the earlier mentioned techniques. Further, fairly complex restrictions are imposed on the sip strategy used, and the rewriting, even for the non-recursive case, is more complicated.

10 Conclusion

We have introduced the notion of parametrized constraints, and developed techniques for pushing parametrized constraints into rules. Our technique enables query restrictions provided by constraints which hitherto could be made use of only in full constraint logic programming systems, to be used in environments which support only the constraint operations of testing and assignment. Our technique properly generalizes Magic Sets rewriting, and is particularly important in the database context where the query evaluation mechanisms typically do not support non-ground facts/constraints. Our technique is generic in the constraint domain, and we have illustrated its use in two different constraint domains.

Acknowledgements

We would like to thank Divesh Srivastava for useful discussions.

References

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Procs. of the ACM Symposium on Principles of Database Systems*, 1–15, 1986.

[BR86] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Procs. of the ACM SIGMOD Conf. on Management of Data*, 16–52, 1986.

[BR87a] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3), 1987.

[BR87b] Catriel Beeri and Raghu Ramakrishnan. On the power of Magic. In *Procs. of the ACM Symp. on Principles of Database Systems*, 269–283, 1987.

[CDE91] Michael Codish, Dennis Dams, and Yardeni Eyal. Derivation and safety of an abstract unification algorithm for groundness and aliasing analysis. In *Procs. of the Int. Conf. on Logic Programming*, 79–93, 1991.

[CoC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Procs. of the ACM Symp. Principles of Programming Languages*, 238–252, 1977.

[JL87] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *Procs. of the ACM Symp. Principles of Programming Languages*, 111-119, 1987.

[KKR90] P.C. Kanellakis, G.M. Kuper, and P. Revesz. Constraint Query Languages. In *Procs. of the ACM Symp. on Principles of Database Systems*, 299–313, 1990.

[KRB85] W. Kim, D. S. Reiner, and D. S. Batory. *Query Processing in Database Systems.* Springer-Verlag, 1985.

[KRBM89] D. B. Kemp, K. Ramamohanarao, I. Balbin, and K. Meenakshi. Propagating constraints in recursive deductive databases. In *Procs. of the North American Conf. on Logic Programming*, 981–998, 1989.

[KS93] David Kemp and Peter Stuckey. Analysis based constraint query optimization. In *Procs. of the Int. Conf. on Logic Programming*, 666–682, 1993.

[LS92] A. Levy and Y. Sagiv. Constraints and redundancy in Datalog. In *Procs. of the ACM Symp. on Principles of Database Systems*, 67–80, 1992.

[MFPR90a] I. S. Mumick, S. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *Procs. of the ACM SIGMOD Int. Conf. on Management of Data*, 1990.

[MFPR90b] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic conditions. In *Procs. of the ACM Symposium on Principles of Database Systems*, 314–330, 1990.

[MP94] Inderpal S. Mumick and Hamid Pirahesh. Implementation of Magic-sets in a relational database system. In *Procs. of the ACM SIGMOD Conf. on Management of Data*, 1994.

[Ram88] Raghu Ramakrishnan. Magic Templates: A spellbinding approach to logic programs. In *Procs. of the Int. Conf. on Logic Programming*, 140–159, 1988.

[RS91] Raghu Ramakrishnan and S. Sudarshan. Top-Down vs. Bottom-Up Revisited. In *Procs. of the Int. Logic Programming Symposium*, 1991.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley-Interscience, 1986.

[SR93a] Divesh Srivastava and Raghu Ramakrishnan. Pushing constraint selections. *Journal of Logic Programming*, 1993. To appear. (A shorter version appeared in the Procs. of the ACM Symposium on the Principles of Database Systems, 1992).

[SR93b] S. Sudarshan and Raghu Ramakrishnan. Optimizations of bottom-up evaluation with non-ground terms. In *Procs. of the Int. Logic Programming Symposium*, 1993.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes I and II.* Computer Science Press, 1989.

or $v \leq u$ where v is a variable, $l \in \Re$ represents the lower bound of variable v, and $u \in \Re$ represents the upper bound of variable v. Note that there is at most one constraint of each form for each v. The domain is ordered by pairwise comparison, that is, for $d_1, d_2 \in VBounds$ $d_1 \sqsubseteq_{VBounds} d_2$ iff for each $v \geq l_2 \in d_2$ there exists $v \geq l_1 \in d_1$ such that $l_1 \geq l_2$, and for each $v \leq u_2 \in d_2$ there exists $v \leq u_1 \in d_1$ such that $u_1 \leq u_2$. The abstraction function $\alpha_{VBounds} : \wp\, Lin \rightarrow VBounds$ is defined as follows:

$$\begin{aligned}\alpha_{VBounds}(\Theta) = \\ &\{v \geq l \mid v \in vars(\Theta), l = glb(v, \Theta) \text{ exists }\} \\ &\cup \{v \leq u \mid v \in vars(\Theta), u = lub(v, \Theta) \text{ exists }\}\end{aligned}$$

The widening operation on the bounds description is defined as follows

$$\begin{aligned}C_1 \bigtriangledown_{VBounds} C_2 = \\ &\{v \geq l_1 \mid (v \geq l_1) \in C_1, (v \geq l_2) \in C_2, l_1 \leq l_2\} \cup \\ &\{v \leq u_1 \mid (v \leq u_1) \in C_1, (v \leq u_2) \in C_2, u_1 \geq u_2\}\end{aligned}$$

Note that $(C_1 \bigtriangledown_{VBounds} C_2) \subseteq C_1$ and hence the finiteness property holds. □

Let us now consider an example of the use of the *VBounds* description during evaluation. For the query $?lpath(a, Y, N), N \geq 2$, after generating the fact $query(lpath(b, 1))$ and then $query(lpath(b, -1))$ the widening operation is applied on the instantiated *PVBounds* descriptions, $\{N \geq 1\} \bigtriangledown_{VBounds} \{N \geq -1\} = \{\}$, and it removes the lower bound. This is achieved by replacing both query facts with $query(lpath(b, -\infty))$.[3] In effect this removes the effect of the constraint rewriting for node b. But note it does not prevent the possible benefits on other paths (e.g. $a \rightarrow e \rightarrow f$).

Given we use a widening operation to maintain exactly one constraint query fact for each original magic fact we can show the following result.

Theorem 7.2 Let P be a program and Q a query. If $Magic(P, Q)$ is finitely evaluable, then so is $CMagic(P, Q)$ given we use a widening evaluation.

Note we can delay the use of widening until some arbitrary finite number of constraint query facts corresponding to a single original magic fact have been generated, and still maintain the above result. For non-recursive programs, widening for the purpose of preventing infinite chains is not required, but subsumption checking is useful.

[3] In general we have to replace both query facts with a query fact for a different annotation not including the widened away parametric constraint. Usually, as in this case, the effect can be achieved using the same annotation.

8 Discussion

The algorithms of [SR93a] and [KS93] can be used beneficially before applying our algorithm, since they introduce more constraints into rules. In addition to directly pruning derivations, the constraints can be used in constraint analysis to deduce tighter parametrized constraints.

If a constraint has been pushed into a predicate definition, it may no longer be necessary to test it where the predicate is used. Techniques from [KS93] can be used to remove redundant constraints after constraint rewriting.

By treating predicates as constraints, our technique can be used to generalize predicate pushdown optimizations (see, e.g., [Ull89]) that are used in database query optimization.

9 Related Work

The most closely related work we are aware of all deal with pushing constraints at compile time: Kemp et al. [KRBM89], Mumick et al. [MFPR90b, MP94], Srivastava and Ramakrishnan [SR93a], Levy and Sagiv [LS92] and Kemp and Stuckey [KS93]. The techniques of [KRBM89] and [MFPR90b] propagate constraints purely syntactically, and do not make deductions using the constraints. The others make use of semantic information about constraints. However, all use only constraints that are fully specified at compile time, and are thus static. Our technique on the other hand uses parametrized constraints, which provide the effect of pushing constraints dynamically at run time. As a result, our technique can handle programs such as the one in Example 1.1, unlike the earlier techniques. Unlike [Ram88] and constraint logic programming implementations, our technique performs constraint manipulation only at compile time, and thereby avoids the cost of constraint manipulation at run time.

Among the earlier constraint propagation techniques, the one most closely related to ours is [KS93]. We borrow from [KS93] the idea of abstract interpretation to determine query constraints. Unlike [SR93a, KS93], we allow multiple adornments per predicate.

The idea of extending Magic Sets beyond equality constraints is present in [MFPR90b, MP94]. However, they allow only simple constraints, and, more importantly, no notion of parametrized constraints is developed. The adornments they use carry incomplete constraint information, and actual constraint information is passed only via static rewriting, not dynamically via parameters. Thus they suffer from the same drawbacks as

$$
\begin{array}{ll}
answer\_f(X, Y, D) & \leftarrow X = sydney, f^a(X, Y, D), D \geq 0, D \leq 10000. \\
f^a(X, Y, D) & \leftarrow query(f^a(\$d_X, \$l_D, \$u_D)), X = \$d_X, \$l_D \leq \$u_D, . \\
 & \quad e(X, Y, D), D \geq 0, D \geq \$l_D, D \leq \$u_D \\
f^a(X, Y, D) & \leftarrow query(f^a(\$d_X, \$l_D, \$u_D)), X = \$d_X, \$l_D \leq \$u_D, e(X, Z, D1), D1 \geq 0, \\
 & \quad f(Z, Y, D2), D2 \geq 0, D = D1 + D2, D \geq \$l_D, D \leq \$u_D. \\
query(f^a(\$d_Z, \$l_{D2}, \$u_{D2})) & \leftarrow query(f^a(\$d_X, \$l_D, \$u_D)), X = \$d_X, \$l_D \leq \$u_D, e(X, Z, D1), D1 \geq 0, \\
 & \quad \$d_Z = Z, \$l_{D2} = max(0, \$l_D - D1), \$u_{D2} = \$u_D - D1. \\
query(f^a(sydney, 0, 10000)). &
\end{array}
$$

Figure 2: Constraint Magic Rewritten Form of Flights Program

are greater than 10000 away from the start node. For such queries on such nodes, the constraint is unsatisfiable, and the satisfiability check in the rule that generates $query(f^a(\ldots))$ facts fails. Thus cities that are at distance greater than 10000 from *sydney* are not explored further. In contrast, the rewritten programs generated by the optimizations of [SR93a] and the optimizations of [KS93], explore all paths of less than 10000, from *all* cities that are connected to *sydney*. $CMagic(P, Q)$ is guaranteed to terminate, so long as all edges have finite positive weights, and we start with a finite bound. □

Adding extra constraints to a constraint program that executes top down (without memoing) cannot compromise its termination since new constraints only serve to fail additional branchs of the derivation tree. The new constraints are added in a manner that ensures they are safe, hence:

Proposition 7.1 If each derivation in $deriv_P(\langle Q, true\rangle)$ is finite and safe then each derivation in $deriv_{CR(P,Q)}(\langle CQuery(P, Q), true\rangle)$ is finite and safe.

The above result does not hold true in general for bottom-up execution. The rewriting introduces new variables into *query* facts, and this can introduce more opportunities for non-termination (beyond what magic sets introduce in any case).

Example 7.2 (Non-Termination) Consider the following program and query.

$$
\begin{array}{l}
lpath(X, Y, N) \leftarrow N = 1, edge(X, Y). \\
lpath(X, Y, N) \leftarrow edge(X, Z), lpath(Z, Y, N1), \\
\qquad\qquad N = N1 + 1, four(N)
\end{array}
$$

with an *edge* relation containing the tuples (a, b), (b, c), (c, b), (c, d), (a, e), (e, f), and *four* relation containing tuples $(1), (2), (3), (4)$. Suppose we had a query $?lpath(a, Y, N), N \geq 2$. Evaluation of the program using a top-down CLP system would loop, due to the cycle between nodes b and c. Evaluation of the Magic Set rewritten program using bottom-up evaluation with a ground constraint solver would terminate as the repeated subqueries would be detected.

The constraint adornment process using *PVBounds* descriptions gives $lpath(X, Y, N) : X = \$d_X, N \geq \$l_N$. The Constraint Magic rewritten program (after redundancy removal) is given below.

$$
\begin{array}{l}
lpath(X, Y, N) \leftarrow query(lpath(X, \$l_N)), N = 1, \\
\qquad N \geq \$l_N, edge(X, Y). \\
lpath(X, Y, N) \leftarrow query(lpath(X, \$l_N)), edge(X, Z), \\
\qquad lpath(Z, Y, N1), N = N1 + 1, N \geq \$l_N \\
query(lpath(a, 2)). \\
query(lpath(Z, \$l_{N1})) \leftarrow query(lpath(X, \$l_N)), \\
\qquad \$l_{N1} = \$l_N - 1, edge(X, Z).
\end{array}
$$

A bottom-up evaluation of this program using duplicate checking will not terminate since it generates an infinite sequence of queries: $query(lpath(a, 2))$, $query(lpath(b, 1))$, $query(lpath(c, 0))$, $query(lpath(b, -1))$, $query(lpath(c, -2))$, and so on. □

To avoid this problem we use (at run-time) the constraint description *VBounds* (presented below) underlying the *PVBounds* description used for adornment. Using the abstract operations defined below on *VBounds* it is simple to maintain a single query fact per original magic fact that subsumes all the query facts already generated. Using the widening operation $\triangledown_{VBounds}$ defined below, infinite chains of query facts will not be generated; if a new bound is tighter than the old bound, the widening operator discards it, and if the new bound is weaker, both bounds are discarded.

Definition 7.1 (Bounds Description) The *bounds description* $\langle VBounds, \alpha_{VBounds}, \wp Lin\rangle$ is defined as follows. The description domain *VBounds* consists of all finite sets of primitive constraints of the form $v \geq l$

$$R0 : answer\_f(X, Y, D) \leftarrow f^a(X, Y, D : sydney, 0, 10000).$$
$$R1 : f^a(X, Y, D : \$d_X, \$l_D, \$u_D) \leftarrow X = \$d_X, \$l_D \leq \$u_D, e(X, Y, D), D \geq 0, D \geq \$l_D, D \leq \$u_D$$
$$\begin{aligned} R2 : f^a(X, Y, D : \$d_X, \$l_D, \$u_D) \leftarrow & X = \$d_X, \$l_D \leq \$u_D, e(X, Z, D1), D1 \geq 0, \\ & \$d_Z = Z, \$l_{D2} = max(0, \$l_D - D1), \$u_{D2} = \$u_D - D1, \\ & f(Z, Y, D2 : \$d_Z, \$l_{D2}, \$u_{D2}), D2 \geq 0, D = D1 + D2, D \geq \$l_D, D \leq \$u_D. \end{aligned}$$

Figure 1: Constraint Adorned Form of Flights Program

If we have a constraint solver that handles safe constraints (i.e. constraints which are tests or assignments when reached) the adornment transformation can cause a nonterminating top-down execution to become terminating. Suppose for example the e relation is cycle free except for edges from *newark* to *newyork* and vice versa, but there is no path of length less than or equal to 10000 from *sydney* to either *newark* or *newyork*. The original program will run forever when it reaches the cycle (assuming no memoing is performed). The adorned program will never reach the cycle.

6 Constraint Magic Rewriting

One of the prime motivations for compiling constraints in the manner we describe in this paper is to restrict computation required in bottom-up evaluation of programs with constraints. This is because typically constraint solvers in bottom-up evaluation are only able to deal with test and assignment constraints, since they do not wish to involve non-ground atoms. If we apply a Magic Sets transformation to the program $CR(P, Q)$ using a complete left-to-right sip strategy and a (magic) adornment such that all parameters are b (bound) arguments and all original arguments are f (free), then we arrive at a range-restricted magic program where all the constraints are tests or assignments when evaluated.

But there is some redundancy in this approach since it adds extra arguments to the atoms, and in fact we are only interested in the non-parameter arguments of the answers and do not require they be connected to the parameter descriptions which generated them. Hence we define an extended Magic Templates rewriting which takes a program P and query Q and produces a program $CMagic(P, Q)$ combining the effects of the Constraints rewriting and Magic rewriting. Example 6.1 illustrates how this separation is managed.

Example 6.1 (Constraint Magic Rewriting of Flights Program) We continue with the adorned program from Example 5.1. Given rule $R2$ Constraint Magic rewriting replaces it by the rule

$$\begin{aligned} SR2 : f^a(X, Y, D) \leftarrow & query(f^a(\$d_X, \$l_D, \$u_D)), \\ & X = \$d_X, \$l_D \leq \$u_D, e(X, Z, D1), D1 \geq 0, \\ & f(Z, Y, D2), D2 \geq 0, D = D1 + D2, \\ & D \geq \$l_D, D \leq \$u_D. \end{aligned}$$

which is restricted to generate a fact $f^a(\overline{t})$ only if there is a fact $query(f^a(\overline{s}))$ which specifies a constraint that is satisfied by $f^a(\overline{s})$. As noted earlier, the constraint parameters are not required in adorned predicates, and have been deleted in the above rule. The literals defining the constraint parameters have also been dropped since they are no longer used.

Further, the following rule is added to the rewritten program in order to generate *query* facts for the literal in the body of the above rule.

$$\begin{aligned} MR2.1 : query(f^a(\$d_Z, \$l_{D2}, \$u_{D2})) \leftarrow & \\ & query(f^a(\$d_X, \$l_D, \$u_D)), X = \$d_X, \\ & \$l_D \leq \$u_D, e(X, Z, D1), D1 \geq 0, \\ & \$d_Z = Z, \$l_{D2} = max(0, \$l_D - D1), \\ & \$u_{D2} = \$u_D - D1. \end{aligned}$$

The above rules constitute the Constraint Magic rewriting of a single rule from the original program. The other rules from the original program are rewritten as well to get the Constraint Magic rewritten program, shown in Figure 2. □

7 Evaluation

We consider issues that arise in the evaluation of a constraint rewritten programs. First we examine the running example.

Example 7.1 (Evaluation of Flights Program) Consider the Constraint Magic rewritten program $CMagic(P, Q)$ from Example 6.1. Semi-naive evaluation (see, e.g. [BR87a]) can be used to evaluate the program. We omit details, but note that the bounds arguments of the $query(f^a(\ldots))$ facts grow tighter as paths are explored further, and the upper bound becomes less than the lower bound for cities that

are simply computed as the tightest bounds, hence $\$l_{D2} = max(0, \$l_D - \$D1)$.

Because the new adornment is the same (modulo renaming) as the original the analysis terminates. prop_adorn also generates similar adornments for $e(X, Y, D)$ which may be used in database retrieval. Although, for brevity, the above example does not illustrate it, in general multiple adornments are computed for each atom. □

5 Constraint Adornment

The results of analyzing the query constraints of a program P are parameterized constraints that hold of all calls to each atom (for some suitable values of the parameters). And guarantee that for each call to an atom the values of the parameters are fixed before the call is made. *Constraint Rewriting* modifies the program to allow the parameterized constraints to be available during execution, thus applying constraint information earlier, in the case that the solver is restricted to only solve ground constraints.

Constraint rewriting takes a program P and query Q and produces a new program $CR(P, Q)$ and query $CQuery(P, Q)$. It proceeds in three phases: first for each rule for atom A extra arguments representing the parameters of the parameterized constraint for A are added to the head atom, and the parameterized constraints are added to the rule body. Next for each atom occurring in the body of a rule or in the query Q, extra arguments are added to the atom representing its parameters and constraints are added to the rule to calculate the parameter values. Finally the constraints in each rule and the query are reordered so that by the time they are reached they are either ground tests or assignments.

The answers of the resulting program are guaranteed to be equivalent to the answers of the original for the query Q because the rewriting procedure has just added redundant constraints. Formally,

Theorem 5.1 Let P be a range-restricted program. Then,

$$\theta \in answers_P(\langle Q, true\rangle)$$
$$\text{iff } \theta \in answers_{CR(P,Q)}(\langle CQuery(P, Q), true\rangle).$$

Example 5.1 (Constraint Adornment) Consider the program P in Example 1.1 with query $Q = (? - f(sydney, Y, D), 0 \leq D, D \leq 10000)$. The Constraint Adornment rewriting propagates parametrized query constraints into programs, and proceeds as follows. Constraint analysis, described in Example 4.2, determines the single parametrized query adornment $f(X, Y, D) : X = \$d_X \wedge D \geq \$l_D \wedge D \leq \$u_D$ for the predicate f. Hence we generate an adorned predicate f^a, where a is defined as the above parametrized constraint. Here, $\$d_X, \$l_D$ and $\$u_D$ are the three parameters to the constraint. The predicate has as arguments the arguments of f, and the values of the parameters $\$d_X, \$l_D$ and $\$u_D$.

We then create specialized forms of the two rules defining f, which use the above constraint to prune their derivations. Each primitive constraint is introduced at the earliest point in the rule where enough of its variables are bound that the values of all variables in the constraint are fixed. In this process, in addition to the given constraint and the constraints already in the rule body, we also use primitive constraints that can be derived. From constraint a, we deduce $\$l_D \leq \$u_D$ by projecting out variable D; it gets added at the beginning of the rule. (The technique is from [KS93], and details will be presented in the full paper.) Finally, from the query on the program we introduce a new predicate $answer\_f$ to compute answers to f that satisfy constraint a with the particular values $\$d_X = sydney$, $\$l_D = 0$ and $\$u_D = 10000$. So $CQuery(P, Q) = (? - answer\_f(X, Y, D))$. The rewritten rules are as follows:

$$answer\_f(X, Y, D) \leftarrow f^a(X, Y, D : sydney, 0, 10000).$$
$$f^a(X, Y, D : \$d_X, \$l_D, \$u_D) \leftarrow X = \$d_X, \$l_D \leq \$u_D,$$
$$e(X, Y, D), D \geq 0, D \geq \$l_D, D \leq \$u_D.$$
$$f^a(X, Y, D : \$d_X, \$l_D, \$u_D) \leftarrow X = \$d_X, \$l_D \leq \$u_D,$$
$$e(X, Z, D1), D1 \geq 0, f(Z, Y, D2), D2 \geq 0,$$
$$D = D1 + D2, D \geq \$l_D, D \leq \$u_D.$$

The analysis phase has derived constraints on the rule body literals. Constraints on database predicates can be used to perform indexing, but for brevity we ignore this. The analysis phase in Example 4.2 determined that the literal $f(Z, Y, D2)$ in the body of the second rule defining f^a also has the parametrized constraint a on it, and deduced the following expression for computing the parameter values at run time:

$$\$d_Z = \$Z, \$l_{D2} = max(0, \$l_D - \$D1), \$u_{D2} = \$u_D - \$D1$$

The equality constraints in this expression are added at the earliest point in the rule body where they are safe. Thus we get the program shown in Figure 1.

We have now obtained specialized rules for all the adorned predicates that we generated, and the adornment step now terminates. The resulting program is $CR(P, Q)$. □

The parametrized bounds descriptions do not involve constants as described above, but can they be extended to allow constants by merging them with *VBounds* descriptions (Definition 7.1); we do not discuss details for lack of space.

4.2 Adornment Analysis

We use "adornments" on predicates to represent (parametrized) constraints specified by queries on the predicate. Let *ACons* be a constraint description. A *query adornment* is a pair in $Atom \times ACons$. We define a function prop_adorn : $\wp(Atom \times ACons) \rightarrow \wp(Atom \times ACons)$ that propagates query adornments. It takes a set S of query adornments for predicates in the program and deduces descriptions of query adornments that queries of the forms in S would generate for the atoms in the bodies of the rules in the program (assuming a left-to-right computation).

Given a sequence $B_1, \ldots, B_m$ we let the notation $\tilde{B}_i, 1 \leq i \leq m$ represent the sequence $B_1, \ldots, B_i$ and let $\tilde{B}_0$ represent the empty sequence,

prop_adorn(S)
  $S = S \cup (Q, C_Q)$ where Q is the initial query on the program and C_Q the query adornment
  for each $(A, C) \in S$
    for each $(A \leftarrow \theta \mid B_1, \ldots, B_m) \in defn_P(A, \{\})$
      for $i = 0$ to $m - 1$
        $S = S \cup \{(B_{i+1}, Arestrict(vars(B_{i+1}), C, \theta, vars(\tilde{B}_i)))\}$
  return S

The crucial feature of the propagation function is the function *Arestrict*. $Arestrict(W, C, \theta, V)$ takes the arguments: C is the abstract parameterized (calling) constraint, θ is the constraint appearing in the current clause, and V are the variables that are fixed by the time execution reaches this point, and computes an the projection of the conjunction of C and θ onto variables W as an abstract parameterized constraint C'. That is, if Θ is a set of constraints and $C \propto \Theta$ then $Arestrict(W, C, \theta, V) \propto \{\overline{\exists}_W \theta \wedge \theta' \mid \theta' \in \Theta\}$. The crucial additional property of *Arestrict* is that it ensures that there is a deterministic function that takes the values for the parameters appearing in C and values for the variables V and computes the values of the parameters in C'. (The function which computes parameter values is inserted into the rule in the adornment rewriting phase.)

The least fixpoint of prop_adorn gives a set of adornments such that each actual query (at run-time) falls within those described by this set. This can be proved quite simply using the theory of abstract interpretation. Formally, using the definition of top-down operational semantics in Section 2,

Theorem 4.1 For each state $\langle A : B, \theta \rangle \sigma$ in $states_P(\langle Q, true \rangle)$, where σ is a renaming, there exists $(A, C) \in lfp(\text{prop\_adorn})$ such that $C \propto \{\overline{\exists}_{vars(A)} \theta\}$.

To finitely evaluate prop_adorn we must perform the evaluation modulo renamings. Under this assumption the least fixpoint of prop_adorn is guaranteed to be finite if either the descriptions are finite for a finite set of variables (as is the case for *PVBounds*), or if the program is non-recursive. In the full paper we present a more complicated version of prop_adorn which uses widening operations to guarantee finite evaluation for infinite domains that have widening operations, even if the program is recursive.

Example 4.2 (Constraint Analysis) Consider the program in Example 1.1. We use the parametrized bounds description (Definition 4.3) in our analysis. The query adornment for $f(X, Y, D)$ is $f(X, Y, D) : C$ where $C \equiv (X = \$d_X \wedge D \leq \$u_D \wedge D \geq \$l_D)$. The only rule that generates new adornments for derived (i.e., non-base) predicates is

$$f(X, Y, D) \leftarrow e(X, Z, D1), D1 \geq 0, f(Z, Y, D2), D2 \geq 0, D = D1 + D2$$

The adornment for the literal $f(Z, Y, D2)$ is calculated as $C' = Arestrict(W, C, \theta, V)$, where $\theta \equiv (D1 \geq 0 \wedge D2 \geq 0 \wedge D = D1 + D2)$, $V = \{X, Z, D1\}$ and $W = \{Z, Y, D2\}$. The resulting adornment (derived as described below) is $C' \equiv (Z = \$d_Z \wedge D2 \leq \$u_{D2} \wedge D2 \geq \$l_{D2})$ with the supporting deterministic function $\$d_Z = \$Z, \$u_{D2} = \$u_D - \$D1, \$l_{D2} = max(0, \$l_D - \$D1)$. (A \$ sign before a variable indicates that the variable is bound.)

The adornment is computed as follows. Fourier-Motzkin elimination (see e.g. [Sch86]) is used to project out the variables $D, D1$ and X from the constraint $D \leq \$u_D \wedge D \geq \$l_D \wedge D1 \geq 0 \wedge D2 \geq 0 \wedge D = D1 + D2 \wedge X = \$X \wedge Z = \$Z \wedge D1 = \$D1$ (which is itself derived from the rule body constraints and the constraints on the head predicate). Parametric variables are just treated as ordinary variables in this process, which is completely independent of the actual values the parametric variables will take at run-time. The result is $(D2 \geq 0 \wedge D2 \geq \$l_D - \$D1 \wedge D2 \leq \$u_D - \$D1)$, which gives the form of the adornment C'. The values

an "abstraction function". (Functions in the original domain are also mapped to functions in the description domain in such a way that the abstract functions are 'correct' w.r.t. the original function, and are discussed in the full version of the paper.) More formally:

Definition 4.1 A *description* $\langle D, \alpha, E\rangle$ consists of a *description domain* (a complete lattice) $(D, \sqsubseteq_D)$, a *data domain* (a complete lattice) $(E, \sqsubseteq_E)$, and a continuous *abstraction function* $\alpha : E \rightarrow D$.
We say that $d \in D$ α-approximates e, written $d \propto_\alpha e$, iff $\alpha(e) \sqsubseteq_D d$. When α is clear from the context we say that d approximates e and write $d \propto e$. □

It is important that whether a given program is terminating or not, an analysis of the program must always terminate. If the description domain has an infinite number of elements, and the given program is recursive, there is a potential for an analysis to run for ever. To avoid this problem, we use a "widening" operation:

Definition 4.2 A *widening operator* [CoC77] for a description domain D is a function $\bigtriangledown_D : D \times D \rightarrow D$ such that $\forall x, y \in D$ both $x \sqsubseteq_D (x \bigtriangledown_D y)$ and $y \sqsubseteq_D (x \bigtriangledown_D y)$ and (crucially) for each increasing chain $x_0 \sqsubseteq_D x_1 \sqsubseteq_D \ldots$ the chain defined by $y_0 = x_0, \ldots, y_{i+1} = y_i \bigtriangledown_D x_i$ contains only a finite number of different elements. Note that by construction $y_i \sqsubseteq_D y_{i+1}$. □

In this paper, we are interested in analyzing constraints (on queries), and we use approximate descriptions of constraints to perform analysis. We call descriptions of sets of constraints *constraint descriptions*. As a simple example, sets of linear arithmetic constraints can be approximated by bounds constraints on individual variables. For instance, the bounds constraint $X < 5 \wedge Y > 0$ approximates $\{X + Y < 5 \wedge Y > 0, X = 4 \wedge Y = 3\}$. As another example, a set of linear arithmetic constraints can be approximated by their convex hull.

There are many possible approximations of a constraint domain; which one to use depends on the 'accuracy' of description that is desired, on the cost of the program analysis, and on the cost of program evaluation based on the analysis. In particular, when analyzing non-recursive programs, we may be able to use the given constraint domain itself, but when analyzing recursive programs we typically want 'coarser' approximations in order to ensure termination of analysis.

In this paper we use parametrized constraints instead of actual constraints so that constants in query constraints can be provided at run time. Hence we use parametrized constraint descriptions to perform program analysis on parametrized query constraints. We need constraint descriptions of the actual constraints (as opposed to parametrized constraints) during evaluation of the program which we discuss in a later section.

We give a formal definition of a particular parametrized description, the 'parametrized bounds description' of the powerset of linear arithmetic constraints, $\wp\, Lin$, below; the non-parametrized version of the description (Section 7) is used during evaluation. In the following definition, parameters $\$l$ and $\$u$ represent bounds that will be fixed only at runtime.

Definition 4.3 The *parameterized bounds description* $\langle PVBounds, \alpha_{PVBounds}, \wp\, Lin\rangle$ is defined as follows. The description domain $PVBounds$ is the set of all finite conjunctions of primitive constraints of the form $v \geq \$l_v$, $v \leq \$u_v$ or $v = \$d_v$ where v is a variable, $\$l_v$ is a parametric variable representing the lower bound of variable v, $\$u_v$ is a parametric variable representing the upper bound of variable v and $\$d_v$ is a parametric variable representing a definite value for v. The domain is ordered by implication, that is, for $d_1, d_2 \in PVBounds$ $d_1 \sqsubseteq_{PVBounds} d_2$ iff $d_1 \rightarrow d_2$. The abstraction function $\alpha_{PVBounds} : \wp\, Lin \rightarrow PVBounds$ maps a set of linear arithmetic constraints (representing their disjunction) to its best description.

$$\alpha_{PVBounds}(\Theta) = \bigwedge \begin{array}{l} (\{ v \geq \$l_v \mid v \in vars(\Theta), glb(v, \Theta) \text{ exists } \} \\ \cup \{ v \leq \$u_v \mid v \in vars(\Theta), lub(v, \Theta) \text{ exists } \} \\ \cup \{ v = \$d_v \mid v \in vars(\Theta), v \text{ is fixed wrt each } \\ \quad \theta \in \Theta \}) \end{array}$$

where $glb(v, \Theta)$ is the greatest lower bound of v values that are compatible with any linear constraint $\theta \in \Theta$. $lub(v, \Theta)$ is defined similarly, and both can be computed using Fourier-Motzkin elimination. The domain has no infinite ascending chains and hence a widening operation is unnecessary. □

Example 4.1 (Bounds Description) Let $\theta_1 \equiv (-X \geq Y \wedge Y \geq 2 \wedge X \geq 0 \wedge Z = 2)$ and $\theta_2 \equiv (Z = 2 \wedge X \leq 10 \wedge Y \geq 2 \wedge Y \leq 4)$. Then $\alpha_{PVBounds}(\{\theta_1\}) = (X \leq \$u_X \wedge X \geq \$l_X \wedge Y \geq \$l_Y \wedge Z = \$d_Z)$, $\alpha_{PVBounds}(\{\theta_2\}) = (X \leq \$u_X \wedge Y \leq \$u_Y \wedge Y \geq \$l_Y \wedge Z = \$d_Z)$ and $\alpha_{PVBounds}(\{\theta_1, \theta_2\}) = (X \leq \$u_X \wedge Y \geq \$l_Y \wedge Z = \$d_Z)$. □

Magic Templates rewriting. Bottom-up evaluation of a constraint-rewritten program has two phases:

Constraint Magic Templates The Magic rewriting technique [BMSU86, BR87b, Ram88] introduces query predicates that carry the parameters of the query constraints deduced earlier. Constraint Magic Templates (Section 6) is a version of Magic Templates rewriting [Ram88] optimized to deal with constraint-adorned programs. We call programs generated by this rewriting *Constraint Magic* programs.

Constraint Evaluation Constraint evaluation (Section 7) is a version of Semi-Naive bottom-up evaluation (see, e.g., [BR87a]), optimized to deal with Constraint Magic Templates rewritten programs. This stage performs bottom-up evaluation of the Constraint Magic rewritten program. For non-recursive programs, standard database view evaluation techniques can be used instead, optionally optimized for evaluating Constraint Magic rewritten programs.

Constraint evaluation of the Constraint Magic rewritten version of a program mimics a top-down evaluation of the program, and has the following benefits over non-memoing top-down evaluation: (a) evaluation is complete (generates all answers in the limit), (b) evaluation does not loop if cyclic subgoals are present, and (c) memoization of answers is performed, so computation need not be repeated.

Example 3.1 (Ground Compilation of Query Constraints) We present a brief example to illustrate some of the components of our technique. We use the domain of equality constraints on structures in this example. Suppose *person* is a collection of objects which have an attribute *age* and an attribute *addr* which itself has an attribute *zipcode*. The following rules selects adults in target zipcodes to send mail to:

$$\begin{aligned} R1: mail\_to(X) &\leftarrow target\_zips(Z), adult(X), \\ &\quad X.addr.zipcode = Z \\ R2: adult(X) &\leftarrow person(X), X.age > 21. \end{aligned}$$

Let the query on the program be $?mail\_to(X)$, which provides no constraints. The selection on zipcode in the definition of R1 cannot be pushed into the definition of *adult* by Magic Sets rewriting [BR87b] since X is not fully ground before the literal $adult(X)$. Magic Templates [Ram88] can push the constraint, but requires the use of non-ground query facts in order to do so.

Our technique deduces that the queries on $adult(X)$ have a parametrized constraint $X.addr.zipcode = \$1$, where the bindings for the parameter \$1 can only be deduced at run-time. Hence it creates an adorned version $adult^a$ of *adult* specialized for the query. It then adds the parameter as an argument of $adult^a$, and introduces the parametrized constraint in the body of the rule defining $adult^a$. It gives the following adorned program (where : is used to separate parameter values from other arguments of adorned predicates):

$$\begin{aligned} R1': mail\_to(X) &\leftarrow target\_zips(Z), adult^a(X:Z), \\ &\quad X.addr.zipcode = Z. \\ R2': adult^a(X:\$1) &\leftarrow person(X), \\ &\quad X.addr.zipcode = \$1, X.age > 21. \end{aligned}$$

We can see that the parametrized constraint on *adult* in rule R1 has been pushed into rule R2', making it available for indexed retrieval of *person* and for restricting the set of $adult^a$ facts generated.

The Constraint Magic rewriting of the adorned program is as follows:

$$\begin{aligned} SR1: \quad & mail\_to(X) \leftarrow target\_zips(Z), \\ & \quad adult^a(X), X.addr.zipcode = Z \\ MR1.1: \; & query(adult^a(Z)) \leftarrow target\_zips(Z). \\ SR2: \quad & adult^a(X) \leftarrow query(adult^a(\$1)), person(X), \\ & \quad X.addr.zipcode = \$1, \\ & \quad X.age > 21. \end{aligned}$$

The constraint parameters are now part of the query predicate, and are no longer part of the predicate $adult^a$.

Bottom-up evaluation of the program generates only ground facts, and yet has pushed the constraint on zipcode into the rule defining $adult^a$. This cannot be achieved by Magic Sets or Magic Templates. We use the example for pedagogical reasons – the program can be optimized by, for example, the techniques of [KRBM89]. However, if the constraint involved a predicate that is recursive with *adult*, or if the definition of *adult* involves aggregation and cannot be unfolded into rule $R1$, earlier techniques are not applicable. □

4 Constraint Analysis

In this section we present a program analysis technique to deduce query constraint patterns.

4.1 Abstract Interpretation

Program analyses, such as groundedness analysis, or in our case constraint analysis, are often performed using *abstract interpretation* [CoC77]. The idea is to map elements of the domain on which the program operates to elements in a description (or abstract) domain which approximates the original domain, but is more convenient for analysis. The mapping is called

set of constraints. A *renaming* is a bijective mapping from Var to Var. We let Ren be the set of renamings, and naturally extend renamings to mappings between atoms, clauses, and constraints. Syntactic objects s and s' are said to be *variants* if there is a $\rho \in Ren$ such that $s\rho = s'$. The *definition of an atom A in program P with respect to variables W*, $defn_P(A, W)$, is the set of variants[2] of clauses in P such that each variant has A as a head and has variables disjoint from $W - vars(A)$. (The above definitions leads to an infinite set, in general, which is finite modulo renaming.)

2.1 Top-Down Operational Semantics

We use the top-down operational semantics of a program in order to prove correctness of our optimization techniques. The top-down operational semantics of a program is defined in terms of its "derivations" which are reduction sequences of "states" where a state consists of the current sequence of atoms and primitive constraints, or "goal", and the current answer constraint. More formally, $Goal = (Atom + Prim)*$, $State = Goal \times Cons$.

A *derivation* of state s for program P is a sequence of states $s_0 \rightarrow \cdots \rightarrow s_n$ where $s = s_0$ and there is a reduction from s_i to s_{i+1} where state $\langle L : G, \theta \rangle$ can be *reduced* as follows:

1. If $L \in Prim$ and $L \wedge \theta$ is satisfiable, it can be reduced to $\langle G, L \wedge \theta \rangle$;
2. If $L \in Atom$, it can be reduced to $\langle B :: G, \theta \rangle$ where $\exists (L \leftarrow B) \in defn_P(L, vars(G) \cup vars(\theta))$.

Note that : denotes an infix *cons* operation and :: denotes concatenation of sequences.

A derivation is *safe* if for each state $\langle L : G, \theta \rangle$ where L is a primitive constraint, L is safe wrt θ. A derivation is *successful* if the last state in the derivation is of the form $\langle \epsilon, \theta \rangle$, where ϵ is the empty goal. The constraint $\overline{\exists}_s \theta$ is an *answer* to state s if there is a successful derivation from s to a final state with constraint θ. We denote the set of answers to s for program P by $answers_P(s)$, the derivations by $deriv_P(s)$, and the set of states appearing in derivations of s by $states_P(s)$.

3 Overview Of The Optimization Technique

We introduce the notion of parametrized constraints in this paper. A *parametrized constraint* is a constraint where parameters of the form $\$i$ may be used as place holders for constants whose value will be known at run-time but is not known at compile time. The $\$i$'s are referred to as parameters to the constraint. We sometimes refer to parametrized constraints as *constraint forms*. For example, $X \geq \$1$ is a parametrized constraint, and represents a constraints that at run-time will have the form $X \geq n$ where n is some constant value.

Given a program, and a query with a parametrized constraint, our optimization technique analyzes the program, deduces the form of constraint subqueries needed to solve the given query, and generates a program which has query constraints compiled in; we call the rewriting technique *Constraint Adornment*, and the generated program a *constraint-adorned* program. The constraint-adorned program can be evaluated top-down, for example using Prolog, or can be evaluated bottom-up after performing a version of the Magic Templates rewriting of [Ram88], modified to handle constraint queries, which we call Constraint-Magic rewriting.

The steps used to generate the constraint adorned program are as follows.

Constraint Analysis: Constraint analysis (Section 4) deduces the possible constraint forms on queries that are generated by the program. For example, queries on a predicate may have the constraint form $X \geq \$a$, where $\$a$ is a parameter dependent on the actual query. The analysis may produce an approximation in case there are too many different constraint forms generated by the program. The approximation is safe in that for every constraint form that could actually be generated, there is a less restrictive constraint form in the approximation; a more restrictive query can be replaced by a less restrictive query without affecting correctness.

Constraint Adornment In this step (Section 5), an 'adorned' form of the original program is generated, with a specialized form of each original rule corresponding to each constraint query form deduced above for the head predicate of the rule. The parameters of the query constraints are added as extra arguments of predicates, and constraint checking code is compiled into the rules, using the values provided by the parameter arguments of the query. This stage can be viewed as an extension of the adornment phase of Magic Sets rewriting [BR87b].

We now have to evaluate the constraint adorned program. This can be done by evaluating the adorned program top-down, using a constraint solver that need only solve ground constraints. Alternatively, we can evaluate the program using bottom-up evaluation with

[2] Since we assumed that all constants are part of the constraints, and do not occur in literals, we need not consider unification here.

queries. Our technique should provide similar benefits, while providing a more powerful selection pushing technique.

Path-selections in object-oriented database query languages can be treated as constraints, and pushed into programs using our technique. By treating stored (base) relations as constraints, our techniques can also be used to generalize predicate pushdown optimizations (see, e.g., [Ull89]).

The query constraints that we consider are different from integrity constraints in that they are not required to hold of the relations; they merely specify constraints on the answers that are required. However, our techniques can be used beneficially as part of other methods to check if integrity constraints are satisfied. Conversely, integrity constraints can be used in our optimization technique to improve the pushing of query constraints.

We present an overview of our technique in Section 3, and a comparison with related work in Section 9.

2 Preliminaries

A *constraint domain* $\mathcal{A}$ consists of a domain of values, an alphabet of constant, function and (constraint) relation symbols, and a fixed interpretation for each of the defined symbols. A *primitive constraint* over a constraint domain $\mathcal{A}$ is of the form $r(t_1, \ldots, t_n)$ where r is an n-ary relation symbol from $\mathcal{A}$ and $t_1, \ldots, t_n$ are terms over $\mathcal{A}$. A *constraint* over $\mathcal{A}$ is a conjunction of primitive constraints over $\mathcal{A}$. For example, the constraint domain $\mathcal{L}$ of linear arithmetic of inequalities on the reals (e.g. [Sch86]) consist of the domain: reals, the alphabet: of rational constants, functions $\{+, -, \times\}$, and relations $\{\leq, <, =, \geq, >\}$, and the usual real interpretation of these symbols. $X < Y$ and $3 \times Z + Y \leq 2$ are primitive constraints in this domain, while $(X < Y) \wedge (3 \times Z + Y \leq 2)$ is a constraint which is not primitive.

A *valuation* σ is a mapping from variables to values in $\mathcal{A}$. We naturally extend valuations to map terms, primitive constraints and constraints. We extend the fixed interpretation of $\mathcal{A}$ under the valuation σ in the usual way. In particular a valuation σ is said to *satisfy* a constraint θ if $\mathcal{A} \models \theta\sigma$. For example, $6 > 5 \wedge 3 < 4$ is satisfied in the reals, and the valuation $\{X/6, Y/3\}$ satisfies the constraint $X > 5 \wedge Y < 4$.

A *projection function*, denoted $\exists_W \theta$, where $W = \{V_1, \ldots, V_n\}$ is a set of variables and θ a constraint, is a (possibly non-deterministic) function that returns a constraint ϕ such that $\mathcal{A} \models (\phi \leftrightarrow \exists\, V_1 \exists\, V_2 \cdots V_n \theta)$. In other words, the function *projects out* the variables $V_1 \ldots, V_n$ from the constraint. The projection function depends on the constraint domain, and we assume that it is provided. For example, given a linear arithmetic inequality constraint $Z \geq Y + 1 \wedge X > Y \wedge Y \geq 4$, the variable Y can be projected out, using the well-known Fourier-Motzkin elimination algorithm (see, e.g., [Sch86]), to get the constraint $Z \geq 5 \wedge X > 4$.

We define $\overline{\exists}_S \theta$, the *projection onto* S of θ, where S is an expression as $\exists_{vars(\theta) \setminus vars(S)} \theta$ where function *vars* takes a syntactic object and returns the set of (free) variables occurring in it.

An *atom* is of the form $p(x_1, \ldots, x_n)$ where p is a predicate symbol and $x_1, \ldots, x_n$ are distinct variables (for simplicity).[1] A *constraint program* over a domain $\mathcal{A}$ is a set of rules of the form $H \leftarrow B_1, \ldots, B_n$ where H, the *head*, is an atom, and in the *body* each B_i is an atom or primitive constraint over $\mathcal{A}$. Often we will be interested in separating the constraint part of a rule, in which case it will be written $H \leftarrow \theta \mid B_1, \ldots, B_m$ where $B_1, \ldots, B_m$ are atoms, and θ is the constraint. The following is an example of a rule in a constraint program:

$$goodpath(X, Y, C, L) \leftarrow C < 500, L \leq 4000, \\ path(X, Y, C, L)$$

A primitive constraint L is a *test* wrt to a constraint θ if θ implies each variable in L is fixed (i.e. for each $v \in vars(L)$ $\theta \rightarrow (v = a)$ for some constant a). A constraint $x = t$ is an *assignment* wrt θ if θ implies all variables in t are fixed and x does not occur in θ. A constraint L is *safe* wrt θ if L is either a test or an assignment wrt θ.

In this paper, we consider programs that generate only ground answer facts and all of whose constraints are safe wrt the bindings (equality constraints) provided by atoms earlier in the rule. A rule is said to be *range-restricted* iff every variable that appears in the rule also appears in an atom in the body of the rule. For example, the rule defining *goodpath* above is range-restricted. A program is said to be *range-restricted* iff every rule in the program is range-restricted. It is easy to show that any fact derived by a range-restricted program is ground. (We can weaken the range-restrictedness (sufficient) condition for groundedness by instead using an analysis such as [CDE91].)

Let *Var* be the set of variables, *Atom* the set of atoms, *Prim* the set of primitive constraints, and *Cons* the

[1] $p(t_1, \ldots, t_n)$ can be replaced by $x_1 = t_1 \wedge \cdots \wedge x_n = t_n \wedge p(x_1, \ldots, x_n)$.

involve parameters. We illustrate the limitations using an example.

Example 1.1 (Flights Query) Consider the following program that computes flights along with their distances, and a query that requests flights of distance less than a specified value.

$$f(X, Y, D) \leftarrow e(X, Y, D), D \geq 0.$$
$$f(X, Y, D) \leftarrow e(X, Z, D1), D1 \geq 0, f(Z, Y, D2), D2 \geq 0, D = D1 + D2$$
Query: ?-$f(sydney, Y, D), 0 \leq D, D \leq 10000$.

The predicate e is assumed to be defined in the database and is also referred to as a *base* predicate.

Evaluation using the algorithms of [SR93a] or [KS93] compute no paths of length greater than 10000, and generate only ground facts and queries. However, if a query asks for paths from node *sydney* of length less than 10000, and we find that node *rio* is at a distance 6000 from *sydney*, we can actually ignore paths from *rio* of length greater than 4000 (i.e., $10000 - 6000$) when answering the query. The above mentioned query evaluation techniques are unable to deduce this, although if the program were evaluated top-down with a full constraint solver this restriction could be inferred. □

In this paper, we present a program rewriting that given a program generates rewritten programs that can propagate query constraints, such as the above, while generating only ground queries, and ground answers provided the original program generates only ground answers. In particular, on Example 1.1 our technique is able to infer and use the tight restrictions on path length, without using a full constraint solver at runtime.

The contributions of this paper are as follows:

1. We introduce the notion of parametrized constraints (Section 3). Parametrized constraints are constraints where some of the constants are not specified, but are provided as parameters (at run time). Parametrized constraints can be thought of as specifying constraint forms.
2. We present a program analysis technique to deduce the forms of constraints in queries and subqueries on a program (Section 4). The technique is based on abstract interpretation, and is generic in that it can be used with any constraint domain. Exact constraint analysis techniques such as [SR93a] can potentially generate an infinite number of constraints on recursive programs; our analysis technique, like that of [KS93], uses safe approximations to ensure that only a finite number of query constraint forms are generated.
3. We present a program adornment technique which rewrites the program rules to explicitly introduce the parametrized query constraints deduced in the analysis phase (Section 5). We call the resultant program the constraint adorned program. The analysis and adornment phase can be viewed as performing compilation on patterns of constraints present in queries.

   The constraint adorned program can be evaluated using any evaluation technique (either top-down or bottom-up) that handles *ground* constraints. This is particularly important in the database context, since most evaluation mechanisms do not handle non-ground facts or constraints.
4. We present Constraint Magic rewriting, which is a version of the Magic Sets query optimization technique [BR87b] tailored to deal with constraint adorned programs (Section 6). The constraint adornment and Constraint Magic rewriting together can be viewed as a powerful generalization of the Magic Sets idea of *compiling* query calling patterns so as to push selections into rules/views (which may be recursive). Magic Sets handles only equality constraints in queries, whereas our technique is applicable to any constraint domains, for example linear arithmetic inequalities.
5. We present an evaluation technique for Constraint Magic rewritten programs (Section 7). The selections provided by the query constraints can enable termination in cases where evaluation of the Magic rewritten program would not have terminated. With respect to a top-down non-memoing approach, we show that our optimizations never introduce non-termination. However, in general, it is possible for a program that terminates when evaluated bottom-up without using query constraints to generate an infinite number of different constraint queries. Our evaluation technique performs 'widening' on query constraints, thereby guaranteeing termination whenever the evaluation of the Magic rewritten program would have terminated.
6. Our approach is applicable to non-recursive as well as to recursive queries on databases. Mumick et al. [MFPR90a] demonstrate the evaluation cost benefits of Magic rewriting for non-recursive

Compiling Query Constraints

(Extended Abstract)

Peter J. Stuckey*
Department of Computer Science,
University of Melbourne
Parkville 3052, Australia
pjs@cs.mu.oz.au

S. Sudarshan
AT&T Bell Labs.
Murray Hill, NJ 07974, U.S.A.
sudarsha@research.att.com

Abstract

We present a general technique to push query constraints (such as *length* ≤ 1000) into database views and (constraint) logic programs. We introduce the notion of parametrized constraints, which help us push constraints with argument values that are known only at run time, and develop techniques for pushing parametrized constraints into predicate/view definitions. Our technique provides a way of compiling programs with constraint queries into programs with parametrized constraints compiled in, and which can be executed on systems, such as database query evaluation systems, that do not handle full constraint solving. Thereby our technique can push constraint selections that earlier constraint query rewriting techniques could not. Our technique is independent of the actual constraint domain, and we illustrate its use with equality constraints on structures (which are useful in object-oriented query languages) and linear arithmetic constraints.

1 Introduction

The area of constraint logic programming [JL87, KKR90] has been receiving a lot of attention in recent times. Such programs generate queries and answers that contain constraints. There are many applications where constraints are very useful, and that also require database support. For instance, queries on flight databases often have upper bound constraints on the total cost of the flights and the number of hops on the flight. Queries on parts hierarchies may specify constraints on the cost of the composite part. Constraint logic programming systems can derive constraints on subqueries from constraints on queries, and prune the set of answers that are generated correspondingly. In general, answers may also have constraints. Constraints are typically expressed as expressions on the variables in a query or answer; for example, a rule $q(X) :- X > 5, p(X)$ gives rise to a constraint query $p(X) : X > 5$.

Query evaluation techniques developed for databases (see, e.g. [KRB85, BR86]) cannot be directly extended to handle constraints since they assume that all facts are ground, i.e., do not contain variables, whereas constraint facts and queries contain variables. (Extensions to handle non-ground facts have been proposed [Ram88, SR93b], but few systems implement them currently, and efficiency is a concern [RS91, SR93b].)

We use the term *program* to refer to both database view definitions and to constraint logic programs, since our techniques are applicable to both domains. Programs where answer facts do not contain any constraints, but where there are constraints (such as $X \leq 0$) in rule bodies, are quite common. If such programs are evaluated in a constraint logic programming system, subgoals that are generated include constraints, and the constraints are used to avoid generating answers that will not be useful. This can provide very significant time and space benefits, and can even allow for termination on programs where evaluation without query constraints would not terminate. On the other hand, subgoals for such programs include variables, hence cannot be evaluated using standard database query evaluation techniques.

An approach used in the past to avoid generating non-ground constraint queries was to rewrite constraint programs to push, as far as possible, constraints into rules at compile time, and derive programs that generate only ground queries. Such approaches were presented in [KRBM89, MFPR90b, SR93a, KS93]. However, these approaches are limited in their ability to push constraints, and cannot handle constraints that

*Research partially supported by Centre for Intelligent Decision Systems and ARC Grant A49130842

In Proceedings of the Symp. on Principles of Database Systems (PODS) 1994