Generating specialized rules and programs for demand-driven analysis[⋆]

K. Tuncay Tekle, Katia Hristova, Yanhong A. Liu

SUNY Stony Brook, NY, USA
{tuncay,katia,liu}@cs.sunysb.edu

Abstract. Many complex analysis problems can be most clearly and easily specified as logic rules and queries, where rules specify how given facts can be combined to infer new facts, and queries select facts of interest to the analysis problem at hand. However, it has been extremely challenging to obtain efficient implementations from logic rules and to understand their time and space complexities, especially for on-demand analysis driven by queries.

This paper describes a powerful method for generating specialized rules and programs for demand-driven analysis from Datalog rules and queries, and further for providing time and space complexity guarantees. The method combines recursion conversion with specialization of rules and then uses a method for program generation and complexity calculation from rules. We compare carefully with the best prior methods by examining many variants of rules and queries for the same graph reachability problems, and show the application of our method in implementing graph query languages in general.

1 Introduction

Many complex analysis problems can be most effectively and easily described using a declarative language. The declarative specification makes it easy to understand the nature of the problem, without being distracted by implementation details. One way of writing a declarative specification is to write logic rules and queries.

Logic rules specify how given facts in a problem setting can be combined to infer new facts. For example, for program analysis, definitions of flow and dependence relations can be specified as rules; for model checking, definitions of system behaviors can be specified as rules; and for system security, access control policies can be specified as rules.

Once the specification of a problem is given by logic rules, queries can be used to select facts of interest to the analysis problem at hand. For program analysis, flow and dependence information involving particular program points of interest can be specified as queries; for model checking the properties to be

[⋆] This work was supported in part by NSF under grants CCR-0306399 and CCF-0613913.

checked can be specified as queries; and for system security, checking access to resources by users can be specified as queries. Queries can be used to filter the facts inferred by the rules, and moreover be a guide in the inference of the facts of interest. We use *on-demand analysis* to refer to an analysis that is expressed by a query, querying over facts that can be inferred from the rules.

Even when logic rules and queries are implemented in, say, a Prolog system, evaluated using various existing methods, or rewritten using methods such as magic set transformations to allow more efficient evaluation, such implementation is typically for fast prototyping. Furthermore, the running times of implementations using these methods can vary dramatically depending on the order of rules and the orders of hypotheses in rules, and even less is known about the space usage. Developing efficient implementations for answering queries on-demand for any given rules and queries with time and space guarantees is a nontrivial, recurring task.

This paper describes a powerful method for generating specialized rules and programs for demand-driven analysis from Datalog rules and queries, and for providing time and space complexity guarantees. Datalog [6] is an important logic-based language for specifying rules. Especially in recent years, Datalog-like rules have been used increasingly for expressing complex analysis problems, for example, pointer analysis and program analysis in general [16], model checking push-down systems [11], role-based access control [3], trust management [13], and information flow analysis [12]. Datalog-based languages are also important in graph queries [8, 20] and semantic web applications [9] in general.

Given a set of rules and a kind of query, i.e., a query predicate with indications of which arguments will be bound, our method generates a set of rules and a program that is specialized for the kind of query, and produces complexity formulas for the time and space complexities of the generated program. The generated program for the specialized rules can take any set of given facts and any values of the bound parameters of the query predicate, and return the query result with the calculated time and space complexities. The method combines three transformation steps.

Recursion conversion: transforms recursive rules into appropriate left or right linear recursive forms based on the kinds of queries, so that the connection between the queries and given facts can be established efficiently. Queries can then be answered equally efficiently for equivalent but slightly different recursive rules, which could otherwise differ asymptotically in running times. **Specialization:** specializes the transformed rules with respect to the kinds of query, so that bound parameters of the query predicate are used to restrict possible instantiations of the rules as much as possible. This is a drastically simplified form of partial evaluation [17] and may yield asymptotic improvements in running time. **Program generation and complexity calculation:** transforms specialized rules into efficient algorithms and data structures for the given analysis problem, and calculates the time and space complexities of the generated program. This uses the method developed previously [19] for bottom-up evaluation of Datalog rules.

The main contributions of this paper are not in each of the three transformation steps, but in their combination to produce efficient specialized rules and programs for on-demand analysis and to provide complexity guarantees. No less important is the evaluation of the method in precise comparison with the best prior methods whose effect on complexities are well-known to be difficult to understand. We also show the application of our method on graph query languages.

2 Language and cost model

We describe the Datalog language for defining rules and queries and give our cost model.

Datalog rules. *Datalog* is a declarative language for defining facts and rules that are used to infer new facts from given ones. A Datalog program is a finite set of clauses of the form: $p_1(x_{11}, ..., x_{1a_1}), ..., \; p_h(x_{h1}, ..., x_{ha_h}) \rightarrow p(x_1, ..., x_a).$, where h is a natural number, each p_i (respectively p) is a relation of a_i (respectively a) arguments, called a *predicate*, each x_{ij} and x_k is either a constant or a variable, and variables in x_k's must be a subset of the variables in x_{ij}'s. A predicate with arguments is called an *atom*. If $h = 0$, then there are no p_i's or x_{ij}'s, and x_k's must be constants, in which case $p(x_1, ..., x_a)$ is called a *fact*. An atom on the left hand side of a rule is called a *hypothesis*, and the atom on the right hand side is called the *conclusion*. Semantically, a rule of the form above says that if there is a substitution of variables in the rule with constants such that all of the hypotheses instantiated using the substitution are facts, then the instantiated conclusion is a fact.

Datalog queries. A query for a set of Datalog rules and facts is of the form $q(y_1, .., y_n)$?, where q is a predicate of n arguments. The meaning is to return all tuples of q that are given or can be inferred based on the rules, restricted by the constants in y_i's, if any. We denote constants by `a`,`b`,`c`, and variables by `x`,`y`,`z`.

Example. A canonical example of a Datalog program is the transitive closure of a relation, which can be expressed with two rules. We can think of the relation as the edges of a graph, and paths between any vertices as the set of transitive closure, then the specifications in Datalog would be the following:

```
Doubly recursive: edge(x,y) → path(x,y).
                  path(x,z), path(z,y) → path(x,y).        (1)

 Right recursive: edge(x,y) → path(x,y).
                  edge(x,z), path(z,y) → path(x,y).        (2)

  Left recursive: edge(x,y) → path(x,y).
                  path(x,z), edge(z,y) → path(x,y).        (3)
```

These three programs can be proven by induction to infer the same `path` facts. The right- and left-recursive versions of the transitive closure concatenate edges from the vertex on the left, respectively right, with paths to the vertex on the right, respectively left. They are *linear* programs, i.e., there is at most one hypothesis in each rule that is recursive with its conclusion, however the doubly recursive program is not.

For these programs, there are 4 possible queries: `path(x,y)?` returns all pairs of vertices that have a path between them. `path(a,y)?` returns all vertices that are reachable from `a`. `path(x,b)?` returns all vertices that can reach `b`. `path(a,b)?` returns whether `b` is reachable from `a`.

Cost model. We use the cost model that resulted from the method in [19], which states the following: For any Datalog rule, the evaluation takes time proportional to the number of combinations of facts that make all hypotheses true. All input facts have to be read in, so the number of input facts must be added to the complexity. For example, for transitive closure, this is the number of edges. We use the following notation for complexity analysis. For queries regarding transitive closure, if the first argument is bound, it is denoted by a, and if the second argument is bound, it is denoted by b.

- V : number of vertices, P: number of paths, E: number of edges.
- $E(a)$: number of edges that are on any path from a to any vertex.
  $IE(a)$: number of edges that are on any path from any vertex to a.
- $o(a)$: outdegree of a, o: maximum outdegree of vertices.
  $i(a)$: indegree of a, i: maximum indegree of vertices.
- $R(a)$: number of vertices reachable from a, R: maximum number of vertices reachable from any vertex.
  $IR(a)$: number of vertices that reach a.

As an example, consider the program in (2). The evaluation of the first rule takes time $O(E)$, since all edges make the single hypothesis true. The second rule has two hypotheses, say we take all edges for the first hypothesis, then `z` becomes bound for the `path` predicate and the number of values that `y` can take is the maximum number of vertices reachable from any node. Therefore, a bound on the running time for this program is $O(E \times R)$.

3 Specialization and complexity of specialized programs

Constants in the arguments of a query are called *static inputs*. For example, in the query `path(a,x)?`, `a` is a static input. Specialization uses static inputs to restrict the number of inferred facts by transforming the rules. Program specialization is also known as partial evaluation, and has been studied in logic programming [17], where it is sometimes called partial deduction.

Specialization for a set of Datalog rules S, and a query Q is obtaining another set of rules S' and a query Q' that satisfy the following: Every fact inferred as an answer to Q' during the evaluation of S' is a *projection* of a fact inferred as an answer to Q during the evaluation of S, where a *projection* of a fact is a selection of zero or more arguments from that fact up to a renaming of the predicate.

As an example, consider S being (3), and `path(a,y)?` being Q. Let S' be:

$$\begin{array}{l}\texttt{edge(a,y)} \rightarrow \texttt{path}_{\texttt{1a}}\texttt{(y).}\\ \texttt{path}_{\texttt{1a}}\texttt{(z), edge(z,y)} \rightarrow \texttt{path}_{\texttt{1a}}\texttt{(y).}\end{array} \tag{4}$$

and Q' be $\texttt{path}_{\texttt{1a}}\texttt{(y)?}$. The original query finds all vertices that are reachable from `a` by selecting the `path` facts whose first argument is `a`. Q' and S' do exactly that, and the answers to Q' are the vertices that are reachable from `a`. By inserting `a` as the first argument in the answers of Q', one trivially reconstructs the answers of Q.

To describe specialization, we need to define substitution. For a set of rules S, we denote the set of hypotheses of all rules by $\mathtt{h(}S\mathtt{)}$. We denote the conclusion of a rule r by $\mathtt{c(}r\mathtt{)}$. A *substitution* is a map from variables to constants. A substitution θ applied to a rule r, denoted $r\theta$, replaces the variables in r with constants according to θ. We say that an atom a' is an *instance* of an atom a if there is a substitution θ such that $a\theta = a'$; in case such a substitution exists, it is denoted $\mathtt{subst(}a, a'\mathtt{)}$.

We specialize a set of Datalog rules with respect to a query via the fixpoint of a function f, which takes a set S of rules and a set A of atoms, and returns both of them with new elements added. At each step of computation, if there is an atom a in A, and a rule r in S for which a is an instance of the conclusion of r, then a new rule r', which is r updated with the substitution that makes a and the conclusion of r identical, is added to S and all hypotheses of r' are added to A. That is:

$$f(\langle S, A\rangle) = \langle S\cup S', A\cup \mathtt{h}(S')\rangle \text{ where } S' = \{r\theta | a \in A, r \in R, \theta = \mathtt{subst}(\mathtt{c}(r), a) \neq \mathit{undef}\}.$$

Given a set of rules S, and a query Q, specialization computes the fixpoint of $f(S, Q)$ and returns the first component of the output pair as the desired set of specialized rules. The output of the function also has the original rules in the specialized set, therefore we need to remove them if they are not needed for the evaluation of the specialized query. An original rule r in the output is not needed, unless a hypothesis of a specialized rule is identical to the conclusion of r up to variable renaming. Once these rules are removed, we rewrite all atoms that have constant arguments to remove constants, and assign names based on the original predicate names and the places and values of bound arguments. We only rewrite the atoms whose predicates appear in the conclusion of some rule.

Specialization of (3) with respect to the query `path(a,y)?` yields:

$$\begin{array}{l} \texttt{edge(a,y)} \rightarrow \texttt{path}_{\texttt{1a}}\texttt{(y).} \\ \texttt{path}_{\texttt{1a}}\texttt{(z), edge(z,y)} \rightarrow \texttt{path}_{\texttt{1a}}\texttt{(y).} \end{array} \tag{5}$$

and the query $\texttt{path}_{\texttt{1a}}\texttt{(y)?}$. Given the same query, if one applies specialization to (1), the original unspecialized rules remain since the `path(z,y)` hypothesis of the second rule is identical to the conclusion of the original rules up to variable renaming. The original rules of (2) also remain after specialization for the same reason.

To make specialization independent of the values of the static input, we perform the following: For any query Q with n distinct static inputs, we generate n fresh constants: say `c1`, ... ,`cn`, and replace the constants in Q with these fresh constants in order (i.e. the first distinct constant by `c1`, the second by `c2`, and so on). Next, we do specialization as described above for the given rules and rewritten Q. Note that, at this point, constants occur in the specialized rules only in the atoms for which no facts are derived by the rules. For any rule in the given set of rules, if a constant `ci` occurs in the rule, we replace it with a variable, say `x`, that does not occur in the rule, add `ci(x)` as a new hypothesis, where `ci` is a fresh predicate name to be used with `ci`, and add the fact `ci(oci)` to the set of rules, where `oci` is the ith original constant in the query. With this result, if another query Q' whose bound arguments are in the same places as Q is given, and Q''s ith constant is different than Q's, we retract the fact related to `ci`, and add a fact of `ci` that represents the new constant. For example, specialization of (3) with respect to the query `path(a,x)?` yields:

$$\begin{array}{l}\texttt{c(a).}\\ \texttt{c(x), edge(x,y)} \rightarrow \texttt{path}_{\texttt{1c}}\texttt{(y).}\\ \texttt{path}_{\texttt{1c}}\texttt{(z), edge(z,y)} \rightarrow \texttt{path}_{\texttt{1c}}\texttt{(y).}\end{array} \tag{6}$$

and the query $\texttt{path}_{\texttt{1c}}\texttt{(y)?}$. If one wants to change the original query to `path(b,x)?`, it is not necessary to re-perform specialization, but just replace the fact `c(a)`, with `c(b)`.

Note that, for any set of rules, specialization does not result in different time complexities of the generated rules when the rule order within the set or the hypothesis order inside the rules is changed.

We have shown that specialization may result in a set with more specialized rules, however it may include unspecialized rules as well. Evaluating a purely specialized set of rules should be more advantageous. The purely specialized rules derived from (3), and the query `path(a,x)?` can be evaluated in linear time in the number of edges. Since the time is proportional to the combination of facts that make the hypotheses true, and `z` can only be assigned the vertices that can be reached from `a` as values, the evaluation takes time proportional to $E(\texttt{a})$. Specialization of the programs (1) and (2) with respect to the same query is evaluated in asymptotically worse time since they include the original rules. Therefore, programs with the same semantics might have different execution times with respect to the same queries, even after specialization.

Differences in time complexity of the specialized programs can only result from the combination of the bound arguments in the query and the version of program that is being specialized, so we show such cases. If the left-recursive version is given and the left argument of the query is bound, or symmetrically if the right-recursive version is given and the right argument of the query is bound, the specialized versions have cost $O(E)$. For the doubly recursive version, no matter which arguments are bound, the complexity is $O(R \times P)$. The following are the complexities of evaluating programs with respect to queries with different bound arguments:

Bound argument	Time complexity		
	Left-rec.	Right-rec.	Doubly-rec.
None	$O(R \times E)$	$O(R \times E)$	$O(R \times P)$
First	$O(E(a))$	$O(R \times E)$	$O(R \times P)$
Second	$O(R \times E)$	$O(IE(b))$	$O(R \times P)$
Both	$O(E(a))$	$O(IE(b))$	$O(R \times P)$

4 Extension by recursion conversion

In the previous section, we showed that specialization might not obtain a more specialized set of rules for a given query. In general, for any set of unspecializable rules, another set of rules that infers the same set of facts may be specializable. For transitive closure, one needs to convert a particular form of recursion into another for the specialization to work. We give a general transformation which is applicable to transitive closure. Given the following set of rules:

$$\begin{array}{l} \mathtt{p}_1(\boldsymbol{x}_1),\ ..\ ,\ \mathtt{p}_n(\boldsymbol{x}_n)\ \rightarrow\ \mathtt{r}(\boldsymbol{x}). \\ \mathtt{r}(\boldsymbol{y}),\ \mathtt{r}(\boldsymbol{z})\ \rightarrow\ \mathtt{r}(\boldsymbol{x}). \end{array}$$

where $\boldsymbol{x}$, $\boldsymbol{x}_n$, $\boldsymbol{y}$, $\boldsymbol{z}$ each denote one or more variables, $\boldsymbol{y}$ and $\boldsymbol{z}$ have common variables $\boldsymbol{t}$, the uncommon ones are in different places in $\boldsymbol{y}$ than in $\boldsymbol{z}$, and at the same place in $\boldsymbol{x}$ as in $\boldsymbol{y}$ or $\boldsymbol{z}$, and the variables in $\boldsymbol{t}$ do not appear in $\boldsymbol{x}$. Also $\mathtt{p}_i$ is not mutually recursive with $\mathtt{r}$. Then the above rules are equivalent to both sets of rules below:

$$\begin{array}{ll} \mathtt{p}_1(\boldsymbol{x}_1),..,\mathtt{p}_n(\boldsymbol{x}_n)\rightarrow\mathtt{r}(\boldsymbol{x}). & \mathtt{p}_1(\boldsymbol{x}_1),..,\mathtt{p}_n(\boldsymbol{x}_n)\rightarrow\mathtt{r}(\boldsymbol{x}). \\ \mathtt{p}_1(\boldsymbol{y}_1),..,\mathtt{p}_n(\boldsymbol{y}_n),\mathtt{r}(\boldsymbol{z})\rightarrow\mathtt{r}(\boldsymbol{x}). & \mathtt{r}(\boldsymbol{y}),\mathtt{p}_1(\boldsymbol{z}_1),..,\mathtt{p}_n(\boldsymbol{z}_n)\rightarrow\mathtt{r}(\boldsymbol{x}). \end{array}$$

where each $\boldsymbol{y}_i$ (and $\boldsymbol{z}_i$) is obtained by substituting the variables of $\boldsymbol{x}_i$ with the substitution that makes $\boldsymbol{x}$ and $\boldsymbol{y}$ (respectively $\boldsymbol{z}$) identical.

All versions of transitive closure are instances of one of these schemas. Since they are all shown to be equivalent and there is a transformation method to transform from one to another, we exploit this fact before specialization.

We give a detailed complexity analysis of specialization extended with recursion conversion for transitive closure. Recursion conversion is also insensitive to hypothesis order or rule order. We just need to consider the main three versions of the transitive closure.

After applying the described transformations to any version of transitive closure, if any of the arguments is bound in the query, the program can be evaluated in $O(E)$ time, and if both arguments are free then the program can be evaluated in $O(R \times E)$ time. One can revise the $O(E)$ bound by more precise bounds as follows:

Bound argument	Time complexity for all three programs
None	$O(R \times E)$
First	$O(E(a))$
Second	$O(IE(b))$
Both	$O(\min(E(a), IE(b)))$

Recursion conversion as described is possible only for the given schema, i.e., doubly-recursive or linear Datalog programs, so it is of significance to convert a Datalog program into a linear one if possible. The question whether it is possible to perform such a transformation has been answered negatively in general, and a subset of Datalog programs have been shown to be convertible to linear ones [1].

For our purposes, any linearization procedure for a subset of Datalog is useful. If we obtain a program which obeys the schema for recursion conversion, we apply the recursion conversion to obtain different versions of the same program. We then apply our specialization algorithm to these different versions. After these steps, we can generate the program as in [19] and automatically analyze the time complexity of the bottom-up evaluation of each resulting program and choose the best one. In any of the steps if the transformation is not possible, we skip that step. The whole method can be summarized as: linearize (if possible), apply recursion conversion (if possible), specialize all versions, generate program, calculate complexity and choose the best. The algorithm is presented in Figure 1.

Algorithm *Demand-driven analysis*
Input: A set of Datalog rules S and a query Q
Output: A sequential program for the generation of answers to Q, with time complexity guarantees

```
if any rule in $S$ is linearizable
  then $S$ = Linearize($S$)
       $RS \leftarrow \{S\}$
for each predicate $p$ in $S$ that fits the recursion conversion schema
    do $S' = p$'s recursion type converted in $S$
       $RS \leftarrow RS \cup \{S'\}$
$RSC = \{\}$ : to keep rule sets with complexities
for each set $R$ of rules in $RS$
    do $R' \leftarrow R$ specialized for $Q$
       $C \leftarrow$ Time complexity of evaluating $R'$
       $RSC \leftarrow RSC \cup \{(R', C)\}$
Among all pairs in $RSC$, remove the ones that are provably worse in complexity
  than at least one pair.
for each pair $(R, C)$ in $RSC$
    do generate program from $R$
       output $C$ as the time complexity associated with it
```

Fig. 1. Algorithm for demand-driven analysis.

The time complexity of the method is dominated by the specialization step, which has a super-exponential upper bound in the maximum arity of the predicates. In practice, the arity of the predicates is relatively small, 2-3 in many realistic Datalog programs and almost never exceeds 10. Thus, assuming a small constant for the maximum arity of predicates, the transformation takes linear time in the size of the set of rules, since for each rule, there is a constant number of different atoms that can unify with its conclusion, and specialization of a rule with respect to an atom takes time proportional to its size.

There are Datalog programs for which recursion conversion is not possible; and specialization cannot succeed in obtaining better running time. In this case, a transformation method such as magic sets may obtain asymptotic speedup with tighter complexity bounds, but the worst-case running times of programs transformed by both our method and magic sets are the same.

5 Comparison

This section discusses the power and limitations of our method in contrast to other work. We consider 12 versions of the transitive closure: the left, right and doubly-recursive programs, and for each program, different order of the two rules, and different order of hypotheses in the recursive rule. We denote the versions by three fields, the first being the recursion type (right, left, or doubly), the second being the order of rules (base-first or recursion-first), the third being the order of hypotheses (regular or inverse). Then for each version, we ask 4 different kind of queries: both arguments bound, only the first argument bound, only the second argument bound, and both arguments free. All results are summarized in Figure 2.

In this figure, we omit the order of rules, because the complexities and inferred facts remain the same for static filtering and magic sets, since they are bottom-up methods. For tabling, since termination is guaranteed, the complexities and inferred facts also remain the same. However, for Prolog evaluation, if the program does not terminate, there will be no inferred facts if the recursive rule is first, otherwise the evaluation will infer some facts, before it gets stuck in an infinite loop.

Method	Bound argument	Time complexity					
		Left-rec.		Right-rec.		Doubly-rec.	
		Regular	Inverse	Regular	Inverse	Reg.	Inv.
Prolog, cyclic gr	Any	Infinite					
Prolog, acyclic gr	Any	Infinite	Exponential	Exponential	Infinite	Infinite	
Tabling	None	$O(V^3)$	$O(V \times E)$	$O(V^3)$	$O(V \times E)$	$O(V^3)$	
	First	$O(E)$	$O(V \times E)$	$O(V^2)$	$O(V \times E)$	$O(V^3)$	
	Second	$O(V^3)$	$O(V^2)$	$O(V^3)$	$O(E)$	$O(V^3)$	
	Both	$O(E)$	$O(V^2)$	$O(V^2)$	$O(E)$	$O(V^3)$	
Static filtering	None	$O(V \times E)$		$O(V \times E)$		$O(R \times P)$	
	First	$O(R(a) \times o)$		$O(R \times E)$		$O(R \times P)$	
	Second	$O(R \times E)$		$O(IR(b) \times i)$		$O(R \times P)$	
	Both	$O(R(a) \times o)$		$O(IR(b) \times i)$		$O(R \times P)$	
Magic set	None	$O(V \times E)$		$O(V \times E)$		$O(V^3)$	
	First	$O(R(a) \times o)$	$O(E)$	$O(V \times R(a) \times o)$	$O(V \times E)$	$O(V^3)$	
	Second	$O(V \times E)$	$O(V \times IR(b) \times i)$	$O(E)$	$O(IR(b) \times i)$	$O(V^3)$	
	Both	$O(R(a) \times o)$	$O(E)$	$O(E)$	$O(IR(b) \times i)$	$O(V^3)$	

Fig. 2. A comparison of time complexities of computation using existing methods.

Prolog. Prolog evaluation resolves subgoals in a top-down fashion. It has the general vulnerability that for any version of the transitive closure, for cyclic graphs, it will not terminate once it enters a cycle, because it will be doomed to resolve the same subgoals infinitely many times. Even when the input is restricted to acyclic graphs, it may still not terminate or it may terminate in exponential time. Prolog does not keep track of discovered vertices and discovers a vertex through all possible paths, which is exponential in the worst case. For versions whose first hypothesis is recursive in the recursive rule, the evaluation will be infinite with respect to all queries regardless of the graph structure. The doubly-recursive versions are always infinite; what differs is the generated facts due to the order of rules and hypotheses.

Tabling. Tabling adds memoization to Prolog evaluation to avoid repeating subgoals. It is guaranteed to be finite and be bounded by $O(V^3)$ for any version and query. If during tabled execution, one ever encounters a `path` call with both arguments free, the time complexity bound will be either $O(V \times E)$ or $O(V^3)$. If one encounters calls to `path` with both or one of the arguments bound, but bound to different values during the execution, then the time is $O(V \times E)$ or $O(V^3)$. If one only encounters calls to `path` with one of the arguments bound to the same value and the other argument free, then the time is $O(E)$ or $O(V^2)$. The criterion on obtaining the bounds in Figure 2 is the amount of data kept for each tabled predicate.

Static filtering and off-line partial evaluation. These are bottom-up procedures, and are not affected by the order of rules and hypotheses. Static filtering and partial evaluation work in essence as the specialization procedure described. Static filtering restricts, i.e. *filters*, the facts used during the evaluation using constants in the query. It is vulnerable to changes of the recursion type in the definition. For example, the method will be able to impose filters on the first argument for the rules in case the left-recursive version is used and the first argument is bound in the query, but will not be able to impose any filters on rules if such a query is asked to the right-recursive version. The doubly-recursive version is not *filterable*.

If static filtering yields linear time evaluation, it does so using less than all edges (except the time to read in all facts); more precisely speaking it only looks at edges reachable from a, which is bound by $R(a) \times o$. Symmetrically, using the right-recursive program with the second argument bound, the evaluation only considers edges that can reach b, which is bound by $IR(b) \times i$.

Dynamic filtering. Dynamic filtering is a version of filtering where the filters are set according to the underlying database during the evaluation. It is not easy to analyze, because the complexity measure may drastically change from one data set to another. As a simplistic overview, we can say that for dense graphs, dynamic filtering behaves exactly the same as static filtering; in contrast, for sparse graphs the filters imposed may remain fairly strict and the evaluation may be better than static filtering, although even for sparse graphs, the filters may reduce to those imposed by static filtering.

Magic set transformation. Generalized supplementary magic set transformation is a transformational method that is used to pass information from one hypothesis to another to mimick top-down evaluation. The resulting time complexity is not affected by the order of rules, but it is asymptotically affected by the version of recursion, and the order of hypotheses in the recursive rule. Another drawback of magic-set variants is that they produce programs that are significantly larger, containining new predicates, new rules and transformed rules with new hypotheses. The time complexity of the evaluation of the transformed programs are $O(E)$ or $O(V \times E)$ depending on how the transformation infers tuples of the given rules using supplementary predicates. For the transitive closure facts inferred, if the supplementary predicates can restrict one of the arguments to a specific value, then it is $O(E)$, otherwise it is $O(V \times E)$ for the left and right-recursive versions, and it is always $O(V^3)$ for the doubly-recursive versions regardless of the queries because no restrictions are possible for at least one of the two recursive hypotheses.

Our method. We have shown that, if any argument is bound in the query, we always obtain $O(E)$ time, which is not possible using other methods. We also present tighter bounds for our method in Figure 2. We believe that our method is strong because it is at least as efficient as other methods and better most of the time, when other methods fail to evaluate these rules efficiently with respect to a query. Also the rules that we generate are simpler, each rule becoming a specialized version of an original rule with respect to the bound arguments, and thus can be understood with respect to the original rules. Therefore, combining all the methods described, i.e., recursion conversion, specialization and program generation, prove to be a powerful method for efficient on-demand analysis.

A drawback of our method is that the context-free reachability queries [22] are not effectively specializable using our method, however we believe that this is not a major drawback since a solution to this problem would be a solution to the famous open problem for proving lower bounds on such problems.

6 Implementation and applications

We have implemented the method and applied it to many problems including program analysis problems. Two examples are described below.

Implementation. The implementation consists of approximately 600 lines of Python code. Even though the running-time is super-exponential in the arity of predicates, since this number is generally small, in all the examples discussed below, the transformations are completed in under 1 second.

Application: strongly connected vertices. A small and illustrative example is computing pairs of vertices in the same strongly connected component. Suppose we use any version of transitive closure and we have the following additional rule: `path(x,y), path(y,x)` → `sameSCC(x,y)`.

Given any argument bound, all prior methods discussed take cubic time in the number of vertices, since there are two subgoals where one has one argument bound, and the other has the other argument bound, therefore resulting in worst case for at least one of them. Our method generates a program that takes linear time in the number of edges.

Another interesting predicate is `notSameSCC`, whose facts are pairs of vertices that are not in the same strongly connected component, which can be obtained by negating either one of the hypotheses in the rule defining `notSameSCC`. First, if the negated predicate is the first one, then top-down evaluation methods will not be able to return correct answers due to negation as failure; and in case this program is rewritten using magic sets, the program does not even remain stratified, so the evaluation of the resulting program is inefficient.

Application: graph query examples. Graph query languages [8, 20, 18] express graph analysis problems as queries on graphs. We take the examples from [18] for program analysis and model checking problems. For example, given a start point, to find the program points `y` such that an uninitialized variable `x` is used for the first time, one may write the following expression: `y: [start] (¬(def(x)|use(x)))* use(x)[y]`.

Intuitively, this says that there is a path from `start` to `y`, such that the path consists of operations that are neither definitions nor uses of `x`, and the path ends with a use of `x`. This is transformed to the following set of rules [20]:

```
def(x1,x2,x) → deforuse(x1,x2,x).
use(x1,x2,x) → deforuse(x1,x2,x).
¬ deforuse(x1,x2,x) → notdef(x1,x2,x).
notdefs(x1,x2,x), notdef(x2,x3,x) → notdefs(x1,x3,x).
notdefs(start,x2,x), use(x2,y,x) → notdefsuse(start,y,x).
```

and a query `notdefsuse(start,y,x)?` would retrieve the answers. For the query `notdefsuse(s,y,x)?`, our method produces the following set of rules:

```
def(x1,x2,x) → deforuse(x1,x2,x).
use(x1,x2,x) → deforuse(x1,x2,x).
¬ deforuse(x1,x2,x) → notdef(x1,x2,x).
notdefs_{1s}(x2,x), notdef(x2,x3,x) → notdefs_{1s}(x3,x).
notdefs_{1s}(x2,x), use(x2,y,x) → notdefsuse_{1s}(y,x).
```

This program is much faster than the original program, since only the program points reachable from a particular point `s` is considered.

Moreover, our method does not require any modification in the presence of stratified negation and the complexity calculation remains the same since with stratified negation, negated hypotheses are looked up in the facts. In case the program is not stratified, we believe that our specialization method still keeps the semantics of the original program with respect to semantics such as well-founded semantics and stable model semantics.

Most graph query representations can automatically be translated into Datalog. This has been shown explicitly for GraphLog in [8]. We take examples from [18] and show the complexity results that our method yields for each of the problems.

We give a table of problems and associated complexities using our method in Figure 3. Shorthands like *undefvars*, *openfiles* are generally self-explanatory abbreviations, denoting the number of undefined variables, and the number of files that are opened, respectively.

Problem	**Complexity**
Uninitialized variables	E(start) × undefvars
Live variables	E(end) × usedvars
Available expressions	E(start) × expr
Constant folding	E(start) × def
Files	E(start) × files
Freed memory	E(start) × freedvars
Interrupts	E(start) × savedvar
Security	E(start) × openfiles
Deadlock avoidance	E(start) × locks^2
Deadlocks	states × outdegree(act) + E(start)
Livelocks	action × states + E(start)

Fig. 3. Time complexities for solving analysis problems.

All the complexities in Figure 3 are asymptotically better than the results without specialization, which is $O(E \times V^n)$ in the worst case, where V is the number of vertices, and n is the number of variables in the query.

We conducted experiments for deadlock and livelock analysis using the VLTS benchmark[1]. Figure 4 shows the results obtained using the specialized rules automatically generated from the description of the problem using the graph query language described above. The first two columns show the vertices and edges in each input file, and the next two columns show the time taken by the analyses in seconds. The experiments were conducted using the Python 2.4.1 interpreter, on a Core 2 Duo 2.8GHz with 2 GB of free memory, running SuSE Linux.

Input	**Vertices**	**Edges**	**Deadlock**	**Livelock**
vasy0_1	289	1224	0.03s	0.01s
cwi1_2	1952	2387	0.09s	0.02s
vasy1_4	1183	4464	0.12s	0.03s
vasy5_9	5486	9676	0.30s	0.06s
cwi3_14	3996	14552	0.43s	0.13s
vasy8_24	8879	24411	0.71s	0.17s
vasy8_38	8921	38424	0.93s	0.25s
vasy10_56	10849	56156	1.35s	0.39s
vasy18_73	18746	73043	2.08s	0.65s

Fig. 4. Experimental results for model checking applications.

The experiments verify the expected results from the time complexity analysis as they grow linearly with the size of the graph. The unspecialized rules for these analyses could only complete on the first input, and even an example as small as the second one could not be completed in 30 minutes. These applications involve computing reachable vertices from a given start node. We ran experiments on XSB [24] to perform the same task using different versions of transitive closure, and verified our bounds presented in Figure 2. For example, given (3) and a query with the first argument bound, the running time ranged from 1 millisecond for 5000 edges, to 7 milliseconds for 30000 edges, behaving linearly as expected. Given the inverse version of (2) for the same data and query, we obtained 830 milliseconds for 5000 edges, and 34280 milliseconds for 30000 edges, reflecting the $O(V \times E)$ bound.

The running times in Figure 4 parallel the results in [18], however they are worse by a constant factor of about 2.5, because our generated program is in Python and the results in [18] are obtained using programs in C++.

7 Related work and conclusion

Datalog has been extensively studied in the literature [6]. Bottom-up evaluation strategies originated from naïve evaluation and extended semi-naïve evaluation. Source-to-source transformations, such as magic set transformations [2, 4] for faster query evaluation, try to mimick the benefits of top-down evaluation.

Although these methods offer a way of possibly faster evaluation, they do not have a succinct method for calculating the time complexity of the evaluation. A

[1] Available at: `http://www.inrialpes.fr/vasy/cadp/resources`

method that generates imperative programs from Datalog rules was developed by Liu and Stoller, and the time complexity bounds given by this method are tighter than the former [19].

Top-down evaluation methods have also been considered for the evaluation of Datalog programs. For recursive query processing, standard Prolog evaluation [21] is not feasible. An extension for Prolog evaluation called tabling, i.e., memoization, has been developed. A particular system that implements tabling is XSB and has been used for deductive databases [7, 24]. One disadvantage for the evaluation of Datalog programs in a top-down fashion is that, there is no well-defined way for calculating the time complexity by only analyzing the rules.

Other methods for efficiently evaluating Datalog programs such as static filtering [15] and dynamic filtering [14] have also been proposed. These methods use special data structures for evaluating Datalog programs rather than using traditional evaluation engines. For static filtering, the computational complexity of the evaluation can be analyzed easily from the rules. For dynamic filtering, however, the computational complexity depends on input data therefore cannot be determined statically.

Using static filtering for the evaluation a Datalog program can be shown to be the same as using partial evaluation combined with the program generation method described. Partial evaluation for logic programming [17] is a general framework for taking static inputs into account for general logic programs. The specialization method that we describe in Section 3 is a simplified form of partial evaluation for Datalog programs.

Borrowing ideas from the theory of grammars for logic programming is natural since the evaluation of both involve similar components. We have incorporated one such idea [5] for our conversion between left-recursive and right-recursive programs. Grammar related ideas for Datalog programs can also be found in, e.g., [10]. Forms of recursion conversion have been discussed in other contexts as well. The conversion from doubly-recursive rules to rules with only one recursive hypothesis is a specific instance of linearization [25, 23].

Our work distinguishes from the previous work in several aspects. Previous work generally focus on one aspect, such as specialization or evaluation alone. Our work combines several techniques: using recursion conversion to obtain different programs with the same semantics in order to specialize better, using specialization for on-demand evaluation, and using automatic program generation with complexity calculations. These together produce efficient specialized programs for demand-driven analysis and provide complexity guarantees specialized for each problem. We have extensively compared and contrasted our method with previous work, and showed that it outperforms previous methods in readability, efficiency, and usability: it generates specialized rules that are simpler than the original rules and are more efficient than using other methods, for a large class of Datalog programs such as the programs that are generated from the query language in [20], and the user just provides the set of rules and the query and our method produces specialized rules for efficient evaluation and generates a program ready to be executed.

References

1. Foto N. Afrati, Manolis Gergatsoulis, and Francesca Toni. Linearisability on datalog programs. *Theoretical Computer Science*, 308(1-3):199–226, 2003.
2. François Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. of the 1986 ACM SIGMOD Intl. Conf. on Management of Data*, pages 16–52, 1986.
3. Steve Barker, Michael Leuschel, and Mauricio Varea. Efficient and flexible access control via logic program specialisation. In *Proc. of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 190–199, 2004.
4. Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *J. Logic Programming*, 10(1/2/3&4):255–299, 1991.
5. Derek R. Brough and Christopher J. Hogger. Grammar-related transformations of logic programs. *New Generation Computing*, 9(2):115–134, 1991.
6. Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases.* Springer, 1990.
7. Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996.
8. Mariano P. Consens and Alberto O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *Proc. of the 9th ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 404–416, 1990.
9. Georg Gottlob and Christoph Koch. Monadic datalog and the expressive power of languages for web information extraction. *J. ACM*, 51(1):74–113, 2004.
10. Sergio Greco, Domenico Saccà, and Carlo Zaniolo. Grammars and automata to optimize chain logic queries. *Intl. J. Foundations of Computer Science*, 10(3):349–, 1999.
11. Katia Hristova and Yanhong A. Liu. Improved algorithm complexities for linear temporal logic model checking of pushdown systems. In *Proc. of 7th Intl. Conf. on Verification, Model Checking and Abstract Interpretation*, pages 190–206, 2006.
12. Katia Hristova, Tom Rothamel, Yanhong A. Liu, and Scott D. Stoller. Efficient type inference for secure information flow. Technical Report DAR 07-35, Computer Science Department, SUNY Stony Brook, May 2007. A preliminary version of this work appeared in *Proc. of the 2006 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security.*
13. Katia Hristova, K. Tuncay Tekle, and Yanhong A. Liu. Efficient trust management policy analysis from rules. In *Proc. of the 9th ACM SIGPLAN Intl. Conf. on Principles and Practice of Declarative Programming*, July 2007.
14. Michael Kifer and Eliezer L. Lozinskii. A framework for an efficient implementation of deductive database systems. In *Proc. of the Advanced Database Symposium.*, 1986.
15. Michael Kifer and Eliezer L. Lozinskii. On compile-time query optimization in deductive databases by means of static filtering. *ACM Trans. Database Systems*, 15(3):385–426, 1990.
16. Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *Proc. of the 24th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 1–12, 2005.
17. Michael Leuschel. Logic program specialisation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation*, volume 1706 of *Lecture Notes in Computer Science*, pages 155–188. Springer, 1998.
18. Yanhong A. Liu, Tom Rothamel, Fuxiang Yu, Scott D. Stoller, and Nanjun Hu. Parametric regular path queries. In *Proc. of the ACM SIGPLAN 2004 Conf. on Programming Language Design and Implementation*, pages 219–230, 2004.
19. Yanhong A. Liu and Scott D. Stoller. From datalog rules to efficient programs with time and space guarantees. In *Proc. of the 5th Intl. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming*, pages 172–183, 2003.
20. Yanhong A. Liu and Scott D. Stoller. Querying complex graphs. In *Proc. of the 7th Intl. Symp. on Practical Aspects of Declarative Languages*, pages 199–214, 2006.
21. David Maier and David S. Warren. *Computing with logic: logic programming with Prolog.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1988.
22. David Melski and Thomas W. Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248(1-2):29–98, 2000.
23. R. Ramakrishnan, Y. Sagiv, J. D. Ullman, and Vardi. Proof-tree transformation theorems and their applications. In *Proc. of the 8th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 172–181, 1989.
24. Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. XSB as a deductive database. In *Proc. of the 1994 ACM SIGMOD Intl. Conf. on Management of Data*, page 512, 1994.
25. Weining Zhang, Clement T. Yu, and Daniel Troy. Necessary and sufficient conditions to linearize double recursive programs in logic databases. *ACM Trans. on Database Systems*, 15(3):459–482, 1990.