added:
  $assign\_obligation(i, 'y')$

Note that obligation assignment code is classified as maintenance code of kind (1), because it is done when an object is first referenced by a variable, which also causes inverse maps to be updated.

The method above ensures that if an object is ever referred to by a variable v, the object is assigned obligation v. An obligation assigned to an object is never removed from the object. So, the cost of assigning obligations is constant amortized over object creation, element addition, and field assignment. However, the maintenance code may be run even after an object can no longer affect the result of a query, until it is garbage collected.

7. Generating Code for Executing the Query

Finally, we generate code for query executing. Recall that we keep combinations of values of unconstrained parameters that have been queried on. Each query, for a combination of values of unconstrained parameters, is computed once from scratch—the first time it is encountered; after that, the query result is incrementally maintained.

The query execution code first determines if the query is being incrementally maintained, i.e., if a tuple consisting of the values of the unconstrained parameters is in the set *UncParamsVals*. If it is, then the incrementally maintained result is returned. If not, the query execution code computes the result from scratch, and begins incremental maintenance; this has four steps:

1. Call $assign\_obligation$ to assign $obligation_p$ to the object referred to by each unconstrained parameter p. This will then ensure that obligations are assigned to every object the query depends on.
2. Add a tuple of the values of the unconstrained parameters to the set *UncParamsVals*.
3. Compute *DAsgnSet* for the addition to *UncParamsVals* in Step 2, using the method in Section 4.
4. Maintain the result map, using the method in Section 5.

At the end, the values of the unconstrained parameters are used to retrieve a live result set from the result map. This set is the result of the query.

For our running example, the generated query execution code is given in Figure 4.

```
if (wifi) not in UncParamsVals:

  assign_obligation(wifi, obligation_wifi)
  UncParamsVals.add((wifi))

  wifi_scan = wifi.scan
  wifi_threshold = wifi.threshold
  for ap in wifi_scan:
    ap_ssid = ap.ssid
    ap_strength = ap.strength:
    if ap_strength > wifi_threshold:
      if (wifi) in UncParamsVals:
        DAsgnSet.add(var_asgn())

  for a in DAsgnSet:
    R.add((wifi), eval(ap_ssid, a))
  DAsgnSet.empty()

return R.img((wifi))
```

Figure 4. Generated code for executing the query for the running example.

8. Discussion

The correctness of our method is guaranteed by the invariants maintained by the generated code for each kind of code generated. The cost of our method is linear in the size of

the given program, because all analyses are local, and all transformations are 1-1 correspondence. The costs of individual operations in the generated code are described with the method; we summarize below the overall performance of the generated code.

Our method allows many extensions and additional optimizations, such as handling tuples in queries (by translating them into objects and back), supporting aggregate operations (such as count and sum), and simplifying the generated code (by eliminating pair-domain variables that only do copying). We describe below an interesting join optimization.

Performance. To allow the second and later executions of a query to occur in constant time, our method requires that updates to data maintain the result map. The cost of this maintenance depends on the structure of the comprehension and the update.

For example, for the wifi query, assuming that each ap is in the $scan$ field of exactly one wifi object, the maintenance corresponding to the following changes will be performed in constant time: (1) assigning to $ap.strength$, (2) assigning to $ap.ssid$, and (3) adding an ap to or removing an ap from set $wifi.scan$. Other updates require linear time to maintain the result: assigning to $wifi.threshold$, and assigning to $wifi.scan$, i.e. assigning a new set to the $wifi.scan$ field, not updating the content of the set. The linear time for this maintenance is because a single value for ap cannot be determined from the update, so iteration over the $wifi.scan$ set is needed.

Our maintenance code is asymptotically faster than recomputation code when the values of the variables bound at an update allow some iterations to be eliminated, and thus, our method will always produce an asymptotic speedup as long as the frequency of updates is not asymptotically higher than that of queries. In no case does our method produce incremental update code that is asymptotically slower than code that executes the query from scratch, because the worst-case maintenance code is identical to recomputation code; therefore, when the frequency of queries is asymptotically the same as that of updates, our method will never produce an asymptotically slower program.

Optimizing joins. Equality joins on object identity asserts that two fields must refer to the same object. They are expressed in object-set comprehensions as conditions of the following form—recall that a selector is a variable optionally followed by one or more field selections:

$$selector == selector$$

Our method already handles joins as conditions, but we can optimize them and further decrease the asymptotic running time.

We optimize joins by modifying the pair-domain representation of the query. After translating the query into the pair domain, we search for conditions of the form $v1 == v2$. Because selectors are always translated to pair-domain variables, all joins will be of this form. When a join is found, it is eliminated from the pair-domain comprehension. In the remaining conditions, enumerations, and the result expression, $v1$ and $v2$ are replaced with the new variable $v1\_v2$. We repeat this process until no joins remain.

This optimization reduces the number of variables in the comprehension. It usually increases the number of generated clauses containing bound variables, and thus can decrease the asymptotic running time of the code for computing *DAsgnSet*.

9. Implementation and Experiments

To evaluate our method, we developed an implementation in Python. Our implementation consists of a single Python module, named `incr.py`. It consists of 617 lines of Python, and requires only the Python standard library to run. It provides as a public interface a single function, `run_query`, which takes as arguments a comprehension with values for its parameters, and returns the result of the query. This function first checks to see if it has encountered the query before. If not, it generates obligation and query execution code corresponding to the query, passes the parameters to the query execution code, and returns the result.

We implement obligations by creating classes. Each object in the system starts with an initial class, the class it was initially created with. For each combination of initial class and set of obligations, we create a new class that inherits from the initial class and runs the maintenance code in the obligations. When an obligation is assigned to an object, we find the class corresponding to the object's initial class and the set of obligations. We assign this new class to the object, an operation Python allows. We then run any initial code required by the obligation being assigned to the object.

Our experiments were run on a computer with an Intel Core 2 Duo processor running at 2.13GHZ. The machine has 2GB of RAM, although memory usage was not a concern in our experiments. Our experimental code is written in Python, using Python 2.4.4, running under Ubuntu Linux. All times reported are CPU usage. They do not include the time used for code generation, which was about 0.2 seconds for the running example.

All of the programs that our method has been applied to were written by us. Some may object to this, preferring that our method be applied to open source programs written by others. However, open-source programs were generally written with efficiency in mind, and the programmers generally incrementalized the repeated expensive computations by hand. Our method would not improve the performance of such programs. Instead, our method would allow programmers to write simpler, more readable, and more maintainable programs, and perform incrementalization automatically.

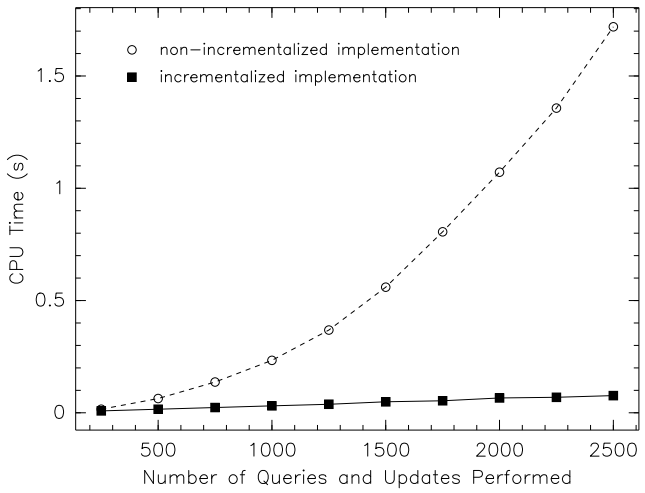

Figure 5. Time taken to perform a varying number of updates and queries, using our running example.

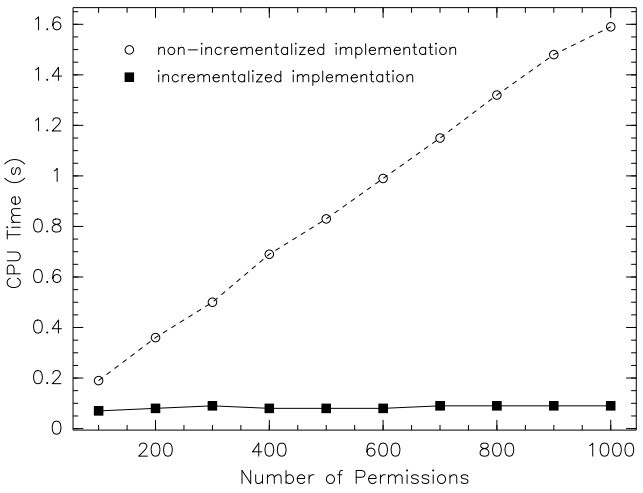

Figure 6. Time taken to perform 10,000 RBAC UserOperationsOnObject queries, for a varying number of database permissions.

Running example. Our experiment consisted of creating a single $wifi$ object, and looping n times, where n varied between 250 and 2500 at intervals of 250. In each iteration loop, we create an ap object and add it to $wifi.scan$, and then perform the wifi query. All ap objects were created with $ap.signal > wifi.threshold$, so the number of objects returned from this query equals to the number of iterations through the loop. For a given n, each experiment was repeated 20 times, with the reported times being the average of 20 runs.

Figure 5 shows the results of this experiment. The incrementalized implementation is linear in the number of queries and updates performed. This is consistent with our prediction that the incrementalized code can perform the query and the update in constant time. The running time of the non-incrementalized incrementalization is quadratic, reflecting the increase in time it takes to perform a query as sets get larger.

Role-based access control. Role-Based Access Control (RBAC) [12, 6] is an ANSI standard for controlling access to operations on objects. Core RBAC controls access by assigning permissions to perform operations on objects to roles, and then assigning users to those roles. To demonstrate how our method can be used to simplify an implementation of Core RBAC, we created an object-oriented variation, OO-RBAC. In OO-RBAC, users, roles, sessions, operations, objects, and permissions are all implemented as objects. The role to permission and user to role assignment multimaps are implemented as fields on the user and role objects, respectively.

Our method was used to automatically create incremental implementations of the Core RBAC review functions. One such function is UserOperationsOnObject: given a user and an object, it returns the operations the user can perform on that object:

$$
\begin{array}{l}
rbac, user, object \texttt{ ->} \\
\texttt{\{ } perm.operation \texttt{ : } user \texttt{ in } rbac.users, \\
\qquad\qquad\qquad\quad role \texttt{ in } user.roles, \\
\qquad\qquad\qquad\quad perm \texttt{ in } role.perms, \\
\qquad\qquad\qquad\quad perm.object == object \texttt{ \}}
\end{array}
$$

The $rbac$ parameter and the first clause make $user$ and $object$ constrained parameters, rather than unconstrained parameters. This allows us to perform repeated queries involving the same $rbac$ in constant time.

To show how incrementalization improves this query performance, we populated the RBAC database with 10 users, 1 role (assigned to all of the users), and a varying number of permissions (all assigned to the role), with each permission granting access to a unique object. We then timed how long it took to perform 10,000 queries.

Figure 6 shows the results of this experiment. The non-incrementalized implementation obviously takes linear time, as each query needs to access every permission. Our incrementalized implementation also takes linear time, but with a much less steep slope. This is because our method causes the first query to take linear time in the number of permissions, while the second and later queries take a constant amount of time. This gives substantial time savings when the database is large.

This decrease in query time is paid for by an increased cost of updates. For this query, adding a new permission to a role takes time proportional to the number of users assigned that role. This is exactly the size of the change to the result set.

There are 11 possible updates that can affect the result of this comprehension: assigning to 5 fields, adding to 3 sets, and removing from 3 sets. Of these, OO-RBAC only uses

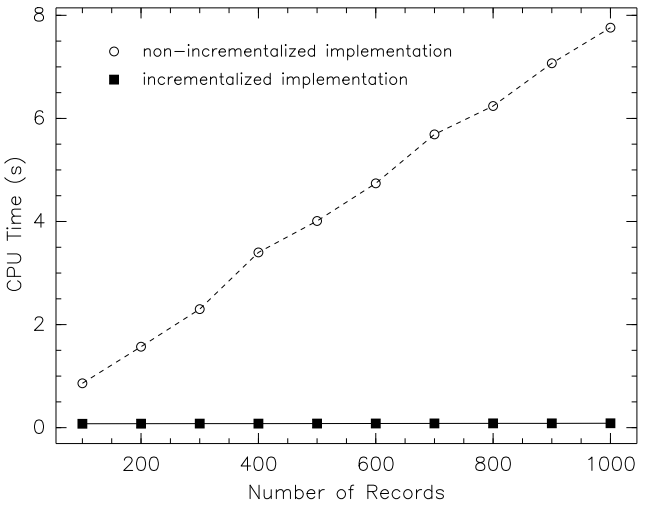

Figure 7. Time taken to perform 10,000 queries, for a varying number of electronic patient records.

the 6 set update operations. As each of these set updates corresponds to an administrative function, the use of our method lets us remove code from these 6 functions and centralize it in a single query.

Electronic health record policy. We have also written an object-oriented version of the United Kingdom's Electronic Health Record (EHR) service policy, based on the specification by Becker [7], and applied our automatic incrementalization method to it. The following query corresponds to a rule that determines if a clinician is authorized to view a patient's electronic patient record into an object-set query:

```
org, cli ->
{ record: tm in org.team_memberships,
          tm.cli == cli,
          tm.spcty == cli.spcty,
          record in org.records,
          record.group == tm.team }
```

It finds the set of records a clinician is authorized to view, based on his team memberships and specialty.

To demonstrate how our method improves the query performance, we populated the database with a single clinician, team, and team membership, and with a varying number of records, all of which are accessible to members of the team. We then measured the time to perform 10,000 queries that determined the records accessible to the clinician.

As shown in Figure 7, the results of this experiment are similar to the results of the RBAC experiment. While both queries take linear time, the very low slope of the incrementalized version makes it look constant. In both EHR and RBAC, our method is able to replace expensive duplicate queries with constant-time retrievals, significantly improving program performance.

10. Related Work and Conclusion

Incrementalization of programs has been a subject of much research, and automatic incrementalization techniques have been developed for queries in many areas. Our method improves over previous methods in two main respects, putting aside many finer distinctions. First, our method handles a query as a whole, while most previous methods decompose it into smaller queries that need to be maintained independently, which may have additional cost in time and memory. Second, our method incrementalizes high-level queries in object-oriented programs, while previous methods handle only sets and tuples, use representations of objects that are not suitable for incrementalized modules to interact with the remaining modules, or incrementalize only with the granularity of method calls.

The earliest work in this area was intended to provide a way of performing strength reduction on sets and maps [11, 13], ultimately yielding the finite differencing method [30, 40]. This work, while automatic, only considered sets and pairs, and decomposed queries to incrementalize them. Finite differencing also requires finite differencing rules to be given manually. Our method can derive many of those rules automatically.

In database area, techniques for incremental view maintenance over sets of tuples, as in relational databases, have been known for a long time [42, 8]. Finite differencing was applied to the problems of integrity constraints [20, 29] and incremental view maintenance [35, 17] in databases containing sets of tuples. It has also been extended to support views with duplicates [16]. These methods work by decomposing queries into incrementalizable parts. A method exists that handle a query as a whole and reference-counts intermediate results [18], as ours does the result map. However, all these methods do not handle objects.

Other methods are also used for incremental view maintenance, in relational or object-oriented databases. One approach is to use a database engine to perform queries with some parameters bound, in relational [10] or object-oriented [4, 3] databases. This is similar to our strategy that computes *DAsgnSet* without decomposing a query into sub-queries. Unlike these techniques, our method generates code for computing *DAsgnSet* and does not require executing queries in a database engine.

A number of other methods deal with incremental view maintenance in object-oriented databases. These methods either create their own copies of objects used during incremental maintenance [45, 21], or require the system to represent objects in ways not suitable for program execution [5, 15, 27]. In contrast, our method can use existing objects and allow queries to return existing objects, without changing object representations, and we only require that objects support the ability to intercept method and field accesses. This allows our incrementalized components to be

usable in the same way in all the contexts they were usable before.

The problem of choosing an optimal order for nesting clauses when generating *DAsgnSet* computation code is similar to join order optimization [19]. The optimal solution to this problem is known to be NP-complete. While a number of approaches to solving this problem exist [41], these require estimates of the size of each set used, and the selectivity of each join. When *DAsgnSet* computation code is generated, much of this information might not be available. Our heuristic produces good and reasonable results for object-oriented programs when such information is not available. Note that it takes into account the different costs of executing the same special for-loop in different code contexts.

There has also been work on incrementalizing object-oriented programs. One method is based on applying incrementalization rules [24, 26], but those rules are written manually. Our method can automatically generate incremental implementations. While our method incrementalizes high-level queries in object-oriented programs, other methods in incrementalization of programs are developed for languages based on sets or bags [13, 30, 40, 44], functional programs [33, 25, 2, 1], logic programs [18, 37, 23], XSLT programs [28], and functions in object-oriented programs [39]. The last one is the most closely related and is discussed separately next.

The Ditto system [39] incrementalizes queries that consist of recursive functions in Java, using a method similar to those for incrementalizing functional programs [33, 2, 1]. This method works by memoizing the results of method calls, and recomputing a stored result of a method call when a field accessed by the method changes or when the result of a call to another method changes. Recomputation is done by re-running the method, hence Ditto has a method-call-level granularity for incrementalization. Thus, any update to data used by a high-level query in a method will result in recomputation of the method including the entire high-level query. Ditto also restricts the results of queries to be of primitive types, while our method allows the results of queries to be sets of elements of any type.

One might note that the obligations we generate can be thought of as aspects, and their application can be considered to be dynamic aspect-oriented programming (AOP) [32, 31]. Implementation techniques developed for AOP can be used to apply our obligations to objects at runtime, but programmers would have to write incremental update code for different queries as aspects. Our method allows programs to be written clearly and modularly by generating efficient implementations automatically, without the need to write incremental update code by hand.

References

[1] U. A. Acar, G. E. Blelloch, M. Blume, and K. Tangwongsan. An experimental analysis of self-adjusting computation. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 96–107, New York, NY, USA, 2006. ACM Press.

[2] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 247–259, New York, NY, USA, 2002. ACM Press.

[3] R. Alhajj and A. Elnagar. Incremental materialization of object-oriented views. *Data and Knowledge Engineering*, 29(2):121–145, 1999.

[4] R. Alhajj and F. Polat. Incremental view maintenance in object-oriented databases. *SIGMIS Database*, 29(3):52–64, 1998.

[5] M. A. Ali, A. A. A. Fernandes, and N. W. Paton. MOVIE: An incremental maintenance system for materialized object views. *Data and Knowledge Engineering*, 47(2):131–166, 2003.

[6] American National Standards Institute, Inc. Role-based access control. ANSI INCITS 395-2004.

[7] M. Y. Becker. A formal security policy for an NHS electronic health record service. Technical Report UCAM-CL-TR-628, University of Cambridge, March 2005.

[8] P. A. Bernstein, B. T. Blaustein, and E. M. Clarke. Fast maintenance of semantic integrity assertions using redundant aggregate data. In *Proceeding of the 6th international conference on Very Large Databases (VLDB)*, Montreal, Canada, Oct. 1980.

[9] The boo programming language. PDF, 2005. `http://boo.codehaus.org/BooManifesto.pdf`.

[10] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB '91: Proceedings of the 17th international conference on Very Large Data Bases*, pages 577–589, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[11] J. Earley. High level iterators and a method for automatically designing data structure representation. *Journal of Computer Languages*, 1(4):321–342, 1976.

[12] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[13] A. C. Fong and J. D. Ullman. Induction variables in very high level languages. In *POPL '76: Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 104–112, New York, NY, USA, 1976. ACM Press.

[14] S. M. Freudenberger, J. T. Schwartz, and M. Sharir. Experience with the SETL optimizer. *ACM Transactions on Programming Languages and Systems*, 5(1):26–45, 1983.

[15] D. Gluche, T. Grust, C. Mainberger, and M. H. Scholl. Incremental updates for materialized OQL views. In *Deductive and Object-Oriented Databases*, pages 52–66, 1997.

[16] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of Data*, pages 328–339, New York, NY, USA, 1995. ACM Press.

[17] T. Griffin, L. Libkin, and H. Trickey. An improved algorithm for the incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):508–511, 1997.

[18] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of Data*, pages 157–166, New York, NY, USA, 1993. ACM Press.

[19] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems*, 9(3):482–502, September 1984.

[20] S. Koenig and R. Paige. A transformational framework for the automatic control of derived data. In *Proceedings of the 7th international conference on Very Large Data Bases*, pages 306–318, Sept. 1981.

[21] H. A. Kuno and E. A. Rundensteiner. Incremental maintenance of materialized object-oriented views in MultiView: Strategies and performance evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):768–792, 1998.

[22] The LINQ project. web page, March 2006. `http://msdn.microsoft.com/netframework/future/linq/`.

[23] Y. A. Liu and S. D. Stoller. From datalog rules to efficient programs with time and space guarantees. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and Practice of Declaritive Programming*, pages 172–183, New York, NY, USA, 2003. ACM Press.

[24] Y. A. Liu, S. D. Stoller, M. Gorbovitski, T. Rothamel, and Y. E. Liu. Incrementalization across object abstraction. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object Oriented Programming Systems, Languages, and Applications*, pages 473–486, New York, NY, USA, 2005. ACM Press.

[25] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems*, 20(3):546–585, May 1998.

[26] Y. A. Liu, C. Wang, M. Gorbovitski, T. Rothamel, Y. Cheng, Y. Zhao, and J. Zhang. Core role-based access control: efficient implementations by transformations. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial Evaluation and semantics-based Program Manipulation*, pages 112–120, New York, NY, USA, 2006. ACM Press.

[27] H. Nakamura. Incremental computation of complex object queries. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 156–165, New York, NY, USA, 2001. ACM Press.

[28] M. Onizuka, F. Y. Chan, R. Michigami, and T. Honishi. Incremental maintenance for materialized xpath/xslt views. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 671–681, New York, NY, USA, 2005. ACM Press.

[29] R. Paige. Applications of finite differencing to database integrity control and query/transaction optimization. In H. Gallaire, J. Minker, and J.-M. Nicolas, editors, *Advances in Database Theory*, volume 2. Plenum Press, New York, 1984.

[30] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, 1982.

[31] A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for Java. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-Oriented Software Development*, pages 100–109. ACM Press, 2003.

[32] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-Oriented Software Development*, pages 141–147. ACM Press, 2002.

[33] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 315–328. ACM Press, 1989.

[34] Python 2.4.2 documentation. web page, September 2005. `http://www.python.org/doc/2.4.2/`.

[35] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *Knowledge and Data Engineering*, 3(3):337–341, 1991.

[36] T. Rothamel and Y. A. Liu. Efficent implementation of tuple pattern based retrieval. In *PEPM '07: Proceedings of the workshop on Partial Evaluation and semantics-based Program Manipulation*, pages 81–90, Nice, France, January 2007. ACM Press.

[37] D. Saha and C. R. Ramakrishnan. A local algorithm for incremental evaluation of tabled logic programs. In S. Etalle and M. Truszczynski, editors, *ICLP*, volume 4079 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 2006.

[38] J. Schwartz. Programming in SETL. web page. (draft in progress) `http://www.settheory.com`.

[39] A. Shankar and R. Bodik. Ditto: Automatic incrementalization of data structure invariant checks (in Java). To appear in PLDI 2007.

[40] M. Sharir. Some observations concerning formal differentiation of set theoretic expressions. *ACM Transactions on Programming Languages and Systems*, 4(2):196–225, 1982.

[41] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB Journal: Very Large Data Bases*, 6(3):191–208, 1997.

[42] M. Stonebraker. Implementation of integrity constraints and views by query modification. In *SIGMOD '75: Proceedings of the 1975 ACM SIGMOD international conference on Management of Data*, pages 65–78. ACM Press, 1975.

[43] D. Willis, D. J. Pearce, and J. Noble. Efficient object querying for Java. In D. Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 28–49. Springer, 2006.

[44] D. M. Yellin and R. E. Strom. INC: A language for incremental computations. *ACM Transactions on Programming Languages and Systems*, 13(2):211–236, 1991.

[45] G. Zhou, R. Hull, and R. King. Generating data integration mediators that use materialization. *Journal of Intelligent Information Systems*, 6(2/3):199–221, 1996.