The Efficient Implementation of Partially-Bound Retrieval *

Tom Rothamel   Yanhong A. Liu
Stony Brook University
{rothamel,liu}@cs.sunysb.edu

Abstract

Partially-bound retrieval is a language construct that matches a tuple pattern against a set of tuples to retrieve components of those tuples, where a tuple pattern consists of components that are either expressions or unbound variables. We give a syntax and semantics for this construct, and then present a method for its efficient implementation. We describe two tools that we have developed that implement this method, and through experiments show that it can simplify writing programs in a range of applications, including graph algorithms, program analysis, and security.

1. Introduction

An ongoing and welcome trend in the evolution of programming languages is the addition of support for higher-level data types into the language proper. This integration provides the programmer with a twofold benefit. By providing constructs for working with high-level data types (such as sets and lists) in the language proper, boilerplate code is eliminated and programs are easier to understand. At the same time, such constructs make the queries and updates that are performed explicit, making it easier to develop tools that provide for their efficient implementation. By including the right set of constructs in a language, we can improve code readability while at the same time maintaining or improving practical performance.

A new language construct that meets this test is *partially-bound retrieval*, which is the matching of a tuple pattern against a set or list of tuples to retrieve components of those tuples. A tuple pattern consists of components that are either expressions or unbound variables. When a tuple matches the pattern, that is, when they are of the same length and the values of the expressions in the pattern are equal to the corresponding components of the tuple, then the unbound variables in the pattern are assigned their corresponding components of the tuple. This allows for simple but powerful queries over sets and lists of tuples, inspired by how such queries are often written in pseudocode.

For a language construct to be accepted, it should be at least as efficient as the equivalent hand-written user code. Otherwise, programmers will be faced with a choice between simple but slow code and efficient but complicated code. Partially bound retrieval does not force this choice upon a programmer. We design data structures that allow us to efficiently locate tuples matching a pattern, and we incrementally maintain these data structures when the set or list they represent is updated. These data structures allow us retrieve a single matching tuple in amortized constant time, and to iterate through the tuples matching a pattern in amortized time proportional to the number of tuples matched. As the time spent at each update to maintain our data structures is also constant, these times are asymptotically optimal.

Finally, it is important to show that partially-bound retrieval can benefit practical applications. To do this, we have developed two implementations of partially-bound retrieval, and used them to perform a number of experiments. These experiments show how this new construct can simplify programming a variety of applications, in problem domains from program analysis to security policy frameworks. For each problem, we give the size of a program written using partially-bound retrieval, and the size of the program translated into a language that does not support it. For several examples, we present the performance of that program translated in both straightforward and optimal ways. In this way, we show that partially-bound retrieval can lead to code that is both simple and efficient.

We expand on these concepts throughout the rest of this paper. Section 2 presents a syntax and semantics for partially-bound retrieval, detailing how it can be used as part of the `while`, `if`, and `for`, statements of a language. Section 3 develops a method for the efficient implementation of this construct, and shows that this implementation is asymptotically optimal. Section 4 discusses related issues. Section 5 describes the tools we have developed to implement partially-bound retrieval, while section 6 presents the results of our experiments. The final section of this paper discusses related work and concludes.

2. Syntax and Semantics

In this section, we give the syntax and semantics of partially-bound retrieval in two ways. We first informally describe the syntax and semantics as they are used in a short example program. We then describe the syntax and semantics precisely.

2.1 Example

An example of the use of partially-bound retrieval is given in figure 1, which presents a program that topologically sorts the vertices of a graph. Apart from partially-bound `for`, `while`, and `if`statements, the language contains statements that add and remove tuples to and from sets, as well as statements to read from input and print output. It also includes a function that constructs a new empty set. We represent tuples by comma-separated list of components enclosed in parenthesis. To easily distinguish expressions from unbound variables in tuple patterns, we have underlined all expressions used in tuple patterns.

* The authors gratefully acknowledge the support of ONR under grants N00014-04-1-0722 and N00014-01-1-0109 and NSF under grant CCR-0306399.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]. . . $5.00.

```
1  read VERTICES, EDGES
2  INDEGREES = Set()
3
4  for v1 in VERTICES:
5      indegree = 0
6      for (v2, v1) in EDGES:
7          indegree += 1
8
9      INDEGREES add (v1, indegree)
10
11 while (v1, 0) in INDEGREES:
12     for (v1, v2) in EDGES:
13         if (v2, indegree) in INDEGREES:
14             INDEGREES remove (v2, indegree)
15             INDEGREES add (v2, indegree - 1)
16
17     INDEGREES remove (v1, 0)
18     print v1
```

Figure 1. Topological Sort, written using partially-bound retrieval.

This example contains four statements involving partially-bound retrieval. There are two `for` statements, on lines 6 and 12, one `while` statement on line 11, and one `if` statement on line 13. Since the block starting at line 11 is the part of the program that actually computes the topological order (the rest of the program being initialization), we discuss the first three statements in that block in detail.

`INDEGREES` is a set of pairs, each of which consists of a vertex that has not been printed yet, and the number of edges into that vertex from other vertices found as the first component of a tuple in `INDEGREES`. Throughout execution, we maintain the invariant that each vertex is the first component of at most one pair in `INDEGREES`. The `while` loop on line 11 continues as long as there is at least one pair in the `INDEGREES` set that has a second component that is equal to zero. The first component of the pair is then assigned to the variable `v1`. In our example, this means `v1` is given a vertex with an in-degree of zero, a vertex that can be next in the topological order.

The purpose of the `for` statement on line 12 is to iterate through all of the edges leaving `v1`. It iterates through tuples in `EDGES` with a first component equal to `v1`, and assigns their second components to the unbound variable `v2`. Each execution of the `for`statement iterates through each matching element at most once, which ensures that each edge will be considered at most once.

Finally, the if statement on line 13 finds in `INDEGREES` a pair whose first component is equal to `v2`, and assigns its second component to `indegree`. The true block of the `if` statement executes only if such a pair can be found, which is always the case in this example if the input is correct. In this case, the real purpose of the if statement is to find the tuple matching the pattern.

One important thing to note about this example is that we match against the second component of the tuples in `INDEGREES` on line 11, and against the first component on line 13. This means that while this algorithm could be implemented using maps, there would have to be two maps corresponding to `INDEGREES`. Another two maps would need to correspond to `EDGES`. Using partially-bound retrieval from sets of tuples, we halve the number of data structures and reduce the amount of update code that needs to be written by the programmer.

2.2 Syntax

The syntax we use for partially-bound retrieval is given in figure 2. It consists of a retrieval clause, used as part of a `while`, `if`, or `for` statement. Each retrieval clause consists of a tuple pattern, an `in` keyword, and an expression that evaluates to a set. During an execution of partially-bound retrieval, this set is known as the *accessed set*.

$$\begin{array}{rcl} retr_statement & ::= & (\texttt{while} \mid \texttt{if} \mid \texttt{for})\ retr_clause : \\ retr_clause & ::= & tuple_pattern\ \texttt{in}\ set_expression \\ tuple_pattern & ::= & (\ component\ (,\ component)^*\ ) \\ component & ::= & expression \mid pattern_variable \end{array}$$

Figure 2. A grammar for partially-bound retrieval statements in our pseudocode language.

A *tuple pattern* consists of one or more comma-separated components. Each component is either an expression where all referenced variables are bound before the retrieval, or a fresh variable that is not bound to anything at the before the retrieval (a *pattern variable*). Pattern variables may not be used in expressions that are part of the same tuple pattern.

Retrieval clauses can be used as part of `while`, `if`, and `for` statements. A retrieval clause replaces the entire condition of a `while` or `if` statement, or the iteration clause of a `for` loop. When used as the iteration clause in a `for` loop, partially-bound retrieval becomes partially-bound iteration, which guarantees that each matching tuple is retrieved exactly once.

2.3 Semantics

The operational semantics of partially-bound retrieval can be given in terms of binding sets. We describe what these binding sets are, and how they can be computed for a given pattern and set. We then show how they can be used to execute a retrieval as part of the `while`, `if`, and `for` statements. Before we can do these, however, we must first define what it means for a pattern to match a tuple, a concept that we've used informally up until this point.

A tuple *matches* a tuple pattern if the tuple and pattern consist of the same number of elements, and if, at the time of the matching, the value of each expression in the pattern is equal to the corresponding component of the tuple. For our purposes, equality here refers to structural equality or some sort of user-defined equality, not equality of object identity. If there are no expressions in the tuple pattern, that pattern trivially matches all tuples of the same length. Since the value of the expression in the pattern may change over the course of program execution as the variables that are used change, a matching is only valid at a particular point in program execution.

Using this definitions, we can define binding sets. A *binding set* is a set containing, for each tuple in the set matching a tuple pattern, a map from the pattern variables to the corresponding components in the tuple. Such a set could be computed by iterating over the accessed set and performing the matching operation, if we were to ever actually compute it. As a binding set involves matching, its value corresponds to a particular execution of a retrieval.

When a binding set is computed, and how it is used, is determined by the statements in which a retrieval occurs. In a `while` or `for` statement, a binding set is computed each time the condition is evaluated. If this binding set is empty, then the condition is false, and no variables are bound. This will cause the a `while` loop to terminate, or an `if` statement's else clause to execute, when present. If the binding set is non-empty, an arbitrarily selected map is taken from it, and the variables found in it are bound to their associated values. Such bindings are in effect until the end of the body of the if or while statement, at which point they become unbound.

In a `while` statement, the fact that the binding set would be recomputed each time the condition is evaluated means that the loop continues as long as a matching element exists in the accessed set. This property makes such a `while` loop useful for accessing a workset. As the `while` loop would recompute the binding set each time through the set, `while` loops are suitable for use with sets that can be changed in the body of the loop.

Partially-bound iteration, as found in a `for` statement, has slightly different semantics. If implemented using binding sets, each execution of a `for` statement would cause a binding set to be computed once, before the first iteration. The iteration then occurs over the contents of the binding set, with the variables in each map being assigned their associated values while executing the body of the iteration.

We impose on partially-bound iteration the restriction that neither the contents of the accessed set nor the values of the expressions in the tuple pattern change over the course of the iteration. These restrictions are not overly burdensome, as they are similar to the prohibition, found in languages such as Java and Python, against changing collections while iterating over them. They also allow us to perform important optimizations, such as the ones given below.

3. Efficient Implementation

As mentioned above, binding sets only exist as a way to give an operational semantics of partially-bound retrieval. For partially-bound retrieval to become a useful language feature, we must find an efficient and practical implementation.

3.1 Local Implementation

Perhaps the most direct implementation of the partially-bound iteration found in a `for` loop is one that intermixes the computation of the elements of the binding set with the use of those elements. Lacking a better name, we call this a *local implementation*. A local implementation of partially bound iteration consists of iterating through each of the elements of the associated set. For each element that is a tuple matching the tuple pattern, the pattern variables are bound to their corresponding elements in the tuple. The body of the iteration is then executed with such bindings. This continues until all elements of the accessed set have been exhausted.

Partially-bound retrieval, when used in `while` and `if`statements, is implemented similarly. For these statements, however, the iteration is performed once for each time the condition is evaluated, and the iteration terminates once a matching tuple is found. If the iteration proceeds to completion without a matching tuple being found, then the partially bound retrieval has failed. In this case, no variables are bound as a result of the retrieval, and the condition is false.

```
1 for v1 in VERTICES:
2     indegree = 0
3     for (v2, tmp0) in EDGES:
4         if tmp0 != v1:
5             continue
6
7         indegree += 1
8
9     INDEGREES add (v1, indegree)
```

Figure 3. Local implementation of initialization.

An example of a local implementation is found in figure 3, showing the code that implements the initialization part (lines 4 through 9) of the example topological sort program in figure 1.

The advantage of a local implementation is its simplicity. If one was asked to implement partially-bound retrieval, by hand and without regard to efficiency of the result, something similar to a local implementation is what would likely arise. This method requires only a constant amount of memory, and requires us to only modify the statement containing the partially bound access, without touching the rest of the program. Its main problem is inefficiency. The cost of a single partially bound retrieval implemented using the local method is proportional to the size of the accessed set, rather than the number of elements in the set that match the tuple pattern. This can lead to asymptotic slowdowns in common algorithms. These slowdowns, while clearly undesirable, are occasionally accepted by programmers when rewriting for performance would unduly complicate the code.

3.2 Bound-Unbound Maps

The inefficiency of a local implementation stems from having to iterate over the entire accessed set on each partially-bound retrieval. To avoid this, we must develop a data structure that allows us to quickly iterate over only the the tuples in a set that match a given pattern.

A data structure that allows us to do this is a *bound-unbound map*. A bound-unbound map is a multimap (explained below) in which the the keys are groups of values corresponding to the expressions in a pattern, while the values associated with a key are groups containing the values corresponding to pattern variables, taken from tuples matching the key. (We expect that groups will be implemented as tuples. Here, we refer to them as groups to avoid confusion with tuples taken from the accessed set.) A bound-unbound map can be created from a given tuple pattern and accessed set, and can exist for the life of that set.

Multimaps are maps in which a single key may be associated with a number of values. When accessed with a key, a multimap returns a set containing all values associated with the key. If no values are associated with the key, then an empty set is returned. An obvious implementation of a multimap is as a map from keys to sets of values. Care must be taken with this implementation to ensure that keys are garbage collected when their associated sets are empty.

The bound-unbound map for a given tuple pattern and accessed set can be constructed in time proportional to the size of the accessed set. This is done by iterating through all elements of the accessed set. Each element of the same length as the pattern has its components divided into two groups, those corresponding to expressions and those corresponding to pattern variables. These groups become the key and value, respectively, of an association that is added to the map. This construction method results in at most one entry being added to the map for each element of the accessed set, ensuring that the size of the bound-unbound map is proportional to the size of the accessed set.

The use of a bound-unbound map allows us to break the evaluation of partially-bound retrieval into three steps. The first step is the construction of the bound-unbound map, given above. The second step evaluates the expressions in the pattern, and uses their values as a key to access the bound-unbound map. Such a lookup can be easily implemented using hashing as an expected constant time operation, and returns a (possibly empty) set giving the values of the unbound variables in tuples matching the pattern. The third and final step is to use the contents of this set in a manner appropriate to the construct being executed. When executing partially-bound iteration, this entails iterating over the contents of the set, an operation that takes time proportional to the size of the set. When only retrieval is necessary, the set is checked for emptiness and an arbitrary element is taken from it, if it is not empty. Both operations are constant time, as is assigning the values from such an element to

```
1  EDGES_ub = MultiMap()
2  for (a, b) in EDGES:
3          EDGES_ub.add(b, a)
4
5  for v1 in VERTICES:
6          indegree = 0
7          for v2 in EDGES_ub.get(v1):
8                  indegree += 1
9
10         INDEGREES add (v1, v2)
```

Figure 4. Bound-unbound map implementation of initialization.

the pattern variables. In all three steps, the only operation that takes time proportional to the size of the accessed set is the construction of the bound-unbound map.

If we recompute the bound-unbound map each time a retrieval is executed, then the asymptotic running time of such an implementation is no better than that of a local implementation. However, an important difference is that a local implementation requires matching to be performed, while computing a bound-unbound map requires only knowledge of the contents of the set. Information about the actual values of the expressions in the pattern is not needed to compute the bound-unbound map. Because significantly less information is used, it's more likely that the computation of a bound-unbound map will be inside an enclosing loop in which the value of the accessed set, and therefore the bound-unbound map, does not change. In this case, we can move the computation of the bound-unbound set to the outside of the enclosing loop. We can move the computation of the bound-unbound map outside of any loop in which the accessed set does not change, potentially reducing the asymptotic running time of the program.

Figure 4 shows what the initialization phase of figure 1 looks like when implemented using a bound-unbound map. The bound-unbound map is kept in the variable `EDGES_ub`, so named because the first component of the pattern is a pattern variable (and therefore unbound), while the second is an expression (and hence bound). To realize this example, we've added multimaps to our language, with methods to add and remove associations, and get the set of values associated with a key. Sets have a method "any", that returns an arbitrary element. The implementation given in figure 3 takes time proportional to the number of vertices times the number of edges in the graph, while this implementation takes only time proportional to the number of edges. This asymptotic improvement is possible because we can move the computation of `EDGES_ub` outside of the iteration over `VERTICES`, as the contents of `EDGES` do not change during the iteration.

3.3 Incremental Update

While previously we've recomputed the contents of the bound-unbound map each time its associated accessed set has changed, this is neither necessary nor desirable. It is possible to incrementally update the contents of a bound-unbound map when changes to its accessed set occur. Doing so allows us to further move the computation of the bound-unbound map to the outside of loops in which all updates to the accessed set have been incrementalized. If all updates to a set can be incrementalized, then the only times at which a recomputation of the bound-unbound map is necessary is when it is initially constructed. If the accessed set starts off empty, then we can exploit the fact that an empty accessed set produces an empty bound-unbound map (regardless of the pattern) to eliminate such recomputation entirely. Recomputation may also still be necessary when it is not possible to guarantee that all updates to the accessed set have been incrementalized, such as when the set is passed into unknown library code.

We update the bound-unbound map by exploiting the property that each entry in the map corresponds to an entry in the accessed set, and each element in the set produces at most one entry in the bound-unbound map. This means that it's simple to create incrementalization rules for the two set update operations, add and remove. When a tuple corresponding to the pattern is added to or removed from the set, the entry in the bound-unbound map representing that tuple is computed in the same manner as is done when the map is recomputed. This entry is then added to or removed from the bound-unbound map, as appropriate. Updating a bound-unbound map in this way is can be done in a constant amount of time, allowing the asymptotic running time of the add and remove operations to remain constant.

If incrementalization is complete, and all recomputation eliminated, then the only operation that takes non-constant time is partially-bound iteration, which takes time proportional to the number of matched elements in the accessed set, the minimum time that operation can take. All other operations (incremental addition, incremental removal, the lookup in the partially-bound map, and retrieving a single element) take expected constant time. As a result, the incrementalized implementation of partially-bound retrieval is asymptotically optimal for a given input program.

3.4 Associating Maps with Sets

One thing we have neglected up until this point is the precise way in which tuple patterns and accessed sets are associated with bound-unbound maps. In this section, we first describe the conditions under which it is possible that a single bound-unbound map can be associated with multiple partially bound tuple patterns. We then discuss static and dynamic approaches for associating bound-unbound maps with tuple patterns.

Above, we detailed how a bound-unbound map is constructed from a partially bound tuple pattern and an associated accessed set. The only information used in this process from the tuple pattern is information that can be determined from the pattern statically, specifically the length of the pattern and which components of the pattern are bound expressions. We can call this information the *bound-unbound pattern*, which for each component of the tuple pattern, contains information about whether that component is an expression or pattern variable. It is possible that a program contains more than one partially-bound retrieval from a given set, such that both retrievals have the same bound-unbound pattern. In these cases, all retrievals can use the same bound-unbound map, thus saving space and time by reducing the number of bound-unbound maps that need to be maintained.

Static Approach. Even with a reduced number of bound-unbound maps, however, there is still the question of how these maps are associated with accessed sets. If we have a finite number of sets, and the sets are always accessed by a single name, then it's easy to do this statically. We simply create bound-unbound maps corresponding to each of the bound-unbound patterns that are used in retrievals from a set, and insert the code to update these maps whenever the set is updated. While this is a simplistic approach, it works well in practice, especially when dealing with modules of programs that do not pass sets to other modules.

Figure 5 gives an example of the static approach in action, showing how the topological sort example given in figure 1 can be translated into working code, with bound-unbound maps statically associated with sets. Since `INDEGREES` is initialized to an empty set, the two bound-unbound maps corresponding to must also be empty, and so there's no need to produce code to compute their initial value. In addition, as all access to `INDEGREES` is done through `INDEGREES_bu` and `INDEGREES_ub`, we were able to

eliminate INDEGREES itself in favor of maintaining only the bound-unbound maps.

Dynamic Approach. For more complex programs, a dynamic approach is called for. In this approach, we associate with each set object certain bound-unbound maps. Instead of attempting to statically determine which maps need to be constructed for which sets, we only associate a map with a set once that map has been used with a partially bound retrieval. While this means that we do need to compute the contents of a bound-unbound map on its first use, this does not harm the asymptotic performance, as the amount of work to do this once is asymptotically less than the amount of work done to add things into the accessed set in the first place, and we only need compute the bound-unbound map once. After its initial computation, a bound-unbound map is incrementally updated by hooks that are called by the add and remove operations on the accessed set.

This dynamic method has the advantage of not requiring much in the way of static analysis, since all updates are performed by hooks, at runtime. This means that it works well in the presence of library code that cannot be changed, and with dynamic languages where static analysis is difficult, or even impossible in the face of code that can change at runtime.

4. Discussion

Maps and Multimaps. It is often the case that we have a set and pattern such that any partially-bound retrieval can match at most one tuple. Such a property is the equivalent of a key constraint on a database table, and is exhibited in the INDEGREES table of our running example, which has one indegree for each vertex. The bound-unbound map for a pattern in which the vertex is bound will contain at most one entry per vertex. In this case, implementing the bound-unbound map as a multimap can be wasteful, each of the set in the multimap will contain at most one element, while requiring significant overhead. In this case, implementing the bound-unbound map as a simple map suffices.

One solution to this problem is to have the programmer declare key constraints on sets of tuples, and to use this information to select the appropriate implementation of a bound-unbound map. While this method is effective in practice, it does add to the burden of the programmer, and can lead to faulty programs if a set ever violates a declared constraint.

Another answer is to implement bound-unbound maps as data structures that can change their representation. Such a data structure would be implemented as a map as long as the key-constraint holds, but would automatically convert its representation to a multimap if the key constraint is ever violated. As the conversion automatically occurs when elements are added to the set, such a data structure is efficient when the key constraint holds, and robust to cases where it doesn't.

Eliminating Updates. One property of bound-unbound maps is that, for a given length of tuple, every tuple in the set has a corresponding entry in the bound-unbound map. It's therefore unnecessary to store the tuple in the set itself, as it can always be reconstructed when the set is accessed. By updating only the bound-unbound maps, and not the set itself, when elements are added to or removed from the set, we can reduce the cost of an add or remove operation.

Nested Tuple Patterns. While the description of our method only deals with single-level tuples, there's nothing that prevents our method from being used to implement partially-bound retrievals involving nested tuple patterns. The only issue here is that it is necessary to take into account the nesting when finding the components that correspond to expressions and pattern variables, when building or updating the bound-unbound map.

Extension to Lists. While throughout this paper we have been discussing partially-bound retrieval of tuples contained in sets, our method is not limited to sets. We have also developed a data structure that allows us to efficiently perform partially-bound retrieval on lists, subject to limitations on the update operations that are performed on the accessed list. The limitation we impose is that addition and removal from the accessed list must occur at the head or tail of the list, and not at arbitrary points in the middle of the list.

The data structure we use to implement this is an ordered bound-unbound map. This is a ordered multimap in which entries can be added to the start or the end of the map, and are returned as a list in the order in which they appear in the multimap.

An ordered multimap can be implemented as a map from keys to lists, in the same way that a normal multimap can be implemented as a map from keys to sets. The limitation imposed above makes it possible to determine if the addition or removal of an entry in the bound-unbound map should occur at the start or the end of a list. If we allowed addition of an element to occur at an arbitrary point in an accessed list, it would be impossible to determine, in constant time, where in the associated list in the bound-unbound map to add the new entry. When subject to this limitation, however, addition and remove can be done incrementally in constant time, making partially-bound retrieval that accesses a list as efficient as that which accesses a set.

Alternative Syntax. The syntax proposed in this paper is by no means the only syntax that is possible for partially-bound retrieval. There are alternative syntaxes that are potentially more appropriate for specific languages.

When used with a dynamic language, it may make sense to add a keyword or other syntax element that indicates which components of a tuple pattern are bound expressions. This can reduce confusion in languages where variable bindings can leave a block. It may be necessary in languages (such as Python) where our proposed syntax is already legal, but has a different meaning.

Alternatively, one may indicate which components are pattern variables. This may be desirable in languages that require type annotations, as the type annotation can serve both to indicate that a variable is unbound, and to declare its type when it becomes bound.

5. Systems

To gain experience with partially-bound retrieval, we have developed two systems that allow programs to be written using it. Our first system, takes as input programs written in a pseudocode-like language, and produces efficient C++ code. This system has been successfully used in the implementation of a number of algorithms. At the same time, it suffers from a number of limitations. Instead of extending the pseudocode language to address these, we have chosen to develop a second tool that adds partially-bound retrieval to an existing programming language. Our second tool takes as input a program written in a variant of Python extended with the three partially-bound retrieval statements, and outputs efficient standard Python code. In this section, we discuss the history of these systems, the differences in the generated implementations, and the advantages and disadvantages of each.

5.1 Pseudcode Language

Our pseudocode-like language, named PATTON, was originally written to assist in the implementation of parametric regular path queries, as described in [10] and the experiments section below. The inspiration for partially-bound retrieval originally came from the pseudocode found in that paper. This system was developed

with two goals in mind: to allow us to efficiently try variants of the algorithms by translating pseudocode to C++, and to allow us to compare implementations of bound-unbound maps. Specifically, we compared based representations (using records and linked lists, as given in [16, 14, 15, 9] and [3]) with hash-table representations. While we originally added partially-bound retrieval to minimize the differences between an algorithm's implementation and its pseudocode, we quickly began to appreciate it as a programming language construct in its own right.

The pseudocode language is a simple one, but one that allows a number of algorithms to be easily expressed. Along with all three partially-bound retrieval statements, it includes statements for reading input and writing output, and for adding elements to and removing elements from sets. As data it supports sets, tuples, strings, and integers. The support for the latter two is limited to the equality comparisons needed for partially-bound retrieval. This is because the construction of new values through mathematical or string operations can interfere with based representations. The pseudocode language supports two kinds of variables: normal variables that can refer to strings, integers or tuples of such values, and set variables that can only refer to sets. Each set variable refers to a single set, and there is no way to create sets besides declaring set variables. As a result, programs written in this language support only a finite number of sets, all known statically. This, in conjunction with add and remove statements, makes it easy to insert code to maintain the bound-unbound maps, without needing complicated static analysis. An additional statement in the language allows specification of key constraints, which are used to select between map and multimap implementations of the bound-unbound maps.

Our pseudocode language proved to be a success, both on its own and when used as a target to simplify code generation. As we will discuss in the experiments section, we were able to use it to implement parametric regular path queries and Datalog rules. Although not discussed in this paper, we also used it to implement relational calculus queries. In all three cases, partially-bound retrieval reduced significantly the amount of effort needed to produce efficient code.

As we moved into the area of security policies, however, some issues with our psuedocode language became apparent. A lack of support for function or procedure calls makes it impossible to implement security policy frameworks, such as role-based access control. While it would be possible to extend this language to include these features, we felt that it would be a more useful to add partially-bound retrieval to a popular language. This decision led to the creation of our second system.

5.2 Python Preprocessor

We then implemented partially-bound retrieval as a preprocessor that takes as input Python programs augmented with the three partially-bound retrieval statements, and outputs efficient standard Python code. Python was chosen as a source and target language because of its built-in support for tuple construction and set iteration, both of which are syntactically similar to what's used in our partially-bound constructs. Indeed, this similarity is to such an extent that we chose to give the added partially-bound constructs different keywords, to prevent code that uses our extensions from being run by Python itself.

Our preprocessor generates code that uses callbacks to update the bound-unbound maps, with the maps themselves being stored, when present, in fields on the set objects themselves. This allows the modifications to the underlying code to be minimized. The Set class is modified to automatically invoke callbacks when elements are added to or removed from it, a modification that only needs to be done once, regardless of the number of partially-bound retrieval statements and the number of add and remove call sites in the program. All of the other modifications to the program are confined to the immediate vicinity of the partially-bound statements. We do not need to find or modify statements that add and remove elements of the set, as such operations are detected using callbacks. This means that the preprocessor does not require a global static analysis, making it suitable for programs that may use arbitrary library code, and for programs that modify themselves (by constructing and evaluating code) at runtime. The code generated by our preprocessor represents all bound-unbound maps with a data structure that changes from a map to a multimap when necessary.

In the experiments section, we present the results of applying the preprocessor to the running example of topological sort, to graph reachability, and to role-based access control. The experiments revealed a number of optimizations that can be adde, such as eliminating the construction of 1-element tuples, and keeping references directly to the bound-unbound maps. At the same time, the experiments show that the preprocessor is able to generate asymptotically efficient implementations of Python programs written using partially-bound retrieval.

6. Experiments and Applications

In this section, we present the results of a number of experiments we conducted on programs written using partially-bound retrieval. These experiments were conducted over a variety of problem domains: graph algorithms and queries, program analysis, and security policy frameworks. We conducted these experiments to evaluate the advantages of having partially-bound retrieval as a language construct, and to confirm that the predicted efficiency of bound-unbound maps can be actually achieved. This section includes experiments done with both of our tools, and a mix of experiments that were performed as part of research into other areas, and experiments that are original to this paper. The latter allows us to demonstrate on well-known examples the effectiveness of partially-bound retrieval, while the former shows its applicability to a range of problems.

In this section, when lines of code are given, the value provided is the number of lines of code that solve the problem, excluding comments and blank lines. This does not include the size of the library code included by an implementation, nor does it include the size of the test harnesses used to collect statistics. Unless otherwise noted, performance measurements were collected on an AMD Sempron 3100+, running at 1.8 GHz. When preprocessor-generated Python programs were run, Python 2.3.5 was used. The measurements are the average of five runs on the same input data set.

6.1 Topological Sort

The first experiment we perform is on the running example of topological sort. We created four Python implementations of the topological sort algorithm given in figure 1. Two of these implementations are the local and bound-unbound map implementations automatically created by the current version of the preprocessor. A third implementation consists of the bound-unbound map implementation, hand optimized to eliminate unnecessary tuple construction and store bound-unbound maps in local variables to prevent repeated field lookups. This version represents optimizations that are planned for a future version of the preprocessor, but are not yet implemented. The final implementation was a hand implementation of the version of the program given in figure 5. This was implemented by hand so we could compare a static implementation with the dynamic implementation generated by the preprocessor, under similar conditions.

Table 5.2 shows the results of these experiments. The first thing to note is the expansion of the program sizes. The topological sort algorithm, written in Python with partially-bound statements, is 13

implementation	size	Running time for number of edges				
		1,000	**2,000**	**3,000**	**4,000**	**5,000**
local	30	678 (19.94)	3,012 (43.03)	7,190 (66.57)	13,580 (97.00)	23,750 (132.68)
dynamic b-u map	63	56 (1.65)	116 (1.66)	176 (1.63)	234 (1.67)	296 (1.65)
optimized dynamic b-u map	63	34 (1.00)	74 (1.06)	110 (1.02)	150 (1.07)	185 (1.03)
static b-u map	29	34 (1.00)	70 (1.00)	108 (1.00)	140 (1.00)	179 (1.00)

Table 1. Program size in lines and running times in milliseconds for implementations of topological sort. The numbers in parenthesis are running times relative to the static implementation.

implementation	size	Running time for number of edges			
		50,000	**100,000**	**150,000**	**200,000**
dynamic b-u map	24	790 ms (0.98)	1,604 ms (1.00)	2,396 ms (0.99)	3,218 ms (0.99)
static b-u map	16	803 ms (1.00)	1,607 ms (1.00)	2,414 ms (1.00)	3,240 ms (1.00)

Table 2. Program size in lines and running times in milliseconds for implementations of graph reachability.

```
1  read VERTICES, EDGES
2
3  INDEGREES_bu = MultiMap()
4  INDEGREES_ub = MultiMap()
5
6  EDGES_bu = MultiMap()
7  EDGES_ub = MultiMap()
8
9  for (a, b) in EDGES:
10     EDGES_bu.add(a, b)
11     EDGES_ub.add(b, a)
12
13 for v1 in VERTICES:
14     indegree = 0
15     for v2 in EDGES_ub.get(v1):
16         indegree += 1
17
18     INDEGREES_bu.add(v1, indegree)
19     INDEGREES_ub.add(indegree, v1)
20
21 while True:
22     tmp0 = INDEGREES_ub.get(0)
23     if not tmp0:
24         break
25
26     v1 = tmp0.any()
27
28     for v2 in EDGES_bu.get(v1):
29         tmp1 = INDEGREES_bu.get(v2)
30
31         if tmp1:
32             indegree = tmp1.any()
33
34             INDEGREES_bu.remove(v2, indegree)
35             INDEGREES_ub.remove(indegree, v2)
36             INDEGREES_bu.add(v2, indegree - 1)
37             INDEGREES_ub.add(indegree - 1, v2)
38
39     INDEGREES_bu.remove(v1, 0)
40     INDEGREES_ub.remove(0, v1)
41     print v1
```

Figure 5. Topological sort, implemented using static association of bound-unbound maps.

```
1  read start, EDGES
2
3  WORKSET = Set()
4  RESULT = Set()
5
6  WORKSET add start
7  RESULT add start
8
9  while v1 in WORKSET:
10     for (v1, v2) in EDGES:
11         if v2 not in RESULT:
12             WORKSET add v2
13             RESULT add v2
14
15     WORKSET remove v1
16
17 print RESULT
```

Figure 6. Graph Reachability, written using partially-bound retrieval.

lines long. The implementations varied in length from 29 to 63 lines long, long, so using partially-bound retrieval leads to a program that is less than half the size.

To evaluate the performance of the generated code, we ran the implementations on topological sort problems of varying size. In all cases, the input data consisted of a linear chain of vertices, such that there is one unique topological sort.

The first thing to note is that the local implementation of the program is asymptotically slower than the other three versions, as predicted. The unoptimized dynamic version, while still linear, is much slower than the other two versions, while the static version is slightly faster than the optimized dynamic one. We attribute this difference to the increased indirection required in the dynamic version, and while we are working on decreasing this indirection penalty, we recommend using a static implementation when possible.

6.2 Reachability

To attempt to determine the cause of the difference in the running times of the static and dynamic versions of the program, we used the preprocessor to generate implementations of the graph reachability algorithm given in figure 6. This algorithm contains two partially-bound retrievals, one from `WORKSET` and one from `EDGES`. The retrieval from `WORKSET` does not require a bound-unbound map, as no bound expressions are used in it. The only bound-

unbound map is the one used for retrievals from EDGES. As EDGES is not updated in the loop, this bound-unbound map is not changed from when it is created, allowing us to consider the performance of the queries themselves without needing to also consider the cost of updating bound-unbound maps.

We implemented two versions of graph reachability. The first version was a dynamic bound-unbound map implementation automatically generated from a Python translation of the code given in figure 6. We did not create a hand-optimization of this version, as the optimizations only pertain to bound-unbound map update. For comparison purposes, we also created by hand a static version of the graph reachability, similar to what would be generated from our psuedocode language, except in Python instead of C++.

The results are given in table 5.2. The first thing to note is that the dynamic implementation doubles the size of the initial 12-line program, while the static implementation adds a mere four lines. This is because the dynamic implementation still has to include the update code, even if it is never called, while the static version can eliminate it entirely. This has no effect on the running time, as the dynamic implementation is marginally faster than the static one. Achieving similar results on a program that does not contain any updates suggests that it is the update functions that differ in performance between the static and dynamic versions of the program.

6.3 RBAC — Role-Based Access Control

An example of a realistic system that benefits from partially-bound retrieval is role-based access control. RBAC controls access by assigning permissions to perform operations on objects to roles, and then assigning users to those roles. When a user activates a role in a session, that session has all the permissions of the role. This simplifies the management of permissions in systems with many users, objects, and operations.

In the ANSI standard for RBAC [6, 1], RBAC is specified using eight sets and four maps from values to sets. Five of the sets are uninteresting from our perspective, containing only values of a given type, and not tuples. The remaining three are sets of pairs: PRMS contains all possible permissions in the system as object-operation pairs, PA relates roles with the permissions the role has, and UA assigns users to roles. Of the maps used in the core RBAC specification, only user_sessions and session_roles are fundamental, mapping a user to his sessions and a session to its roles, respectively. The other two, assigned_permissions and assigned_users, duplicate information contained in PA and UA, respectively.

To help evaluate partially-bound retrieval, we translated the administrative and system functions of core RBAC from the variant of Z used in [6] to the dialect of Python that our preprocessor can process. As part of the conversion, we eliminated the two redundant maps, and turned the other two maps into sets of pairs. We were able to eliminate 10 of the 30 map and set updates found in the 13 functions we translated. The resulting translation consists of 85 lines of python, comprised of 36 assertions, 19 set updates, 13 function definitions, 8 partially-bound retrievals, 3 function calls, 2 set membership tests, 2 set iterations, and 2 returns.

When translated by our preprocessor into a dynamic bound-unbound map implementation, the size of the program swelled to 211 lines. Inspection of the generated code showed that the preprocesor correctly generated bound-unbound maps corresponding to assigned_permissions and assigned_users. By using an partially-bound retrieval, we were able to eliminate two maps and a third of the update operations from the specification of RBAC.

Figure 7 demonstrate how an efficient implementation of partially-bound retrieval improves asymptotically the running time of our program. It shows the time it takes to perform 100000 check_access

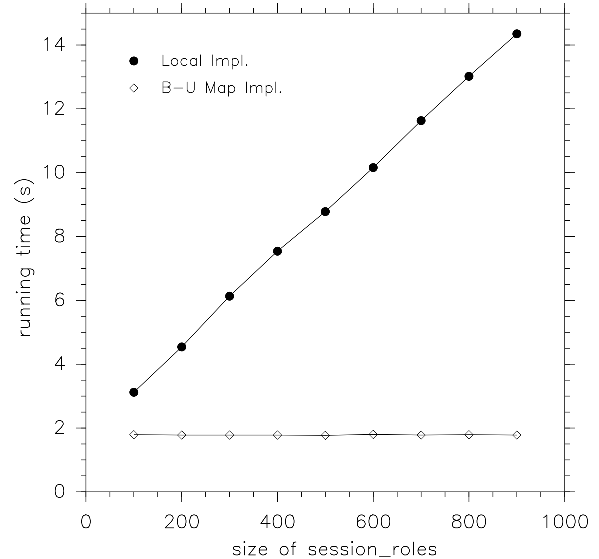

Figure 7. Running time of 100000 RBAC check_access operations.

operations while varying the size of session_roles, the set that maps a session to the roles used by that session. In this graph, the number of roles per session is fixed at 10, while the number of sessions increases. The local implementation takes running time proportional to the total number of session-role pairs in the system, while the dynamic bound-unbound map implementation remains constant, scaling only with the number of roles per session. This beats the asymptotic performance of a straightforward implementation of the Z program, which takes time proportional to the number of roles in the system, of which the roles per session is a subset. This demonstrates that partially-bound retrieval not only simplifies the implementation of RBAC, it also allows us to generate an implementation that is an asymptotic improvement.

6.4 Other Applications

In addition to the experiments performed above, partially-bound retrieval has been successfully applied to a range of problems, each complex enough to have merited its own paper. Here, we discuss how partially-bound retrieval was used in the experiments in those papers. In contrast with the experiments given above, our focus here is on describing how partially-bound retrieval simplified the creation of those implementations, rather than on performance measurements.

Parametric Regular Path Queries [10] match a regular-expression-like pattern containing variables against paths in a graph containing labels. An existential regular path query returns the set of all vertex-substitution pairs such that there exists a path from the start vertex to the returned vertex where the labels on that path match the pattern, after the substitution has been applied to the pattern. This has a number of applications to program analysis, as simple queries can find uses of uninitialized variables, violations of locking disciplines, and other properties of the program.

Partially-bound retrieval was used in the implementation of multiple algorithms that perform parametric regular path queries. Indeed, it was the need to efficiently generate implementations of the algorithms in [10] that lead to the creation of our pseudocode language, and the pseudocode found in that paper that inspired the design of that language. By automatically generating extremely efficient C++ code that uses based representations of the bound-

unbound maps, we significantly reduced the effort required to implement variants of the parametric regular path query algorithms. This allowed us to give experimental performance results for a number of variants, enabling us to give guidance as to when each variant should be used.

When written using partially-bound retrieval, various algorithms for performing parametric regular queries range from 19 to 34 lines of code, counted as described in this section, including 3 partially-bound retrievals. When our psuedocode language is translated into C++, the code size expands substantially. The 19 line example was translated into 696 lines of C++, the 34 line variant to 833 lines. The generated code is fast, processing over 400,000 worklist entries per second. The running time of the implementations scales only with worklist size, further showing how bound-unbound maps can be used to implement partially-bound retrieval in an asymptotically optimal manner.

Datalog Rules. Partially-bound retrieval also shows promise as a construct in intermediate languages that tools can target. [11] describes a method that transforms a set of Datalog rules into efficient low-level implementations with guaranteed time and space complexities, avoiding dependency on a potentially large interpreter. One implementation of this method uses our psuedocde language as an intermediate language, to simplify the code generation process. It first generates code in our psuedocode language language, and then that code is translated to C++. This two-step process simplifies the tool's code generator. Another implementation that generates C directly consists of 327 lines of Python, and uses a library of 2,000 lines of C code. The generator that targets our language was written in two days, and consists of only 114 lines of Python code, and does not require a custom library.

As an example of the effectiveness of this approach, we give two examples. A transitive closure algorithm consisting of two Datalog rules was translated into 27 lines of pseudocode, which in turn became 595 lines of efficient C++. A pointer analysis algorithm was translated into 93 lines of pseudocode, and 1,944 lines of C++.

7. Related Work and Conclusion

Partially-bound retrieval is related to work in a number of fields of computer science. Since it involves querying data, it's related to databases. As a programming language construct, it's related to programming languages. It's also related to the tuple spaces used in distributed programming, and to indexing in Prolog. Lastly, our work can be considered to be in the area of data structure selection.

When working with sets of tuples, an obvious comparison is with relational databases. Partially-bound retrieval can be considered a restricted form of the select operation in relational algebra. By focusing on only one operation, we gain a number of advantages over relational databases, which support more complicated queries. One advantage is that our query syntax is much more succinct than that of embedded SQL, and fits more naturally into programming languages. A second advantage is that we do not require a RDBMS, with the expense (in code size, running time, and occasionally currency) that implies. Finally, because of the low overhead of performing a partially-bound retrieval, it can be used in places where a database query would not be, such as the inner loop of the graph reachability example.

That said, there's much to be learned from relational databases. This paper leaves the query optimization possible with relational algebra up to the user. It is possible to automate some such optimizations, even without the information about set size that is known to a database query optimizer. In [13], it is shown how some relational queries can be translated into efficient code that uses partially-bound retrieval. A second issue is that we maintain bound-unbound maps, which are in many ways equivalent to indices in databases, for every partially-bound retrieval in the program. An area of research in databases is automatically determining which indices most benefit query performance [5, 8]. We may use similar methods to determine which bound-unbound maps most improve program performance. On a memory-limited system, when trading speed for memory we want to ensure that we make the best trade possible.

Moving on to programming languages, we should note that quite a few languages have support for tuple patterns. These include the ML family of languages, where there has been some work done on optimizing pattern matching [7], and dynamic languages such as Python and Perl. However, these constructs do not allow non-constant expressions to be included as part of the tuple pattern, and hence differ from partially bound retrieval.

The languages that we have found that contain the closest analog to partially-bound retrieval are Linda [4] and its successors, such as TSpaces [12]. They provide a simple model for distributed computing by providing shared tuple spaces, which are sets of tuples that can be distributed among multiple computers. Tuples in a space can be matched by providing the values of some of the fields, as in our tuple patterns. There is a difference in focus between Linda systems, which support distributed retrieval from a relatively small number of tuple spaces, and partially-bound retrieval, which provides fast centralized retrieval from a potentially large number of sets. Descriptions of Linda-like systems (such as those in [18]) focus primarily on retrieving a tuple from an appropriate distributed node, and do not address the problem of finding a tuple once that system has been found. In this way, we compliment the work they have done.

One strategy that Prolog implementations use is to index facts to eliminate impossible unifications. By replacing sets of tuples with facts, and matching with unification, our bound-unbound maps can be seen to accomplish a similar purpose as these Prolog indexes. More specifically, by indexing on all expressions, our method is similar to multiple argument indexing (also called multiple position indexing), as found in [17, 2]. Many Prolog systems do not support multiple argument indexing, instead indexing on only a single argument per fact. Systems that do require the user to declare indices explicitly, whereas our method can compute them automatically, even without analyzing the entire program.

Finally, our work can be considered a case of data structure selection, and as such owes much to the pioneering data structure selection work performed with the SETL programming language [16, 14, 15, 9, 3]. Our work extends theirs by providing support for sets of arbitrary-length tuples, instead of sets of pairs, and by providing a syntax that allows us to use expressions to match any component of a tuple, rather than just the first component of a pair.

Partially-bound retrieval is a language construct that raises the level of programming by allowing sets of tuples to be easily accessed. It makes many programs easier to write, by freeing the programmer from having to explicitly maintain index maps. The resulting programs are simpler and easier to understand. We have developed a method to efficiently implement partially-bound retrieval, and shown through experiments that the method produces efficient code, without needing complex global analyses that make writing tools difficult. For all these reasons, we believe that partially-bound retrieval is a worthy addition to a programming or specification language.

Acknowledgments

We thank Scott Stoller for his input on a draft of this paper.

References

[1] American National Standards Institute, Inc. Role-based access

control. ANSI INCITS 395-2004.

[2] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López, and G. Puebla. The ciao prolog system. Reference Manual. The Ciao System Documentation Series. TR CLIP3/97.1.10. School of Computer Science, Technical University of Madrid (UPM), August 2004.

[3] J. Cai, P. Facon, F. Henglein, R. Paige, and E. Schonberg. Type analysis and data structure selection. In *Constructing Programs From Specifications*, pages 126–164. North-Holland, 1991.

[4] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.

[5] S. Choenni, H. M. Blanken, and T. Chang. Index selection in relational databases. In *International Conference on Computing and Information*, pages 491–496, 1993.

[6] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed nist standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274, 2001.

[7] F. L. Fessant and L. Maranget. Optimizing pattern matching. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 26–37, New York, NY, USA, 2001. ACM Press.

[8] M. R. Frank, E. Omiecinski, and S. B. Navathe. Adaptive and automated index selection in RDBMS. In *Extending Database Technology*, pages 277–292, 1992.

[9] S. M. Freudenberger, J. T. Schwartz, and M. Sharir. Experience with the SETL optimizer. *ACM Trans. Program. Lang. Syst.*, 5(1):26–45, 1983.

[10] Y. A. Liu, T. Rothamel, F. Yu, S. D. Stoller, and N. Hu. Parametric regular path queries. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 219–230, New York, NY, USA, 2004. ACM Press.

[11] 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.

[12] S. W. McLaughry and P. Wycko. T spaces: The next wave. In *HICSS '99: Proceedings of the Thirty-second Annual Hawaii International Conference on System Sciences-Volume 8*, page 8037, Washington, DC, USA, 1999. IEEE Computer Society.

[13] G. Priyalakshmi. Generating efficient programs for solving relational database queries. Master's thesis, Stony Brook University, August 2004.

[14] E. Schonberg, J. T. Schwartz, and M. Sharir. Automatic data structure selection in SETL. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 197–210. ACM Press, 1979.

[15] E. Schonberg, J. T. Schwartz, and M. Sharir. An automatic technique for selection of data representations in SETL programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(2):126–143, 1981.

[16] J. T. Schwartz. Automatic data structure choice in a language of very high level. In *Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 36–40. ACM Press, 1975.

[17] T. L. Swift. *Evaluation of Normal Logic Programs*. PhD thesis, Stony Brook University, December 1994.

[18] G. Wilson. Linda-like systems and their implementation. Technical Report 91-13, Edinburgh Parallel Computing Centre, June 1991.