Magic Conditions*

Inderpal Singh Mumick†
AT&T Bell Laboratories
Murray Hill, NJ 07974, USA.
mumick@research.att.com

Sheldon J. Finkelstein‡
ADB/MATISSE
1 Twin Dolphin Drive
Redwood Shores, CA 94065, USA.
sfinkelstein@adb.com

Hamid Pirahesh
IBM Almaden Research Center
650 Harry Road, K55/801
San Jose, CA 95120, USA.
pirahesh@ibm.com

Raghu Ramakrishnan§
Computer Science Department
University of Wisconsin at Madison
Madison, WI 53706, USA.
raghu@cs.wisc.edu

Abstract

Much recent work has focussed on the bottom-up evaluation of Datalog programs [BR88]. One approach, called magic-sets, is based on rewriting a logic program so that bottom-up fixpoint evaluation of the program avoids generation of irrelevant facts ([BMSU86, BR87, Ram91]). It was widely believed for some time that the principal application of the magic-sets technique is to restrict computation in recursive queries using equijoin predicates. We extend the magic-sets transformation to use predicates other than equality ($X > 10$, for example) in restricting computation. The resulting *ground magic-sets transformation* is an important step in developing an extended magic-sets transformation that has practical utility in "real" relational databases, not only for recursive queries, but for non-recursive queries as well [MFPR90b, Mum91].

1 Introduction

A condition is any built-in predicate other than equality. For example, given relations $p(X, Y)$ and $q(Z)$, the local predicate $X > 1$ and the join predicate $Y \leq Z$ are conditions. Equality predicates $X = 1$ and $Y = Z$ are not conditions.

We present an algorithm to rewrite a Datalog [Ull89b] program so that conditions can be applied during an early stage of query evaluation. Our algorithm, called *ground magic-sets transformation*, or *GMST*, can speed up computation by orders of magnitude. We also introduce a new c adornment for conditions.

Query optimization is an important, even critical, phase in the evaluation of database queries. GMST is an optimization technique that pushed down conditions into *complex queries* in both relational and deductive database systems. By "complex," we mean queries that have a number of query blocks (derived predicates, views or subqueries). Recursion may or may not be present. A query with only one query block is *not* complex, even if the single query block has multiple joins and selections.

*An extended abstract appeared as [MFPR90a]

†This work was done while the author was at Stanford University. Part of this work was done at the IBM Almaden Research Center. Work at Stanford was supported by an NSF grant IRI-87-22886, an Air Force grant AFOSR-88-0266, and a grant of IBM Corporation.

‡This work was done while the author was at IBM Almaden Research Center

§Part of this work was done while the author was visiting IBM Almaden Research Center. Work at Wisconsin was supported by an IBM Faculty Development Award and an NSF grant IRI-88-04319.

Traditionally, relational databases have focussed on efficient evaluation of simple, single query block, select-project-join queries that are very common in applications such as corporate databases, airline reservation systems, and banking systems. Many new database applications involve more complex queries. OLAP (On-Line Analytical Processing), Graphics, VLSI design, CAD, and software engineering applications require users to manipulate and operate on data through several levels of abstraction. Amongst traditional applications, bill-of-materials and decision-support queries are complex.

The success of databases in the above applications depends on the efficient evaluation of the complex queries required by the applications. Naive evaluation of a complex query usually takes too long and its performance is unacceptable. Optimization often speeds up the evaluation by one or more orders of magnitude. Many real queries that would take hours or days to evaluate can be executed in seconds or minutes after optimization, making such queries feasible, and in many cases, interactive. Optimization of complex queries is thus very important.

Optimization of a single query block involves choosing join orders and access paths, and considerable work has been done on optimizing relational queries with single query blocks [Ull89b]. However, relatively few techniques (selection pushdown [Ull89b], correlation and decorrelation [Kim82, GW87, Day87]) exist for optimizing nonrecursive complex queries (queries with multiple query blocks). On the other hand, for recursive complex queries, extensive optimization methods have been developed in the context of deductive databases [BR88, Ull89b, CGT90]. Deductive databases are very similar to relational databases. Indeed, once you extend a relational query language (such as SQL [ISO93]) with general recursion that allows for mutual recursion between relations, every query expressed in Datalog, a popular query language for deductive databases, can be expressed in the relational language.

One of the most useful optimization techniques in deductive databases is the magic-sets transformation [BMSU86, BR87]. The idea behind the magic-sets technique is to restrict computation of intermediate relations to exactly those tuples that are needed to answer a given database query. How do we know that a tuple might be needed to compute an answer? In magic-sets, a tuple is considered useful if it would be computed in a top-down (backward chaining) evaluation of the given recursive query. The magic-sets transformation is very similar to predicate pushdown techniques in relational algebra — the difference is that magic-sets transformation can push down join predicates as well as selection predicates, and it can push the predicates through recursion. However, unlike predicate pushdown, the magic-sets transformation cannot push conditions (predicates other than equality, such as $<, \leq, >, \geq$, and $\neq$); magic-sets transformation can only push down equality predicates. Such conditions are widely used in practical database queries. Examples are: salary greater than 50K (a selection condition on one relation), or last year's sales more than this year's sales (a join condition between two relations).

The magic-sets transformation was initially believed to be useful only for recursive queries in deductive databases. However, more recently [MFPR90b, Mum91], it has been demonstrated that the magic-sets transformation is useful for nonrecursive queries in relational databases as well.

We develop a unified technique to push equality predicates and conditions in nonrecursive and recursive queries. The resulting *ground magic-sets transformation* is used as a first step in developing an integrated optimization technique for pushing all types of predicates in Starburst SQL queries [Mum91]. An extended abstract reporting preliminary results of this work was presented at PODS [MFPR90a]. The relationship between this journal paper and the earlier conference papers is discussed in Section 11.

Previous work on magic-sets has used b and f adornments to distinguish between bound (b) and free (f) argument positions. In this paper, we introduce a new c adornment that describes selections involving conditions. We show how to adorn a program when information about conditions as well as equality predicates is to be passed in a top-down fashion. The heart of the paper is the ground magic-sets transformation (GMST) that shows how to push conditions and equality join predicates while preserving range-restrictedness. GMST can thus optimize queries in a relational database system. GMST works by logically first doing a transformation similar to magic templates [Ram91]. This however leads to programs that are not range-restricted. A logically second step restores range-restrictedness by doing a grounding transformation. These two logical steps are interleaved to derive the GMST algorithm.

Section 2 provides background information. We introduce Datalog, define range-restricted rules and strata (our definition differs slightly from that in [Ull89b]), introduce adornments and sips, define the magic-sets transformation, and describe the supplementary magic-sets transformation. Section 3 motivates the need for ground magic-sets transformation (GMST) and for the c adornment. We define the c adornment in Section 4, show that sips can be used to describe how conditions are passed into subgoals of a rule in Section 5, and give an algorithm to adorn programs using b, c, and f adornments in Section 6. The ground magic-sets transformation is presented in Sections 7-9. Section 7 provides an overview of GMST, and Section 8 discusses the conditions under which GMST is applicable. Details of the GMST algorithm appear in Section 9. Possible extensions to GMST are discussed in Section 10, and Section 11 relates GMST to alternative techniques for pushing down conditions. Conclusions are presented in Section 12.

2 Background on the Magic-sets transformation

The query language "Datalog" for deductive databases consists of "if-then" rules, like the rules $T1$ and $T2$ below:

($T1$): $p(X,Y) \mathrel{:-} e(X,Y)$.
($T2$): $p(X,Y) \mathrel{:-} e(X,Z) \ \&\ p(Z,Y)$.

Each rule has two parts: an "if" part called the *body*, and a "then" part called the *head*. In rule $T1$, $e(X,Y)$ is the body, and $p(X,Y)$ is the head. In rule $T2$, $e(X,Z)$ & $p(Z,Y)$ is the body, and $p(X,Y)$ is the head.

Rules $T1$ and $T2$ compute the transitive closure, represented by p for `path`, of the edge relation, represented by e.

A collection of Datalog rules is called a Datalog program. We often use a letter, such as T, to represent a program comprised of rules $T1, T2, \ldots$

Relations such as e and p are called *predicates*, or more specifically, *ordinary predicates*. An ordinary predicate whose extension is stored in the database is called an *extensional database* (edb) or *base* predicate. An ordinary predicate whose extension is *not* stored in the database, but is derived, is called an *intensional database* (idb) or *derived* predicate. An edb predicate cannot appear in the head of any rule. An idb predicate must appear in the head of at least one rule. In program T, e is an edb predicate and p is an idb predicate.

In addition to ordinary predicates, Datalog allows *built-in predicates*, such as $X > Y$, to do arithmetic comparisons. Six built-in predicates are allowed: $<, \leq, >, \geq, =$, and $\neq$. Other built-in predicates can also be handled by the magic conditions algorithm.

2.1 Range-Restricted Rules

Datalog requires each rule to be range-restricted. The ideas behind range-restrictedness is that each variable in the head literal can only assume a finite number of values, so a range-restricted rule derives a finite relation for the head predicate. A program that does not compute infinite relations is said to be *safe*. Range-restrictedness is a sufficient condition for the safety [Ull89b] of Datalog programs.

Definition 1 — Range-Restricted Rule: A variabled X is *limited* in rule r if X appears as an argument of an ordinary subgoal of rule r, or if X appears in a built-in subgoal of the form $X = c$, or if X appears in a built-in subgoal $X = Y$, where Y is known to be otherwise *limited*.

A rule is said to be range-restricted if every variable appearing in the rule is *limited*. □

EXAMPLE 1 In the rule r,

(r): `buys`$(X,Y) \mathrel{:-}$ `rich`(X).

variable X is *limited* by virtue of appearing in the `rich` subgoal. Variable Y is not *limited*, and therefore rule r is not range-restricted. □

The definition of a *limited* variable given above is syntactic. A variable X that appears in two built-in subgoals $X \leq 5$ and $X \geq 5$, is not considered *limited* under this definition. One can extend the definition of *limited* to account for inferences that can be made from the given predicates, thereby making *limited* into a semantic property, dependent upon an inference algorithm. The definitions of bound and condition adornments given later can be similarly extended. Alternatively, one can assume that all the inferred predicates have already been placed in the rules in a pre-processing phase. We choose, without loss of generality, not to deal with predicate inferencing in this paper.

2.2 Non-range-restricted rules

Bottom-up evaluation of programs that are not range-restricted requires us to store *nonground* tuples. With range-restricted programs, only *ground* tuples need to be stored.

A tuple is *ground* if it does not have a variable. $sg(5,3)$ is a ground tuple. A tuple is *nonground* if it contains variables. $p(X,5)$ is a nonground tuple. Conditions may be present on or between the variables of a nonground tuple. $q(Y,3)$ & $Y > 5$ is also a nonground tuple.

Well-known commercial and experimental relational (System R, DB2, Starburst, Ingres) and deductive (NAIL!, LDL) data base management systems do not support nonground tuples. Currently, application queries are mostly range-restricted, and performance is of prime importance. One expects better performance from a system that limits itself to ground tuples due to the following reasons:

- Fast access methods like hashing and indexing are difficult to extend to nonground tuples, though some preliminary work on indexing nonground tuples has been presented in [KRVV93].
- The subsumption problem (is a tuple subsumed in another?) needs to be solved when adding a tuple to a nonground relation, but the problem does not exist for ground relations.
- When trying to find a satisfying substitution for variables in a rule, we need to unify the subgoals with stored tuples. Unification is the process of trying to make two terms identical by finding substitutions for variable in the two terms. When one of the two terms to be unified does not have variables, the task is easier, and is referred to as *matching*. If we limit ourselves to ground tuples, we can use matching.

Due to these performance penalties, the systems listed above do not support non-range-restricted rules. Hence, for a rewrite transformation to be useful in a wide class of database systems, it is critical that the rewrite preserve the range-restrictedness of a program. The ground magic-sets transformation presented in this paper will preserve range-restrictedness while passing conditions.

2.3 Strata

We will assign a number to each predicate that reflects, roughly, the number of derivation steps needed to compute the extension of the predicate from the base predicates. This number is called the stratum of the predicate.

To compute the strata of predicates in a program, we construct a *reduced dependency graph* by collapsing every strongly connected component of the dependency graph [Ull89b] to a single node. A reduced graph is guaranteed to be acyclic. A topological sort [AHU83] of the reduced dependency graph assigns a stratum number to each node. If a node represents a strongly connected component, all predicates in the strongly connected component are assigned the stratum number of the node. By convention, the base predicates are assigned stratum number $= 0$.

EXAMPLE 2 Consider the Datalog program S:

$(S1)$: q :- r.
$(S2)$: q :- s.
$(S3)$: r :- t.
$(S4)$: r :- u & s.
$(S5)$: u :- r & t.

We have ignored the arguments of predicates as they are irrelevant to this example. Figure 1 shows the

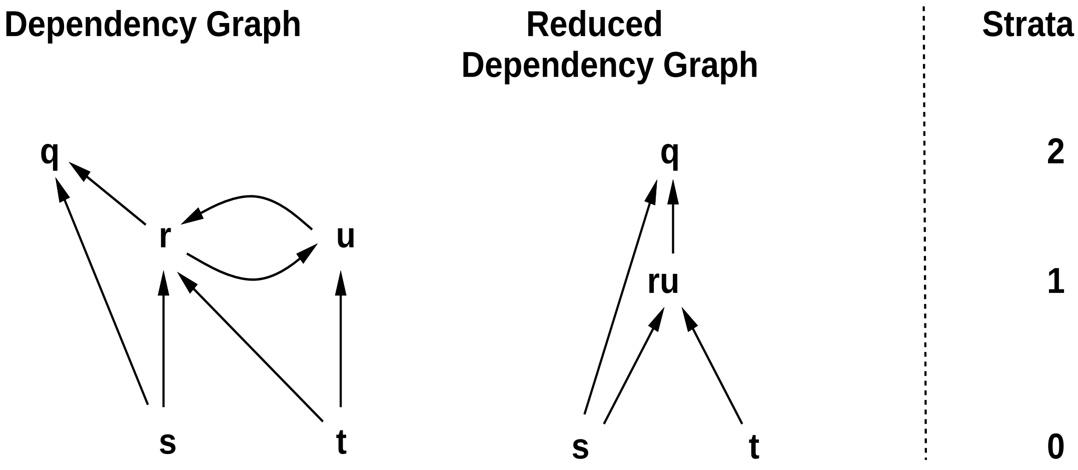

Figure 1: Dependency Graph, Reduced Dependency Graph, and Strata for program S.

dependency graph and the reduced dependency graph for program S. A topological sort assigns stratum 0 to the base predicates s and t, stratum 1 to the condensed node ru, and thereby to the mutually recursive predicates r and u, and stratum 2 to the derived predicate q. □

Our definition of strata differs from the definition given by Ullman ([Ull89b], Chapter 3.6). Given two mutually nonrecursive predicates p and q in a Datalog program (no negation), Ullman would place both predicates into the same stratum. However, we place predicates p and q in different strata. If predicate p depends on predicate q, then predicate p is placed in a higher stratum. If predicates p and q do not depend on each other, then an arbitrary ordering is chosen by the topological sort.

2.4 Predicates

An *ordinary predicate* represents a database relation. A *built-in predicate* represents an interpreted predicate such as $=, \neq, <, \leq, >, \geq$. The built-in predicate $=$ is called an equality predicate, and is usually represented implicitly by using the same variable name in two positions.

A built-in predicate that relates two arguments of one ordinary subgoal in a rule is called a *local predicate*, or a *selection predicate*. A built-in predicate that relates two arguments of two distinct ordinary subgoals in a rule is called a *join predicate*. An equality predicate that is also a join predicate is called an equijoin predicate. An equality predicate that is also a selection predicate is called an equality selection predicate.

2.5 Adornments

An adornment is an annotation on a predicate that gives information about how the predicate is used.

EXAMPLE 3 Consider the transitive closure program T:

$(T1)$: $p(X,Y)$:– $e(X,Y)$.
$(T2)$: $p(X,Y)$:– $e(X,Z)$ & $p(Z,Y)$.

The query:

(Q): ?- $p(1,Y)$.

binds the first argument of p to the constant 1. The binding is used in a top-down evaluation of p. For example, in rule $T2$, the variable X gets bound to 1. We look up e using this binding, and get a set of bindings for variable Z. These bindings are then passed to the p subgoal, generating p goals that have the same *binding pattern* as the query goal, and will be evaluated in a similar manner.

Thus, whenever we generate a p goal during the evaluation of query Q, the first argument of p will be bound to a constant. □

The presence or absence of a binding on an argument of a predicate can be indicated by adorning the predicate. For example, the query goal Q can be adorned $p^{bf}(1, Y)$. The superscript bf is an adornment for predicate p, indicating that the first argument is bound (b) and the second argument is free (f). We do not adorn base predicates.

The adornment of an n-ary predicate is defined to be a string of b's and f's of length n. A b in the i^{th} position means that the i^{th} argument is *bound*, while an f means that the i^{th} argument is *free*.

EXAMPLE 4 The flow of information during the top-down evaluation of query Q as discussed in Example 3 is represented by the adorned program AT:

$(AT0)$: ?- $p^{bf}(1, Y)$.
$(AT1)$: $p^{bf}(X, Y)$:- $e(X, Y)$.
$(AT2)$: $p^{bf}(X, Y)$:- $e(X, Z)$ & $p^{bf}(Z, Y)$.

Adorned predicates p^{bf} and p^{fb} are distinct from each other and from the unadorned predicate p, though the rules for p^{bf} and p^{fb} were derived from the rules for p.

Program AT is called an adorned program. □

The idea of b and f adornments was introduced by Ullman [Ull85]. Since then, many authors have used the b and f adornments, but the meaning they attach to the b adornment varies. In [Ull85, BMSU86, BR87] an argument of a subgoal in a rule is considered bound if the argument is bound to a constant in *all* runtime goals generated from this subgoal during a hypothetical top-down evaluation. In [Ram91], an argument of a subgoal is considered bound if the argument is potentially restricted (that is, it may be bound) by some predicate in all runtime goals generated from this subgoal. In this paper, we use the interpretation of [Ull85].

Definition 2 — The *bf* Adornment Class: The *bf* adornment class is a system of adornments that distinguishes between bound (b adorned) and free (f adorned) argument positions.

An argument of a subgoal in a rule is *bound* if one can guarantee that during a hypothetical, tuple at a time, top-down evaluation of the rule, the goal generated from this subgoal will have a constant in this argument position. Otherwise, the argument is *free*. □

Note that Definition 2 gives the property that b and f adornments must satisfy. It does not tell us how to determine the adornments. Under Definition 2, an argument of a subgoal s can get a b adornment if the argument is *limited* by the ordinary subgoals appearing before subgoal s in the subgoal order, and by the built-in subgoals in the rule. Otherwise the argument must get an f adornment. Whether all arguments that are such *limited* get a b adornment depends upon the sips being used.

2.6 SIPS

A sideways information passing strategy (sips[1]) is a decision on how to pass information sideways in the body of a rule while evaluating the rule. A sips specifies how bindings available in the head will be used, the order in which subgoals in the body will be evaluated (join order), and how bindings will be passed between predicates in the body.

We used the idea of sips in Examples 3 and 4 when talking about passing bindings and determining adornments of subgoals. In a sips, we can pass the head bindings, and bindings obtained from any previously

[1]Note that sips is an acronym for both, *a* sideways information passing strategy, and, *many* sideways information passing strategies.

evaluated subgoal, into a yet to be evaluated subgoal. A sips therefore depends on, and can define, a subgoal order. In Examples 3 and 4 we chose a sips consistent with the given subgoal.

A formal definition of sips was given by Beeri and Ramakrishnan [BR87]. As defined by them, sips only consider how *bindings* (equality predicates) are passed sideways. We therefore refer to these sips as the *bf* sips.

Definition 3 — The *bf* sips: The sideways information passing strategy, $\mathsf{sips}(r, \alpha)$, for a rule r and a head adornment α is a directed labeled graph. The edges of the graph induce a partial order (sips order) in which the subgoals are to be evaluated, while the labels indicate the bindings to be passed from one subgoal to another.

A sips that passes all available bindings is called a full sips. while a sips that does not pass all available bindings is called a partial sips. □

2.7 Magic-sets Transformation (MST)

The idea behind the magic-sets technique is to compute an auxiliary "magic" predicate for each idb predicate in the original program. The magic predicate collects the bindings from all runtime goals for the associated predicate that would be generated in a top-down evaluation of the program. The rules of the program are rewritten using the magic predicates so that "irrelevant" tuples are not generated during a bottom-up evaluation. A tuple is considered irrelevant if it is not an answer to any runtime goal. Thus, magic-sets transformation makes a bottom-up evaluation at least as efficient as a top-down evaluation in the number of computed tuples.

The magic-sets transformation is defined on adorned programs, and is guided by a sips. The following definition is as described in [BR87]. We assume that a full sips is used.

Definition 4 — The Magic-sets Transformation: Given an adorned program AP, an adorned query goal q^α, and a full sips for each rule for AP, the magic-sets transformation derives a magic program MP as follows:

1. Create a new predicate m_p (the magic predicate for p) for each derived predicate p in AP. The arity of m_p is the number of bound arguments of p.

2. For each rule r in AP, add a *modified version* of r to MP. If rule r has head $p(\overline{t})$, where $\overline{t}$ is shorthand for all arguments of p, the modified version is obtained by adding the literal $m\_p(\overline{t^b})$ into the body of r, where $\overline{t^b}$ denotes all bound arguments of $p(\overline{t})$.

3. For each rule r in AP with head $p(\overline{t})$, and for each subgoal $q_i(\overline{t_i})$ in its body such that q_i is a derived predicate, add a *magic rule* to MP. The head is $m\_q_i(\overline{t_i^b})$. The body contains all subgoals that precede q_i in the sips order associated with r, as well as the literal $m\_p(\overline{t^b})$.

4. Create a *seed* fact $m\_q(\overline{c})$, where $\overline{c}$ is the set of constants equated to the bound arguments of the query goal.

□

EXAMPLE 5 Consider the query

(Q): ?- $p(1, Z)$.

on the program T of Example 3. Assume the sips for rule $T2$ for head adornment p^{bf} requires that the head binding be passed into e, and that e pass the binding on Z into the p subgoal.

Program T and the query Q are adorned as in Example 4:

$(AT0)$: ?- $p^{bf}(1, Z)$.
$(AT1)$: $p^{bf}(X, Y)$:- $e_1(X, Y)$.

$(AT2)$: $p^{bf}(X,Y)$:- $e_2(X,Z)$ & $p^{bf}(Z,Y)$.

We have marked the two occurrences of e in rules $AT1$ and $AT2$ by subscripts. Given the adorned program AT and the sips as stated above, the magic program MT is derived as follows:

1. Create the magic predicate m_p^{bf} for the derived predicate p^{bf}.
2. Modify rules $AT1$ and $AT2$ to obtain the following rules $MT1$ and $MT2$:

   $(MT1)$: $p^{bf}(X,Y)$:- $m\_p^{bf}(X)$ & $e_1(X,Y)$.
   $(MT2)$: $p^{bf}(X,Y)$:- $m\_p^{bf}(X)$ & $e_2(X,Z)$ & $p^{bf}(Z,Y)$.

3. From rule $AT2$ and the subgoal $p^{bf}(Z,Y)$, derive the magic rule $MT3$:

   $(MT3)$: $m\_p^{bf}(Z)$:- $m\_p^{bf}(X)$ & $e_2(X,Z)$.

4. Add the seed fact:

   $(MT4)$: $m\_p^{bf}(1)$.

The magic-set, m_p^{bf} collects all the nodes reachable from 1 along the edges e_2. This is exactly the set of X values for which a goal $p(X,Y)$ would be generated by a top-down evaluation. Subsequent evaluation of rules $MT1$ and $MT2$ is then restricted to compute the paths from one of the reachable nodes, just as a top-down evaluation would restrict itself to only computing answers to its top-down goals. □

The magic-sets transformation was first proposed by Bancilhon, Maier, Sagiv and Ullman [BMSU86]. They presented the transformation on the transitive closure program, and did not give a general algorithm. The name *magic-sets* was coined by Maier, partly because of its *magical* effect on query performance, and partly because its mechanics were not fully understood. Beeri and Ramakrishnan [BR87] gave the generalized magic-sets transformation. Ullman [Ull89a] showed that magic-sets mimics a top-down evaluation with memoing.

2.8 Supplementary Magic-sets Transformation (SMST)

In Step 3 of the Magic-sets Transformation (Definition 4), we create a magic rule for each derived subgoal q in rule r by copying the subgoals preceding q in rule r. If rule r has multiple derived subgoals, parts of r will be copied multiple times. Thus, during an evaluation of the magic program, joins may be repeated many times.

EXAMPLE 6 Consider the magic program MT of Example 5:

$(MT1)$: $p^{bf}(X,Y)$:- $m\_p^{bf}(X)$ & $e_1(X,Y)$.
$(MT2)$: $p^{bf}(X,Y)$:- $m\_p^{bf}(X)$ & $e_2(X,Z)$ & $p^{bf}(Z,Y)$.
$(MT3)$: $m\_p^{bf}(Z)$:- $m\_p^{bf}(X)$ & $e_2(X,Z)$.
$(MT4)$: $m\_p^{bf}(1)$.

The subgoals "$m\_p^{bf}(X)$ & $e_2(X,Z)$" in rule $MT2$ are copied into the magic rule $MT3$ during Step 3 of the magic-sets transformation. The join between $m\_p^{bf}(X)$ and $e_2(X,Z)$ is thus repeated twice, once in rule $MT3$ to compute the magic-sets, and once in rule $MT2$ to compute p^{bf}. □

The supplementary magic-sets transformation is a variant of the magic-sets transformation that identifies the common subexpressions between magic and regular rules, stores them in intermediate, or supplementary, predicates, and does some optimization to delete redundant columns.

The transformation works as follows: When constructing a magic rule for m_q from the subgoal q in a rule r, first construct a supplementary predicate. The supplementary predicate is defined by taking the join of all subgoals preceding q in rule r. If rule r contains a magic predicate, it is considered to be the first subgoal, and is included in the join. Any variables not used in the head and not used in a subgoal after q are projected out from the supplementary predicate. Second, replace the subgoals preceding q in rule r by the supplementary predicate, and use the supplementary predicate as a common subexpression to derive the magic predicate m_q. A definition of the supplementary magic-sets transformation can be found in [BR87, Ull89b].

EXAMPLE 7 Consider the adorned program AT of Example 4.

$(AT1)$: $p^{bf}(X,Y)$:- $e_1(X,Y)$.
$(AT2)$: $p^{bf}(X,Y)$:- $e_2(X,Z)$ & $p^{bf}(Z,Y)$.

Before constructing the magic rule for m_p^{bf} from the subgoal p^{bf} in rule $AT2$, we create a supplementary predicate $s_{2.1}$ from the join of $m\_p^{bf}(X)$ and $e_2(X,Z)$. Both variables X and Z are arguments if $s_{2.1}$, since X appears in the head and Z appears in a later subgoal.

$(ST5)$: $s_{2.1}(X,Z)$:- $m\_p^{bf}(X)$ & $e_2(X,Z)$.

The supplementary predicate is subscripted 2.1 to indicate that it is derived from the *second* rule $AT2$, and just after evaluating the *first* subgoal of rule $AT2$. $s_{2.1}$ is used as a common subexpression when modifying rule $AT2$ to $ST2$ and when deriving the magic rule $ST3$:

$(ST2)$: $p^{bf}(X,Y)$:- $s_{2.1}(X,Z)$ & $p^{bf}(Z,Y)$.
$(ST3)$: $m\_p^{bf}(Z)$:- $s_{2.1}(X,Z)$.

We can derive rules $ST1$ and $ST4$ in a manner similar to $MT1$ and $MT4$ in Example 5, completing the supplementary program SP:

$(ST1)$: $p^{bf}(X,Y)$:- $m\_p^{bf}(X)$ & $e_1(X,Y)$.
$(ST2)$: $p^{bf}(X,Y)$:- $s_{2.1}(X,Z)$ & $p^{bf}(Z,Y)$.
$(ST3)$: $m\_p^{bf}(Z)$:- $s_{2.1}(X,Z)$.
$(ST4)$: $m\_p^{bf}(1)$.
$(ST5)$: $s_{2.1}(X,Z)$:- $m\_p^{bf}(X)$ & $e_2(X,Z)$.

□

3 Motivation

Consider the following version of the same generation program:

$(P1)$: $sg(X,Y)$:- $\texttt{flat}(X,Y)$.
$(P2)$: $sg(X,Y)$:- $\texttt{up}(X,U)$ & $sg(U,V)$ & $\texttt{down}(V,Y)$.

We interpret sg as defining a *same level* relation on nodes of a graph. Two nodes X and Y are at the same level if they are connected by a `flat` arc (rule $P1$), or if there is an `up` arc from X to U, a `down` arc from V to Y, and U and V are at the same level. Figure 2 shows a sample database.

Suppose we want to find the nodes that are at the same level as any node > 5:

(Q): ?- $X > 5$ & $sg(X,Y)$.

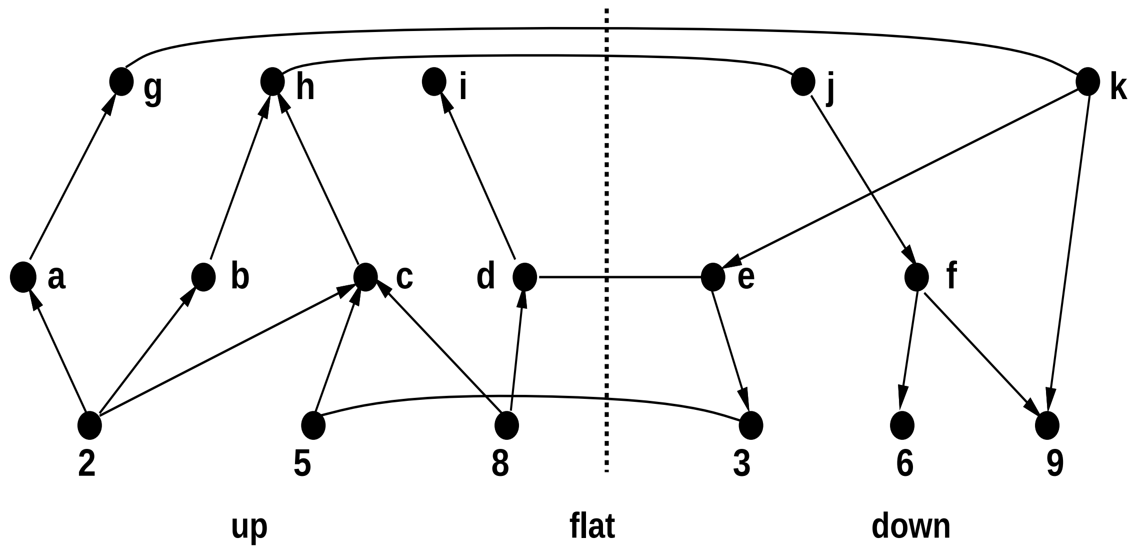

Figure 2: Sample `flat,` `up`, and `down` relations for program P.

Direct Evaluation
Algorithm 1 given below computes query Q without any rewrite optimization.

Algorithm 1 We evaluate query Q in two steps:

1. Compute the full sg relation. For our example database, we will compute

$$sg = \{53, de, gk, hj, 83, a9, bf, cf, 26, 29, 56, 59, 86, 89\}.$$

2. Select the sg tuples that have a value > 5 in the first column. Three tuples are selected: $\{83, 86, 89\}$.

$\diamond$

Algorithm 1 is inefficient in that it computes many irrelevant tuples: tuples such as gk that are not in the answer, and do not contribute to finding an answer tuple. We would like to rewrite program P so that the condition $X > 5$ is used to avoid computing irrelevant sg tuples.

Magic-sets Transformation for query Q
MST (Magic-sets transformation, Definition 4) does not lead to a better evaluation algorithm. Before applying MST, we must adorn query Q. The condition $X > 5$ does not bind X, so we generate the adorned query:

(A_1Q): ?- $X > 5$ & $sg^{ff}(X, Y)$.

suggesting that sg is to be evaluated without any restrictions. MST optimizes by passing restrictions suggested by the adorned program: if no restrictions are suggested, MST cannot optimize.

Target Evaluation
Even though magic-sets cannot optimize query Q, a more efficient execution strategy does exist. Algorithm 2 given below uses the condition $X > 5$ to limit computation of sg tuples.

Algorithm 2 We evaluate query Q in three steps:

1. Determine the set t of nodes reachable by following an `up` arc from nodes > 5.

   $(r1)$: $\quad t(U) \text{ :- } X > 5 \;\&\; up(X, U).$

   For the sample database of Figure 2, the small shaded area in Figure 3 represents set $t = \{c, d\}$.

2. Determine the set s of sg tuples with values c and d in the first column.

   $(r2)$: $\quad s(U, V) \text{ :- } t(U) \;\&\; sg(U, V).$

   MST can optimize the evaluation of s so that we don't need to compute the full sg relation. The subgoal t binds variable U, generating the adorned rule:

   $(S0)$: $\quad s^{ff}(U, V) \text{ :- } t(U) \;\&\; sg^{bf}(U, V).$

   Using the sips and adornments discussed in Section 2.7, we can adorn Program P for the head adornment sg^{bf} to get the following program S:

   $(S1)$: $\quad sg^{bf}(X, Y) \text{ :- } \mathtt{flat}(X, Y).$
   $(S2)$: $\quad sg^{bf}(X, Y) \text{ :- } \mathtt{up}(X, U) \;\&\; sg^{bf}(U, V) \;\&\; \mathtt{down}(V, Y).$

   MST on program S derives the magic program MS:

   $(MS0)$: $\quad s^{ff}(U, V) \text{ :- } t(U) \;\&\; sg^{bf}(U, V).$
   $(MS1)$: $\quad sg^{bf}(X, Y) \text{ :- } m\_sg^{bf}(X) \;\&\; \mathtt{flat}(X, Y).$
   $(MS2)$: $\quad sg^{bf}(X, Y) \text{ :- } m\_sg^{bf}(X) \;\&\; \mathtt{up}(X, U) \;\&\; sg^{bf}(U, V) \;\&\; \mathtt{down}(V, Y).$
   $(MS3)$: $\quad m\_sg^{bf}(U) \text{ :- } m\_sg^{bf}(X) \;\&\; \mathtt{up}(X, U).$
   $(MS4)$: $\quad m\_sg^{bf}(U) \text{ :- } t(U).$

   Figure 3 shows the evaluation of Program MS. The larger shaded area is the magic set $m\_sg^{bf} = \{c, d, h, i\}$. Rule $MS1$ computes the tuples $\{de, hj\}$ for sg^{bf}, and rule $MS2$ adds the tuple $\{cf\}$. Rule $MS0$ then assigns $s^{ff} = \{cf, de\}$.

3. Compute the answer to query Q. sg tuples that satisfy condition $X > 5$ and are computable from rule $P1$ can be derived by pushing the condition $X > 5$ into rule $P1$, as in rule $r3$:

   $(r3)$: $\quad sg(X, Y) \text{ :- } X > 5 \;\&\; \mathtt{flat}(X, Y).$

   sg tuples that satisfy condition $X > 5$ and are computable from rule $P2$ can be derived from an $s(u, v)$ tuple by following a `down` arc from v to y and an `up` arc in reverse from u to x, making sure that condition $x > 5$ is true.

   $(r4)$: $\quad sg(X, Y) \text{ :- } X > 5 \;\&\; \mathtt{up}(X, U) \;\&\; s(U, V) \;\&\; \mathtt{down}(V, Y).$

   In our example database, Rule $r3$ does not compute any tuple, whereas rule $r4$ computes $sg = \{83, 86, 89\}$, which is the answer.

$\diamond$

Ground Magic-sets Transformation

The evaluation of Algorithm 2 is passing the condition $X > 5$ into the sg goal of query Q.

When we tried to apply MST to Q, we first had to choose a sips and adorn Q. Our definition of sips and adornments (Section 2.7) allows us to pass bindings, but not conditions. Hence we were forced to choose a sips that does not pass the condition $X > 5$, and arrived at the adornment A_1Q. MST implements the chosen sips, and consequently cannot pass the condition.

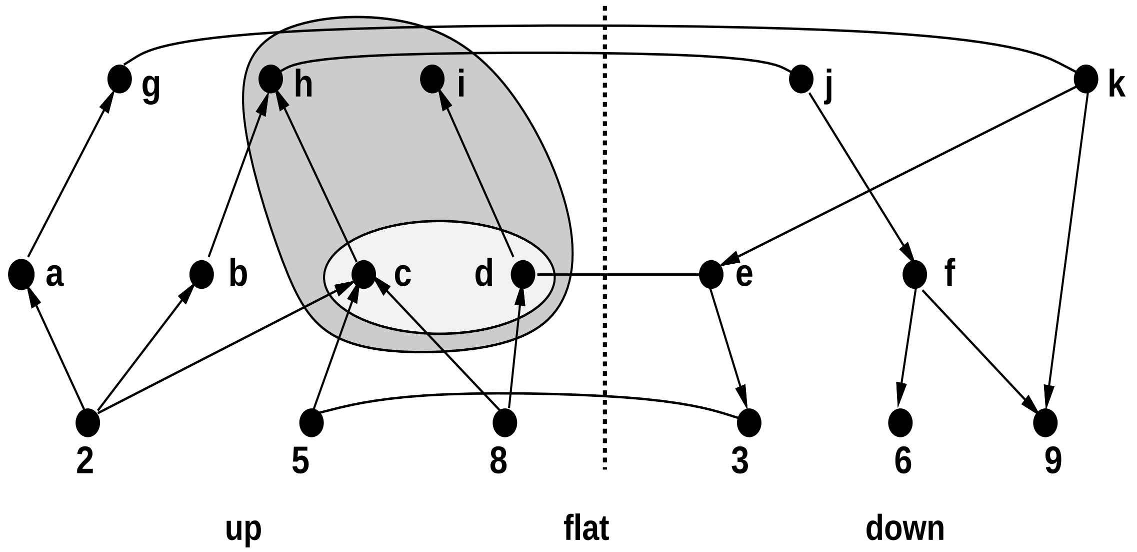

Figure 3: Evaluation of query Q using Algorithm 2.

Thus, to develop a rewrite optimization that achieves Algorithm 2, we first need a way to indicate that a condition is present on an argument, in much the same way as we indicate the presence of a binding. Then, we need a way to specify how conditions are to be passed in a rule. For example, we must be able to say that the condition $X > 5$ is to be passed into the subgoal sg, and that the first column of sg has a condition on it.

We introduce the c adornment to indicate the presence of a condition, and develop bcf sips to express how conditions as well as bindings are to be passed in a rule. We then present an algorithm that rewrites program P so that a bottom-up evaluation follows the execution steps of Algorithm 2. Our algorithm, called ground magic-sets transformation, or GMST, is a generalization of MST. Since conditions are not used to restrict a top-down evaluation, a program transformed by GMST will compute even fewer tuples than computed by a top-down evaluation.

4 The c Adornment

The b and f adornments of the bf adornment class allow us to distinguish between arguments that are bound, and those that are not. An argument restricted by a condition is not bound, and is therefore considered free. An argument that has no restriction is also not bound, and is considered free. Thus, using the bf adornment class, we cannot distinguish between arguments that are restricted by a condition, and those that have no restriction.

Let's consider a new c adornment. If an argument is restricted by a condition, we will indicate the restriction by a c adornment on the argument, and say that the argument is *conditioned.*

EXAMPLE 8 Consider query Q of Section 3. The condition $X > 5$ restricts the first argument sg. We therefore generate the adorned query AQ:

(AQ): ?- $X > 5$ & $sg^{cf}(X, Y)$.

The adornment "cf" indicates that the first argument of sg is conditioned, and the second argument is free.

We can adorn rules $P1$ and $P2$ for the head adornment sg^{cf}:

$(AP1)$: $sg^{cf}(X, Y)$:- $\mathtt{flat}^{cf}(X, Y)$.

$(AP2)$: $sg^{cf}(X,Y)$:- $\mathtt{up}^{cf}(X,U)$ & $sg^{bf}(U,V)$ & $\mathtt{down}(V,Y)$.

In rule $P1$, we pass the condition on X into the first argument of the `flat` subgoal, generating the adornment $\mathtt{flat}^{cf}$. In rule $P2$, we pass the condition on X into the first argument of the `up` subgoal, generating the adornment $\mathtt{up}^{cf}$. After solving $\mathtt{up}^{cf}$, we get a set of bindings for U, and pass these bindings into the sg subgoal.

Adorned rules for sg^{bf} are derived as discussed in Section 2.7.

$(AP3)$: $sg^{bf}(X,Y)$:- $\mathtt{flat}(X,Y)$.
$(AP4)$: $sg^{bf}(X,Y)$:- $\mathtt{up}(X,U)$ & $sg^{bf}(U,V)$ & $\mathtt{down}(V,Y)$.

Program AP has been adorned using b, c, and f adornments. Note that base relations are usually not adorned. The adornments on `flat` and `up` in this example are just to highlight the passing of conditions from the head. □

4.1 Rule Adornments

The variables of a rule can also be given b, c, or f adornments to indicate whether they are bound, conditioned, or free. Since variables get bound as subgoals containing the variables are evaluated, the adornment on variables changes as each subgoal is evaluated. An adornment on all variables of a rule, at a particular subgoal position, is called a *rule adornment*.

EXAMPLE 9 Consider rule $AP2$ in Example 8.

- Just before the first subgoal, $\mathtt{up}(X,U)$, X is conditioned due to the condition coming from the head, and all other variables are free. The rule adornment for (X,U,V,Y) is *cfff*. The condition on X is passed into the first argument of $\mathtt{up}(X,U)$ to generate $\mathtt{up}^{cf}(X,U)$.
- After evaluating $\mathtt{up}^{cf}(X,U)$, X and U get bound, leading to the rule adornment *bbff*. The binding on U is passed into $sg(U,V)$ to generate $sg^{bf}(U,V)$.
- After evaluating $sg^{bf}(U,V)$, V gets bound, leading to the rule adornment *bbbf*. The binding on V is passed into $\mathtt{down}(V,Y)$ to generate $\mathtt{down}^{bf}(V,Y)$.
- After evaluating $\mathtt{down}^{bf}(V,Y)$, Y gets bound, leading to the rule adornment *bbbb*.

□

4.2 Conditions and the *bcf* Adornment Class

Definition 5 — Condition and Independent Condition: A condition is a built-in predicate different from equality. A condition is independent with respect to a set of predicates if it refers to at most one variable that is not *limited* by the set of predicates. The independent condition is said to be *on* the non-*limited* variable, which is said to be independently conditioned. A *dependent* condition is a condition that is not independent. □

For example, the condition $X > 5$ is independent with respect to any set of predicates. Condition $X > Y$ is independent with respect to a set $\mathcal{P}$ of predicates if either X or Y is *limited* by the set $\mathcal{P}$, otherwise condition $X > Y$ is dependent.

We will use a new c adornment for variables that have an independent condition.

Definition 6 — *bcf* Adornment Class: The *bcf* adornment class is a system of adornments that distinguishes between bound (b adorned), independently conditioned (c adorned), and free (f adorned) argument positions.

	bf Adornment Class		*bcf* Adornment Class	
Adornment	Allowed Values	Any Disallowed Values?	Allowed Values	Any Disallowed Values?
b	Finite	Yes	Finite	Yes
c			Infinite	Yes
f	Infinite	No Information	Infinite	No Information

Table 1: Nature of restrictions due to b, c, and f adornments.

An argument of a subgoal in a rule is *bound* if one can guarantee that during a hypothetical, tuple at a time, top-down evaluation of the rule, the goal generated from this subgoal will have a constant in this argument position. The argument is *independently conditioned* if one can guarantee that during a hypothetical, tuple at a time, top-down evaluation of the rule, the goal generated from this subgoal will either have a constant in this argument position, or the variable X in this position will have an independent condition with respect to the set of ordinary subgoals evaluated before the current subgoal and the set of built-in subgoals in the rule. Otherwise, the argument is *free*. □

A more formal definition of *bf* and *bcf* adornment classes is given in [MPR94].

Note that Definition 6 gives the property that b, c, and f adornments must satisfy. It does not tell us how to determine the adornments. Under Definition 6, an argument of a subgoal s can get a b adornment if the argument is *limited* by the ordinary subgoals appearing before subgoal s in the subgoal order, and by the built-in subgoals in the rule. The argument can get a c adornment if the argument is *limited* by the ordinary subgoals appearing before subgoal s and by the built-in subgoals in the rule, or if the argument is independently conditioned by some built-in subgoal in the rule with respect to the ordinary subgoals appearing before subgoal s and the set of built-in subgoals in the rule. Otherwise the argument must get an f adornment. Whether arguments that are such restricted get a b or a c adornment depends upon the sips being used.

An adorned predicate that has at least one c adornment is said to be *conditioned*. An adorned predicate that does not have a c adornment, but has at least one b adornment, is said to be *bound*. Otherwise the adorned predicate is said to be free.

There is one important difference between the restrictions due to a condition and a binding. A binding defines a finite relation to which a bound variable is restricted. A condition may not define a finite relation. For example, the condition $X > 5$ over the domain of integers defines an infinite relation over X. Table 1 captures the nature of restrictions due to b, c, and f adornments, under the *bf* and *bcf* adornment classes, assuming an infinite underlying domain. The c adornment is seen to refine the f adornment of the *bf* adornment class. An argument with a restriction such as $X > 5$ that allows as well as disallows an infinite number of values is adorned f under the *bf* adornment class, but is adorned c under the *bcf* adornment class. For example, the first argument of sg is adorned f in query A_1Q (Page 10), but adorned c in query AQ (Example 8).

EXAMPLE 10 Consider the rule:

(r): $p^{ffff}(U, V, Y, Z)$:- $U > V$ & $X < 5$ & $Y < W$ & $q_1(U, V, W, X)$ & $q_2(W, X, Y, Z)$.

Let the subgoals be evaluated in the order shown.

- Before evaluating q_1, none of the variables are bound. The conditions $U > V$ and $Y < W$ are therefore dependent, and cannot generate a c adornment. The condition $X < 5$ is independent, so variable X can get the c adornment. The rule adornment for (U, V, W, X, Y, Z) is *fffcff*, and q_1 may be adorned q_1^{fffc}.

- After evaluating q_1, the variables U, V, W, and X are bound. The condition $Y < W$ that was dependent before W was bound, is now independent. It can therefore generate a c adornment for variable Y, leading to the rule adornment *bbbbcf*, and the possible adornment q_2^{bbcf} for q_2.

□

EXAMPLE 11 Consider the rules:

$(t1)$: $q1$:- $X \geq 5$ & $X \leq 5$ & $p(X)$.
$(t2)$: $q1$:- $X \geq 5$ & $X < 5$ & $p(X)$.

For both of these rules, p cannot be adorned $p^b(X)$, since we are not allowing inferences from the predicates in the rules (Section 2.1). In both cases, the adornment $p^c(X)$ can be generated. □

The decision not to use the c adornment for dependent conditions is a conscious one. As a consequence we loose the ability to represent and pass dependent conditions such as $U > V$ or $U = V$. For example, in Example 10 we would like to pass the dependent condition $U > V$ into q_1, but we were not able to do so. However, it is shown in ([Mum91], Chapter 3, [MPR94]), that limiting the c adornment to independent conditions buys us many desirable properties of adornments, such as making sure that the restrictions described by the adornment can be fully used for optimization. Mumick [Mum91, MPR94] discusses an adornment class that can be used to represent and pass dependent conditions.

5 The *bcf* SIPS

While deriving the adorned program A in Example 8, we passed the condition $X > 5$ in query:

(Q): ?- $X > 5$ & $sg(X, Y)$.

into the sg subgoal, generating the adorned query AQ (Page 12). Then, in rules $AP1$ and $AP2$ for sg^{cf} (Page 13), we passed the condition on X into the first subgoal.

We would like to be able to specify the passing of conditions in a rule, in a manner similar to the way *bf* sips specify passing of bindings. The specification can then be used to adorn and optimize the program to pass conditions as specified. Towards that goal, we define a class of sips, called the *bcf* sips, that can specify how conditions and bindings are to be passed in a rule. Following the definition of *bf* sips (Definition 3), the *bcf* sips for a rule is also defined to be a directed labeled graph. In a *bf* sips, each incoming edge into a literal represents bindings being passed into the literal, with the label listing the bound variables. In a *bcf* sips, each incoming edge will represent both bindings and conditions being passed into the literal. The edge needs two labels, a bound label, $\beta(e)$, listing the bound variables, and a condition label, $\kappa(e)$, listing the conditioned variables.

EXAMPLE 12 For query Q, the *bcf* sips specifying that the condition $X > 5$ is to be passed into $sg(X, Y)$ is given by the following edge $Q.e1$:

$(Q.e1)$: $\{X > 5\} \xrightarrow{\kappa=\{X\}} sg(X, Y)$.

For rule $P2$ (Page 9) and the head adornment *cf*, the *bcf* sips specifying that the condition from the head (represented below by $sg_h^{cf}(X)$) is to be passed into $\mathtt{up}(\mathtt{X}, \mathtt{U})$, that the resulting binding on U is to be passed into $sg(U, V)$ and that the resulting binding on V is to be passed into $\mathtt{down}(V, Y)$, is given by the following edges:

$(P2.e1)$: $\{sg_h^{cf}(X)\} \xrightarrow{\kappa=\{X\}} \mathtt{up}(X, U)$.
$(P2.e2)$: $\{sg_h^{cf}(X), \mathtt{up}(X, U)\} \xrightarrow{\beta=\{U\}} sg(U, V)$.
$(P2.e3)$: $\{sg_h^{cf}(X), \mathtt{up}(X, U), sg(U, V)\} \xrightarrow{\beta=\{V\}} \mathtt{down}(V, Y)$.

□

5.1 Formal Definition of the *bcf* sips

We define the *bcf* sips formally. First, we introduce a little notation. Consider the generic rule r:

(r): $s :- q_1 \ \& \ q_2 \ \& \ \ldots \ \& \ q_m \ \& \ c_1 \ \& \ c_2 \ \& \ \ldots \ \& \ c_n.$

where each q_i is an ordinary predicate, and each c_j is a built-in predicate of the form X op Y, or of the form X op c, where op is one of $\{<, \leq, \geq, >, \neq\}$, X and Y are variables, and c is a constant. Let α be a head adornment. We use the following notation:

- p_h^α is the special literal denoting the head predicate p restricted to arguments indicated *bound* or *conditioned* by adornment α. As an example, for rule $P2$ on Page 9, the special head literal is $sg_h^{cf}(X)$.
- $L(r, \alpha)$ is the set of all literals in the body of rule r, along with the special head literal p_h^α. That is, $L(r, \alpha) = \{p_h^\alpha, q_1, \ldots, q_m, c_1, \ldots, c_n\}$.

  For rule $P2$, $L(P2, cf) = \{sg_h^{cf}(X), \mathtt{up}(X, U), sg(U, V), \mathtt{down}(V, Y)\}$.
- $OL(r)$ is the set of all ordinary literals in rule r. That is, $OL(r) = \{q_1, \ldots, q_m\}$.

  For rule $P2$, $OL(P2) = \{\mathtt{up}(X, U), sg(U, V), \mathtt{down}(V, Y)\}$.
- $\mathsf{var}(u)$: For u equal to a rule, a literal, or a set of literals, $\mathsf{var}(u)$ denotes the set of variables that appear in u.
- $\mathsf{bvar}(u)$: For u equal to a literal or a set of literals, $\mathsf{bvar}(u)$ denotes the subset of $\mathsf{var}(u)$ that is *limited* by the literals in u. In other words, $\mathsf{bvar}(u)$ is the set of variables that can get the b adornment after u is evaluated. Note that $\mathsf{bvar}(p_h^\alpha)$ is equal to the variables in arguments of p that are indicated bound by adornment α.
- $\mathsf{cvar}(u)$: For u equal to a literal or a set of literals, $\mathsf{cvar}(u)$ denotes the subset of $\mathsf{var}(u)$ that is (1) not *limited* by the literals in u, and (2) is independently conditioned with respect to the literals in u. In other words, $\mathsf{cvar}(u)$ is the set of variables that can get the c adornment, but not the b adornment after u is evaluated. Note that $\mathsf{cvar}(p_h^\alpha)$ is equal to the variables in arguments of p that are indicated conditioned by adornment α, less any variables in $\mathsf{bvar}(p_h^\alpha)$.

bvar and cvar give the bound and conditioned variables of a rule as the rule's subgoals are evaluated. When the rule is invoked, the variables in the bound arguments of the head literal are bound, and the non-bound variables in the condition arguments of the head literal are conditioned. As an ordinary predicate is evaluated, all its variables become bound for subsequent subgoals. As built-in predicates are evaluated, an equality predicate of the form $X = t$, where t is a constant or a bound variable, binds variable X, an independent condition on variable X, listed in the built-in predicates, conditions variable X. Sometimes an independent condition, or even a binding can be deduced from two or more conditions, as from $Z > 10$ and $Z < 20$. We do not consider such inferences in this paper, though the discussion here applies equally well if the inference is encapsulated in procedures to compute bvar and cvar.

EXAMPLE 13 Consider rule $r1$:

$$(r1):\quad s(X,Y) :- X > 10 \ \& \ Z > 10 \ \& \ Z < 20 \ \& \ q_1(Z, X, U) \ \& \\ V < U \ \& \ q_2(T, V, W, Y) \ \& \ W < T.$$

- For head adornment *fb*, the special head literal is $s_h^{fb}(Y)$.
- $L(r1, fb) = \{s_h^{fb}(Y), X > 10, Z > 10, Z < 20, q_1(Z, X, U), V < U, q_2(T, V, W, Y), W < T\}$.
- $OL(r1) = \{q_1(Z, X, U), q_2(T, V, W, Y)\}$.

- $\mathsf{var}(r1) = \{T, U, V, W, X, Y, Z\}$.
  $\mathsf{var}(s_h^{fb}(Y)) = \{Y\}$.

- $\mathsf{bvar}(\{s_h^{fb}(Y), X > 10, Z > 10, Z < 20\}) = \{Y\}$. Y is bound by the special head literal. If Z is known to be an integer, a binding on Z can be deduced from the conditions $Z > 10$ and $Z < 20$, but is not used here.

  $\mathsf{bvar}(\{q_1(Z, X, U), V < U, W < T\}) = \{U, X, Z\}$.

- $\mathsf{cvar}(\{s_h^{fb}(Y), X > 10, Z > 10, Z < 20\}) = \{X, Z\}$.

  $\mathsf{cvar}(\{q_1(Z, X, U), V < U, W < T\}) = \{V\}$. The evaluation of $q_1(Z, X, U)$ binds variable U, so the condition $V < U$ is an independent condition on V.

□

Definition 7 — The *bcf* sips: The sips for a rule r and a head adornment α, denoted by $\mathsf{sips}(r, \alpha)$, is a directed labeled graph (V, E, B). The vertex set V has a node for each literal q in $OL(r)$, and a node for each subset T of $L(r, \alpha)$. Each edge e in E is of the form $T \rightarrow q$, where $T \subseteq L(r, \alpha)$ and $q \in OL(r)$, and indicates that information is passed from the literals in T to the ordinary body literal q. The label set B assigns two labels to each edge e: A bound label, $\beta(e)$, and a condition label, $\kappa(e)$. Each label is a set of variables. The intuition is that the variables of $\beta(e)$ are bound and the variables of $\kappa(e)$ are conditioned by the literals in T, and the sips specifies that these bindings and conditions are to be used in evaluating q. E and B must satisfy the following two constraints:

1. The edges of E induce a $\prec$ relation on $L(r, \alpha)$ as follows: If $T \rightarrow q$ is an edge in E, then for every literal $u \in T$, $u \prec q$.

   The $\prec$ relation induced by E must be a partial order.

2. For each edge e, $\beta(e) \subseteq \mathsf{bvar}(T)$, and $\kappa(e) \subseteq \mathsf{cvar}(T)$, $\beta(e) \subseteq \mathsf{var}(q)$, $\kappa(e) \subseteq \mathsf{var}(q)$, and $\beta(e) \cap \kappa(e) = \emptyset$.

□

$\mathsf{sips}(r, \alpha)$ suggests a computation for a rule r, assuming that bindings and conditions for the head arguments indicated bound and conditioned by the head adornment α are available. Each edge $T \rightarrow q_i$ implies that certain bindings and conditions must be passed from the literals in set T into the subgoal q_i, and used to restrict computation of q_i to only those tuples that satisfy the bindings and conditions. Such a computation requires that the literals of T be evaluated before evaluating subgoal q_i. The partial order on the edge induced $\prec$ relation required by Constraint 1 in Definition 7 guarantees that we can order the evaluation of the subgoals such that all bindings and conditions to be passed into subgoal q_i will be available when evaluating q_i. The partial order also restricts the possible subgoal orders that can be chosen. Constraint 2 in Definition 7 is a sanity check to make sure that the bindings and conditions being passed from set T into subgoal q_i are computable from set T, refer to subgoal q_i, and don't imply both a condition and a binding on the same variable.

A sips can have multiple edges into subgoal q_i. In such a case, the bindings and conditions passed along each edge must be joined together to form one set of bindings and conditions.

EXAMPLE 14 In Example 12, we listed the edges and labels for one choice of a *bcf* sips for query Q and rule $P2$ of the same generation program. The $\prec$ relation derived by $\mathsf{sips}(P2, cf)$:

$$sg_h^{cf}(X) \prec \mathtt{up}(X, U) \prec sg(U, V) \prec \mathtt{down}(V, Y).$$

is a total order on the subgoals of $P2$, specifying an exact order of evaluation. □

EXAMPLE 15 For rule $r1$ of Example 13, a possible $\mathsf{sips}(r1, fb)$ is given by the edges:

$(r1.e1)$: $\{s_h^{fb}(Y), X > 10, Z > 10, Z < 20\} \overset{\beta=\{\},\kappa=\{X,Z\}}{\longrightarrow} q_1(Z, X, U)$.

$(r1.e2)$: $\{s_h^{fb}(Y), X > 10, Z > 10, Z < 20, q_1(Z, X, U), V < U\}$
$\overset{\beta=\{Y\},\kappa=\{V\}}{\longrightarrow} q_2(T, V, W, Y)$.

□

5.2 Full *bcf* sips

A full *bcf* sips is a *bcf* sips that passes all available bindings and independent conditions into each subgoal q. To determine a full *bcf* sips, we need to determine a join order for the ordinary subgoals, and we need to determine a position for each condition.

Definition 8 — $\mathsf{eligibility}$**:** Given an ordering $q_1, \ldots, q_m$ of the ordinary subgoals in a rule

(r): p :- q_1 & q_2 & ... & q_m & c_1 & c_2 & ... & c_n.

the $\mathsf{eligibility}(c)$ of a condition $c \in \{c_1, \ldots, c_n\}$ is the earliest position where the condition can be usefully "applied". Formally, $\mathsf{eligibility}$ is the smallest j such that one of the following two conditions holds:

1. Condition c can be tested after the subgoal q_j, that is,

$$\mathsf{var}(c) \subseteq \mathsf{bvar}(\{p_h^\alpha, q_1, \ldots, q_j\} \cup C(j)),$$

   where $C(j)$ is the set of conditions each with eligibility $\leq j$. The condition requires all variables of c to be bound by subgoals and conditions placed before subgoal q_{j+1}.

2. Condition c binds or independently conditions a variable in subgoal q_{j+1}; that is, if

$$\mathcal{B} = \mathsf{var}(c) - \mathsf{bvar}(\{p_h^\alpha, q_1, \ldots, q_j\} \cup C(j)),$$

   then $\mathcal{B}$ must have only one variable X that must appear in subgoal q_{j+1}.

□

If $\mathsf{eligibility}(c) = j$, condition c should be placed after subgoal q_j in rule r. (If $j = 0$, the condition should be placed before q_1.) In the sips derived from this placement, condition c will be used to pass information into subgoal q_{j+1}, and into every subsequent subgoal. Let $C(j)$ be the set of conditions with eligibility j. Then, in a full *bcf* sips the order of literals in the body of rule r is:

(r): p :- $C(0)$ & q_1 & $C(1)$ & q_2 & ... & $C(m-1)$ & q_m & $C(m)$.

EXAMPLE 16 In rule $r1$ of Example 13,

- $\mathsf{eligibility}(X > 10) = \mathsf{eligibility}(Z > 10) = \mathsf{eligibility}(Z < 20) = 0$, since all of these three conditions can be passed into q_1.
- $\mathsf{eligibility}(V < U) = 1$, since $V < U$ is an independent condition on V after evaluating q_1 and can be passed into q_2.
- $\mathsf{eligibility}(W < T) = 2$, since condition $W < T$ can be tested after evaluating q_2, but cannot be passed into either q_1 or q_2.
- $C(0) = \{X > 10, Z > 10, Z < 20\}$, $C(1) = \{V < U\}$, and $C(2) = \{W < T\}$.

□

Definition 9 — **Full** *bcf* **sips:** The $\mathsf{sips}(r, a) = (V, E, B)$ is a *full bcf* sips if the following conditions hold:

1. The relation $\prec$ induced by E on $OL(r)$ is a total order. Let the total order be $q_1, \ldots, q_m$.
2. For every literal $q_j \in OL(r)$, there is exactly one edge of the form $T \rightarrow q_j$ in E.
3. For every edge $T \rightarrow q_j$ in E,
   - $p_h^a \in T$. (The head passes information into q_j.)
   - $(\forall k)_{0<k<j}(q_k \in T)$. (If q_k precedes q_j in the total order, then q_k passes information into q_j.)
   - If c is a condition with $\mathsf{eligibility}(c) < j$, then $c \in T$. (Every condition eligible before q_j passes information into q_j.)
   - $\beta(e) = \mathsf{bvar}(T) \cap \mathsf{var}(q)$, and $\kappa(e) = \mathsf{cvar}(T) \cap \mathsf{var}(q)$. (All available bindings and conditions are passed into q_j.)

□

6 Adorning Programs with the *bcf* Adornment Class

We present an algorithm to adorn a program for a given query using *bcf* sips and the *bcf* adornment class.

Given $\mathsf{sips}(r, \alpha) = (V, E, B)$ for a rule r, it is straightforward to adorn the rule for the head adornment α. For each ordinary subgoal, q, compute the two sets of variables passed into q: the bound set $\mathcal{B} = \bigcup_{(T \rightarrow q) \in E} \beta(e)$, and the conditioned set $\mathcal{C} = (\bigcup_{(T \rightarrow q) \in E} \kappa(e)) - \mathcal{B}$. The k^{th} argument of q is adorned b if it is a constant or a bound variable, adorned c if it is a conditioned variable, and adorned f otherwise.

We present an algorithm that, given a rule and a head adornment, determines a full *bcf* sips and adorns the rule. Using this algorithm as a subroutine, we develop the algorithm to adorn a program for a given query, deriving full sips for the rules for each head adornment used in the program.

Morris [Mor88] also discusses an algorithm to adorn a program for a given query, determining the sips for each rule. Her work uses *bf* sips, and ignores conditions. The algorithms presented here are based on the work by Morris.

6.1 Adorning a Rule

Given a Datalog rule r and and a head adornment α, Algorithm 3 determines a full *bcf* sips $\mathsf{sips}(r, \alpha)$, and adorns rule r according to this sips.

Algorithm 3 starts with set of bound and conditioned variables coming from the head, uses these to find the subgoal u_1 that should be evaluated first, and then finds the conditions eligible at position 0. Having fixed the first subgoal u_1 and the conditions that go before u_1, the algorithm determines the set of bound and conditioned variables after u_1 is evaluated, uses these to find the subgoal u_2 that should be evaluated second, and then finds the conditions eligible at position 1. The third, fourth, ..., m^{th} subgoals are found in a similar manner. Edges of $\mathsf{sips}(r, \alpha)$ and their labels are built as the position of a subgoal is fixed, and the labels are used to adorn the subgoal.

Before finding subgoal u_j, the variables of Algorithm 3 have the following interpretation: C is the set of conditions eligible at or before position $(j-2)$. $\mathcal{B}$ and $\mathcal{C}$ are the bound and conditioned variables after evaluating subgoals $u_1, \ldots, u_{j-1}$. After finding subgoal u_j, C is extended with $C(j-1)$, the set of conditions eligible at position $j-1$, and $\mathcal{B}$ and $\mathcal{C}$ are extended with variables bound and conditioned by conditions $C(j-1)$ and subgoal u_j.

Algorithm 3 uses a next function to find out which ordinary subgoal should be evaluated next. Given a rule r, the first $j-1$ subgoals $(u_1, \ldots, u_{j-1})$, the conditions C placed before u_{j-1}, and the sets of bound and conditioned variables after evaluating u_{j-1}, next determines the subgoal u_j to be evaluated at position j. The next function corresponds to determining a join order, and is not described in this paper. A simple

Algorithm 3 **(Find a full sips and adorn a rule.)**

Input:

1. A Datalog rule r of the form:
   (r): $\quad p$:- q_1 & q_2 & ... & q_m & c_1 & c_2 & ... & c_n.
   where each q_j is an ordinary subgoal, and each c_i is a condition.
2. A head adornment p^α.

Output:

A full sips, $\mathsf{sips}(r,\alpha) = (V, E, B)$, and an adorned rule ar.

Method:

% Step 1: Initialize
$C = \emptyset$. *% Set of conditions*
$\mathcal{B} = \mathsf{bvar}(p_h^\alpha)$; $\mathcal{C} = \mathsf{cvar}(p_h^\alpha)$; *% Set of bound and conditioned variables*
$V = \{q_1, \ldots, q_m\}$; $E = B = \emptyset$; *% Initialize* $\mathsf{sips}(r,\alpha)$
for $(j = 1$ **to** $m)$
  {*% Step 2.A: Determine the* j^{th} *subgoal*
  $u_j = \mathsf{next}(r, (u_1, \ldots, u_{j-1}), C, \mathcal{B}, \mathcal{C})$; *% Find next subgoal*

  % Step 2.B: Derive $C(j-1)$*; propagate effect to* C*,* $\mathcal{B}$*, and* $\mathcal{C}$
  $C(j-1) = \{c_i \mid \mathsf{eligibility}(c_i) = j-1\}$; *% Conditions eligible at position* $j-1$
  $C = C \cup C(j-1)$; *% Conditions eligible at or before position* $j-1$
  $\mathcal{B} = \mathsf{bvar}(\{p_h^\alpha, C, u_1, u_2, \ldots, u_{j-1}\})$; *% bound variables before subgoal* u_j
  $\mathcal{C} = \mathsf{cvar}(\{p_h^\alpha, C, u_1, u_2, \ldots, u_{j-1}\})$; *% conditioned variables before subgoal* u_j

  % Step 2.C: Extend $\mathsf{sips}(r,\alpha)$ *to record information passed into* u_j
  Insert the edge $e = (\{p_h^\alpha, C, u_1, \ldots, u_{j-1}\} \rightarrow u_j)$ into the edge set E;
  Insert the node $\{p_h^\alpha, C, u_1, \ldots, u_{j-1}\}$ into the vertex set V;
  $\beta(e) = \mathcal{B} \cap \mathsf{var}(u_j)$; $\kappa(e) = \mathcal{C} \cap \mathsf{var}(u_j)$; *% labels for edge* e
  % Step 2.D: Adorn u_j
  Compute adornment α_j for u_j due to the bound and condition labels $\beta(e)$ and $\kappa(e)$;

  % Step 2.E: Determine $\mathcal{B}$ *and* $\mathcal{C}$ *for the next subgoal*
  $\mathcal{B} = \mathcal{B} \cup \mathsf{var}(u_j)$; *% As* u_j *is evaluated, its variables get bound*
  $\mathcal{C} = \mathcal{C} - \mathsf{var}(u_j)$;
  }

% Step 3: Determine conditions evaluable only after the last subgoal
$C(m) = \{c_i \mid \mathsf{eligibility}(c_i) = m\}$;

The adorned rule ar is:
  (ar): $\quad p^\alpha$:- $C(0)$ & $u_1^{\alpha_1}$ & $C(1)$ & $u_2^{\alpha_2}$ & ... & $C(m-1)$ & $u_m^{\alpha_m}$ & $C(m)$.

◇

implementation of next could be a join ordering algorithm that determines the order of ordinary subgoals before Algorithm 3 starts, and then simply returns the subgoals in order upon each invocation.

EXAMPLE 17 We will apply Algorithm 3 on rule $r1$ for the head adornment s^{fb}.

$$\begin{array}{ll}(r1): & s(X,Y) :- X > 10 \ \& \ Z > 10 \ \& \ Z < 20 \ \& \ q_1(Z,X,U) \ \& \\ & \qquad V < U \ \& \ q_2(T,V,W,Y) \ \& \ W < T.\end{array}$$

Initialization yields $\mathcal{B} = \{Y\}$ and $\mathcal{C} = \emptyset$.

The function next has two choices for the first subgoal: q_1 or q_2. If next chooses q_1, conditions $\{X > 10, Z > 10, Z < 20\}$ will be eligible, so that the adornment will be q_1^{ccf}. If next chooses q_2, no condition is eligible, but the head provides a binding for Y, so that the adornment will be q_2^{fffb}. Let us assume that next uses the *bound-is-easier* heuristic and selects q_1 as the first subgoal.

Algorithm 3 now computes $C(0) = \{X > 10, Z > 10, Z < 20\}$, $\mathcal{B} = \{Y\}$, and $\mathcal{C} = \{X, Z\}$. The edge $r1.e1$ of Example 15 is added to $\mathsf{sips}(r1, fb)$, and the edge labels derive the adornment q_1^{ccf}. For the next iteration, $\mathcal{B} = \{U, X, Y, Z\}$ and $\mathcal{C} = \emptyset$.

In the second and last iteration, next returns q_2, the only remaining subgoal, Algorithm 3 computes $C(1) = \{V < U\}$, $\mathcal{B} = \{U, X, Y, Z\}$, and $\mathcal{C} = \{V\}$. The edge $r1.e2$ of Example 15 is added to $\mathsf{sips}(r1, fb)$, and the edge labels derive the adornment q_2^{fcfb}.

The final adorned rule $ar1$ is:

$$\begin{array}{ll}(ar1): & s^{fb}(X,Y) :- X > 10 \ \& \ Z > 10 \ \& \ Z < 20 \ \& \ q_1^{ccf}(Z,X,U) \ \& \\ & \qquad V < U \ \& \ q_2^{fcfb}(T,V,W,Y) \ \& \ W < T.\end{array}$$

□

6.2 Adorning a Program

Definition 10 — Expansion of an adorned predicate: Given an adorned predicate p^α, the process of adorning all rules for p (that is, rules with head predicate p) for the head adornment p^α is called the expansion of p^α. □

Given a Datalog program P and a query Q, Algorithm 4 adorns the (program, query) pair. Algorithm 4 uses Algorithm 3 to select sips and adorn individual rules.

The set new_preds contains unexpanded adorned predicates for which adorned subgoals have been generated. The set expanded_preds keeps track of all adorned predicates that have been expanded. Initially, new_preds contains the adorned query goal $p^{\bar{f}}$. Each step of Algorithm 4 involves picking up a new adorned predicate q^α from new_preds and expanding it. Algorithm 3 is used to adorn each rule r for predicate q. If a new, previously unexpanded adorned subgoal is created in the adorned rule ar, we add it to the set new_preds for later expansion.

EXAMPLE 18 Consider the same generation program P on Page 9, extended with rule $P3$ defining the answer tuples to query Q of Page 9:

$(P1)$: $sg(X,Y) :- \mathtt{flat}(X,Y)$.
$(P2)$: $sg(X,Y) :- \mathtt{up}(X,U) \ \& \ sg(U,V) \ \& \ \mathtt{down}(V,Y)$.
$(P3)$: $sg\_ans(X,Y) :- X > 5 \ \& \ sg(X,Y)$.

The query:

$(P0)$: ?- $sg\_ans(X,Y)$.

is adorned as

Algorithm 4 (Adorn a program for a given query.)

Input:
A Datalog program P and a query Q of the form:
(Q): ?- $p(\bar{X})$.
Output:
An adorned query AQ, an adorned program AP, and a sips for each adorned rule.

Method:
% *Step 1: Derive AQ*
The adorned query is:
(AQ): ?- $p^{\bar{f}}(\bar{X})$.

% *Step 2: Compute AP*
% *Step 2.A: Initialize*
new_preds $= \{p^{\bar{f}}\}$; % *Set of new adorned predicates*
expanded_preds $= \emptyset$; % *Set of expanded adorned predicates*

% *Step 2.B: Expand adorned predicates*
repeat
{% *Step 2.B.1: Pick an element of* new_preds
Choose some q^α from new_preds;
new_preds = new_preds $- \{q^\alpha\}$;
expanded_preds = expanded_preds $\cup \{q^\alpha\}$;

% *Step 2.B.2: Expand adorned predicate* q^α
for (each rule r with head predicate q) % *Adorn r for adornment α*
{Use Algorithm 3 to select a sips and adorn rule r for head adornment q^α;
for (each adorned subgoal u^γ in the adorned rule ar)
if ($u^\gamma \notin$ new_preds $\wedge\ u^\gamma \notin$ expanded_preds)
new_preds = new_preds $\cup \{u^\gamma\}$; % *u^γ is a new predicate to be expanded later*
}
}
until (new_preds $= \emptyset$) % *All adorned predicates have been expanded*
◇

$(AP0)$: ?- $sg\_ans^{ff}(X, Y)$.

Algorithm 4 then starts with new_preds $= \{sg\_ans^{ff}\}$ and expanded_preds $= \emptyset$. The subsequent steps are:

1. Expand sg_ans^{ff}, generating adorned rule $AP5$:

   $(AP5)$: $sg\_ans^{ff}(X, Y)$:- $X > 5$ & $sg^{cf}(X, Y)$.

   new_preds $= \{sg^{cf}\}$, expanded_preds $= \{sg\_ans^{ff}\}$.

2. Expand sg^{cf}, generating adorned rules $AP1$ and $AP2$ of Example 8.
   new_preds $= \{sg^{bf}\}$, expanded_preds $= \{sg\_ans^{ff}, sg^{cf}\}$.

3. Expand sg^{bf}, generating adorned rules $AP3$ and $AP4$ of Example 8.
   new_preds $= \emptyset$, expanded_preds $= \{sg\_ans^{ff}, sg^{cf}, sg^{bf}\}$.

4. Terminate, as there is no adorned predicate to expand.

□

Lemma 1 *For every Datalog program P and for every query Q of the form ?- $p(\bar{X})$, Algorithm 4 terminates, and generates an adorned program query pair (AP, AQ) that is query equivalent to (P, Q).* □

Proof:
Termination: The outer repeat-until loop in Algorithm 4 is the only potentially infinite loop. However, note that a predicate q of arity k can have at most 3^k adornments. The number of predicates in program P is finite, say N. Assuming K is the largest arity of any predicate in program P, the adorned program AP has less than $N3^K$ adorned predicates. Each iteration of the repeat-until loop processes one adorned predicate from new_preds, and the processed predicate is never returned to new_preds. The repeat-until loop would thus terminate in fewer than $N3^K$ iterations.

Correctness: Each rule in program AP is identical to a rule in program P if we drop the adornments. Thus, each adorned fact derived in program AP is derivable as the unadorned fact in program P.

An induction on the height of derivation trees can show that if a fact $q(\bar{a})$ is derived in program P and if q^α is an adorned predicate in program AP, then $q^\alpha(\bar{a})$ is derived in program AP. Since $p^{\bar{f}}$ must be present in program AP, every p fact derived by program P is derived as an $p^{\bar{f}}$ fact by program AP. Beeri and Ramakrishnan [BR87] give details of the induction in the proof for correctness of the bf adornment algorithm. □

7 Ground Magic-sets Transformation: An Overview

A sips specifies a strategy for passing information in a rule: The sips does not pass the information. There are two major ways of passing the information.

- Use an evaluation method that looks at the sips and passes information as specified. For example, the top-down evaluation methods pass bindings from one subgoal to another as they evaluate the rules in a left to right order.

- Rewrite an adorned program so that a subsequent evaluation is guaranteed to pass information as specified by the sips. A bottom-up evaluation is usually used after the rewrite. The Magic-sets Transformation is a general technique to do such a rewrite when the information to be passed consists of bindings.

We define a transformation to rewrite an adorned program so that a subsequent bottom-up evaluation passes conditions and bindings as specified by a given bcf sips for each rule of the given adorned program. Our transformation is called "Ground Magic-sets Transformation," GMST for short.

7.1 Can the Magic-sets Transformation pass conditions?

To help appreciate GMST, let us try to use MST to pass conditions. Given a bcf adorned program, we can treat the conditions as if they are bindings (so that the condition predicate $X > 5$ will be passed in a manner similar to the equality predicate $X = 5$), and apply MST. Example 19 illustrates this approach:

EXAMPLE 19 Consider the adorned same generation program AP listed in Example 8. Treating the condition $X > 5$ as a binding, an application of MST leads to the magic program MP:

(AQ): ?- $X > 5$ & $sg^{cf}(X, Y)$.
$(MP1)$: $sg^{cf}(X, Y)$:– $m\_sg^{cf}(X)$ & $\mathtt{flat}(X, Y)$.
$(MP2)$: $sg^{cf}(X, Y)$:– $m\_sg^{cf}(X)$ & $\mathtt{up}(X, U)$ & $sg^{bf}(U, V)$ & $\mathtt{down}(V, Y)$.
$(MP3)$: $sg^{bf}(X, Y)$:– $m\_sg^{bf}(X)$ & $\mathtt{flat}(X, Y)$.
$(MP4)$: $sg^{bf}(X, Y)$:– $m\_sg^{bf}(X)$ & $\mathtt{up}(X, U)$ & $sg^{bf}(U, V)$ & $\mathtt{down}(V, Y)$.
$(MP5)$: $m\_sg^{cf}(X)$:– $X > 5$.
$(MP6)$: $m\_sg^{bf}(U)$:– $m\_sg^{cf}(X)$ & $\mathtt{up}(X, U)$.
$(MP7)$: $m\_sg^{bf}(U)$:– $m\_sg^{bf}(X)$ & $\mathtt{up}(X, U)$.

The query AQ passes the "binding" (actually a condition) $X > 5$ into sg^{cf}, and rule $MP5$ collects the "binding" in the magic predicate m_sg^{cf}. Bindings passed into sg^{bf} from rules $AP2$ and $AP4$ (Example 8) are collected by rules $MP6$ and $MP7$ respectively.

Program MP succeeds in passing the "binding" $X > 5$ into the evaluation of sg. In Section 3 we gave the target Algorithm 2 for the evaluation of query Q. The set t computed in Step 1 of Algorithm 2 is the same as the subset of m_sg^{bf} computed by rule $MP6$ after substituting for m_sg^{cf} computed by rule $MP5$. Rules $MP3$, $MP4$, and $MP7$ then mimic the computation of Program MS in Step 2 of Algorithm 2, while rules $MP1$ and $MP2$ do the job of rules $r3$ and $r4$ in Step 3 of Algorithm 2.

However, there is a problem with program MP. We derived MP assuming that the condition $X > 5$ is a binding, and thereby defines a finite relation. However, $X > 5$ is not a binding, and actually defines an infinite relation m_sg^{cf} in rule $MP5$. Rule $MP5$ is not range-restricted (Definition 1), and is not allowed in a Datalog program. □

An attempt to pass conditions using MST leads to magic rules that are not range-restricted. MST is thus not appropriate for passing conditions.

7.2 GMST Example

GMST can be best explained as a two step transformation.

- **Step 1:** Treat the conditions as bindings and apply the magic-sets transformation, as we did in Section 7.1. The resulting program will have non-range-restricted magic rules.

- **Step 2:** Make each non-range-restricted magic rule range-restricted. This is called the *grounding* step, since each non-*limited* variable of a non-range-restricted rule is *grounded* to a finite domain (that is, the variable is *limited*) by adding an ordinary subgoal to the body.

EXAMPLE 20 Consider the adorned same generation program AP derived in Example 8. Step 1 of GMST produces the non-range-restricted magic program MP listed in Example 19.

The magic rule $MP5$ for the magic predicate m_sg^{cf} is the only non-range-restricted rule in program MP. This rule is to be ground in Step 2.

The relation m_sg^{cf} has an infinite number of tuples (every value greater than 5), but only a finite number of these tuples are ever needed during an evaluation of program MP. m_sg^{cf} is used in rules $MP1$,

$MP2$, and $MP6$, and each of these rules joins $m\_sg^{cf}(X)$ with a finite relation, $\texttt{flat}(X,Y)$, or $\texttt{up}(X,U)$. We note that rule $MP6$ is a magic rule that collects bindings for sg^{bf} from rule $MP2$. The usage of m_sg^{cf} in rules $MP2$ and $MP6$ is thus correlated; in fact exactly the same values of m_sg^{cf} are used in both rules $MP2$ and $MP6$.

Consequently, we only need to compute the finite subset of m_sg^{cf} that is used in rule $MP1$ or in rule $MP2$. Rules $IP5.1$ and $IP5.2$ define these subsets:

$(IP5.1)$: $m\_sg^{cf}(X)$:- $X > 5$ & $\texttt{flat}(X,Y)$.
$(IP5.2)$: $m\_sg^{cf}(X)$:- $X > 5$ & $\texttt{up}(X,U)$.

$\texttt{flat}(X,Y)$ and $\texttt{up}(X,U)$ are the *limiting relations* for X. Rules $IP5.1$ and $IP5.2$ are derived from the non-range-restricted rule $MP5$ by reducing the magic set to the tuples matching the limiting relations. We refer to the limiting relations as *grounding subgoals*.

A word about our convention for the numbers of restricted magic rules. The first number is inherited from the magic rule being *limited*. For example, both rules $IP5.1$ and $IP5.2$ are derived by restricting the magic rule $MP5$. The second number is obtained from the rule providing the limiting relations. For example, rule $IP5.1$ is derived using a limiting relation from rule $MP1$, while rule $IP5.2$ is derived using a limiting relation from rule $MP2$.

Copying the other rules from program MP, we derive the ground program IP:

(AQ): ?- $X > 5$ & $sg^{cf}(X,Y)$.
$(IP1)$: $sg^{cf}(X,Y)$:- $m\_sg^{cf}(X)$ & $\texttt{flat}(X,Y)$.
$(IP2)$: $sg^{cf}(X,Y)$:- $m\_sg^{cf}(X)$ & $\texttt{up}(X,U)$ & $sg^{bf}(U,V)$ & $\texttt{down}(V,Y)$.
$(IP3)$: $sg^{bf}(X,Y)$:- $m\_sg^{bf}(X)$ & $\texttt{flat}(X,Y)$.
$(IP4)$: $sg^{bf}(X,Y)$:- $m\_sg^{bf}(X)$ & $\texttt{up}(X,U)$ & $sg^{bf}(U,V)$ & $\texttt{down}(V,Y)$.
$(IP5.1)$: $m\_sg^{cf}(X)$:- $X > 5$ & $\texttt{flat}(X,Y)$.
$(IP5.2)$: $m\_sg^{cf}(X)$:- $X > 5$ & $\texttt{up}(X,U)$.
$(IP6)$: $m\_sg^{bf}(U)$:- $m\_sg^{cf}(X)$ & $\texttt{up}(X,U)$.
$(IP7)$: $m\_sg^{bf}(U)$:- $m\_sg^{bf}(X)$ & $\texttt{up}(X,U)$.

Program IP can be further optimized. In rule $IP5.1$ we compute a subset of the X values in $\texttt{flat}(X,Y)$, and then in rule $IP1$ we match the X values with the Y values. We could instead carry the matching Y values from rule $IP5.1$ itself, as done in rule $GP5.1$:

$(GP5.1)$: $s_1(X,Y)$:- $X > 5$ & $\texttt{flat}(X,Y)$.

Predicate s_1 can be used to simplify rule $IP1$:

$(GP1)$: $sg^{cf}(X,Y)$:- $s_1(X,Y)$.

Similarly, as we select the interesting subset of X values from $\texttt{up}(X,U)$ in rule $IP5.2$, we can carry the matching U values along to avoid a later join with $\texttt{up}(X,U)$ in rules $IP2$ and $IP6$. The rules are rewritten as:

$(GP5.2)$: $s_2(X,U)$:- $X > 5$ & $\texttt{up}(X,U)$.
$(GP2)$: $sg^{cf}(X,Y)$:- $s_2(X,U)$ & $sg^{bf}(U,V)$ & $\texttt{down}(V,Y)$.
$(GP6)$: $m\_sg^{bf}(U)$:- $s_2(X,U)$.

The other rules are copied from program IP to derive the final ground magic program GP:

(AQ): ?- $X > 5$ & $sg^{cf}(X,Y)$.
$(GP1)$: $sg^{cf}(X,Y)$:- $s_1(X,Y)$.
$(GP2)$: $sg^{cf}(X,Y)$:- $s_2(X,U)$ & $sg^{bf}(U,V)$ & $\texttt{down}(V,Y)$.

$(GP3)$: $sg^{bf}(X,Y)$:- $m\_sg^{bf}(X)$ & $\texttt{flat}(X,Y)$.
$(GP4)$: $sg^{bf}(X,Y)$:- $m\_sg^{bf}(X)$ & $\texttt{up}(X,U)$ & $sg^{bf}(U,V)$ & $\texttt{down}(V,Y)$.
$(GP5.1)$: $s_1(X,Y)$:- $X > 5$ & $\texttt{flat}(X,Y)$.
$(GP5.2)$: $s_2(X,U)$:- $X > 5$ & $\texttt{up}(X,U)$.
$(GP6)$: $m\_sg^{bf}(U)$:- $s_2(X,U)$.
$(GP7)$: $m\_sg^{bf}(U)$:- $m\_sg^{bf}(X)$ & $\texttt{up}(X,U)$.
□

Predicates s_1 and s_2 can be identified as the supplementary predicates for rules $MP1$ and $MP2$ after evaluating the first subgoal of these rules. GMST can thus be viewed as a hybrid of MST and SMST, avoiding computation of an infinite magic set by directly computing supplementary magic sets that are guaranteed to be finite.

7.3 Grounding Phase

Step 2 of GMST works as follows: given a rule r for a conditioned predicate p^α (a predicate p^α is said to be conditioned if its adornment string α has at least one c adornment), and a non-range-restricted magic rule m for the magic predicate m_p^α:
(r): p^α :- m_p^α & G & NG.
(m): m_p^α :- B & C.
where G and NG are sets of subgoals of rule r and C is a set of conditions for the conditioned arguments of p^α, we can rewrite rules r and m as:

(gr): p^α :- s_r & NG.
(gm): s_r :- B & C & G.

If we choose the set G such that the subgoals in G limit all the conditioned variables of the head p^α, then the conditions C in rule gm will be grounded by subgoals in G, and rule gm will be range-restricted. If the initial rule r is range-restricted, such a set G of subgoals must exist.

The *conditioned variables* of a rule are all the variables that appear in a c adorned argument of the rule head. Subgoals of a rule that limit conditioned variables will be called *grounding* subgoals. As is evident from the rewrite of rules r and m, GMST works by moving the grounding subgoals out of the original rule, and into a magic rule. If a subgoal g_1 does not limit any conditioned variable, but passes information into a limiting subgoal g_2 (according to the sips selected), we will need to move g_1 also into the magic rule if we want to continue passing information from g_1 to g_2. Hence, any subgoal that passes information into a grounding subgoal is also a grounding subgoal.

The set G of grounding subgoal is called the grounding set. The grounding set is a function of the sips chosen for a rule. The remaining *nongrounding* subgoals form the nongrounding set NG.

8 GMST Applicability

In this section we will state a condition for GMST to be applicable on a program. We motivate the condition through an example:

EXAMPLE 21 Consider the adorned query Q on program T:

(Q): ?- $u(W)$ & $Y > W$ & $t^{bc}(W,Y)$.
$(T1)$: $t^{bc}(X,Y)$:- $v(X,Y)$.
$(T2)$: $t^{bc}(X,Y)$:- $q(X,Z)$ & $t^{bc}(Z,Y)$.

Program T defines a path of the form q^*v in a graph. Step 1 of GMST lead to the non-range-restricted program MT:

$(MT1)$: $t^{bc}(X,Y)$:- $m\_t^{bc}(X,Y)$ & $v(X,Y)$.
$(MT2)$: $t^{bc}(X,Y)$:- $m\_t^{bc}(X,Y)$ & $q(X,Z)$ & $t^{bc}(Z,Y)$.
$(MT3)$: $m\_t^{bc}(W,Y)$:- $u(W)$ & $Y > W$.
$(MT4)$: $m\_t^{bc}(Z,Y)$:- $m\_t^{bc}(X,Y)$ & $q(X,Z)$.

Rule $MT3$ is the only non-range-restricted rule. The subgoal $v(X,Y)$ grounds the second argument of predicate t in rule $T1$, and the subgoal $t^{bc}(Z,Y)$ grounds the second argument of predicate t in rule $T2$. In rule $T2$, subgoal $q(X,Z)$ passes information into the grounding subgoal $t^{bc}(Z,Y)$: hence $q(X,Z)$ is also a grounding subgoal. Grounding rule $MT3$, and copying the remaining rules, leads to program IT.

$(IT1)$: $t^{bc}(X,Y)$:- $m\_t^{bc}(X,Y)$ & $v(X,Y)$.
$(IT2)$: $t^{bc}(X,Y)$:- $m\_t^{bc}(X,Y)$ & $q(X,Z)$ & $t^{bc}(Z,Y)$.
$(IT3.1)$: $m\_t^{bc}(W,Y)$:- $u(W)$ & $Y > W$ & $v(W,Y)$.
$(IT3.2)$: $m\_t^{bc}(W,Y)$:- $u(W)$ & $Y > W$ & $q(W,Z)$ & $t^{bc}(Z,Y)$.
$(IT4)$: $m\_t^{bc}(Z,Y)$:- $m\_t^{bc}(X,Y)$ & $q(X,Z)$.

However, program IT is not query equivalent to program T with respect to query Q. Given $v = \{34\}$, $q = \{12, 23\}$, and $u = \{1\}$, program T computes $t^{bc} = \{12, 13, 14\}$, for an answer $\{14\}$ to query Q. Program IT computes $t^{bc} = \emptyset$, for an empty answer to query Q.

Grounded rule $IT3.2$ requires that we compute $t^{bc}(Z,Y)$ for a different set of bindings and conditions than computed by any existing rule. These new bindings and conditions are:

$(IT5)$: $m\_t^{bc}(Z,Y)$:- $u(W)$ & $Y > W$ & $q(W,Z)$.

Rule $IT5$ computes $m\_t^{bc} = \{2Y, 3Y\}$. If this set could be grounded, we would go on to compute $m\_t^{bc} = \{14, 24, 34\}$, and $t^{bc} = \{14, 24, 34\}$ to arrive at the correct answer.
□

The problem in Example 21 can be traced to the fact that we used a recursive subgoal as a grounding subgoal. In rule $MT2$, the subgoal $t^{bc}(Z,Y)$ was recursive with the head of the rule. When we used it for grounding, we had to generate new magic values from the magic rule we were trying to ground.

To avoid the problems similar to Example 21, we require that a grounding subgoal in a rule for predicate p^α be non-recursive with p^α. Thus in Example 21, t^{bc} cannot be used as a grounding subgoal in rule $T2$. This means that we cannot use GMST to push the condition on Y.

Section 10 lists some possibilities for extending GMST to allow grounding by recursive subgoals in Datalog programs.

8.1 Groundable sips

In a rule r for a conditioned literal, p^α, some of the subgoals are grounding, and others are nongrounding. GMST requires that the grounding subgoals be non-recursive with p^α, and moves the grounding subgoals out of rule r to ground the non-range-restricted magic rules for m_p^α.

The above operation restricts the flow of information between the grounding and nongrounding subgoals in rule r. We define *groundable sips* as the sips whose information flow obeys the restrictions required to apply GMST.

Definition 11 — Groundable sips: Let r be a rule for head predicate p, and let $\alpha = *c*$ be an adornment string containing at least one condition.

A sips $s = \mathsf{sips}(r, \alpha) = (V, E, B)$ for rule r and head adornment α is said to be *groundable* if the subgoals of rule r can be partitioned into two sets, the grounding set $G(s)$, and the nongrounding set $NG(s)$, satisfying the following properties:

1. p^α is nonrecursive to all subgoals in $G(s)$.
2. The head literal p_h^α does not pass any condition into a subgoal of $NG(s)$.
3. No information is passed from a subgoal in $NG(s)$ to a subgoal in $G(s)$. That is,

$$((T \rightarrow q) \in E \wedge q \in G(s)) \Rightarrow (T \cap NG(s) = \emptyset)$$

4. If either p_h^α or a subgoal of $G(s)$ passes information into a nongrounding subgoal $q \in NG(s)$, then p_h^α and all subgoals of G must pass information into subgoal q. That is,

$$((T \rightarrow q \in E) \wedge (q \in NG(s)) \wedge (T \cap (G(s) \cup p_h^\alpha) \neq \emptyset)) \Rightarrow T \supseteq (G(s) \cup p_h^\alpha)$$

5. $\mathsf{var}(G(s)) = \mathsf{bvar}(G(s))$.

For a groundable sips s, define
$\mathsf{unusable}(s) = \mathsf{cvar}(p_h^\alpha) - \mathsf{bvar}(G(s))$.
□

Condition 1 guarantees that grounding subgoals are not mutually recursive with the head. Condition 2 prevents us from grounding a condition with a nongrounding subgoal. Consequently, head conditions can be used only to restrict computation of grounding subgoals. Condition 3 allows us to evaluate all grounding subgoals before evaluating any nongrounding subgoal. Condition 4 allows us to treat the grounding subgoals and the magic predicate for p^α as one entity, so that they may be replaced by a supplementary predicate, as we did in rules $GP1$ and $GP2$ of Example 20. Condition 5 ensures that if the grounding subgoals include a built-in predicate, the variables used in the built-in predicate are *limited* by some other grounding subgoal.

Any conditioned head variables that are not *limited* by the grounding subgoals are considered unusable. For example, when a condition is grounded by a recursive subgoal, as in rule $T2$ of Example 21, it will be marked unusable. unusable conditions are dropped by GMST in the grounding step.

Definition 12 — Minimal Grounding Set: The minimal grounding set of a sips s is the set $MG(s)$ of subgoals that includes, for every head condition being passed into the body, at least one subgoal into which the condition is passed. Further no subset of $MG(s)$ should be a grounding set. □

Unless we say otherwise, we will assume that $G(s)$ represents a minimal grounding set of a sips s.

A groundable sips is not required to be a full sips, and a rule may not have a full sips that is groundable. However, we can define sips that pass all possible information within the constraints imposed by groundability.

Definition 13 — Full Groundable Sips: A full groundable sips is one that imposes a total order on subgoals, and passes all information with one exception: A head condition that cannot be passed into a subgoal preceding the first recursive subgoal, is not passed into the first recursive subgoal or any subgoal after the first recursive subgoal. □

Lemma 2 *A full groundable sips has a unique minimal grounding set.* □

Proof: The minimal grounding set of a full groundable sips is the set of subgoals till position j such that the sips passes into the j^{th} subgoal a head condition not passed into any previous subgoal, and all head conditions are passed into at least one of the first j subgoals. □

EXAMPLE 22 The sips chosen for rules $P1$ and $P2$ while adorning them in Example 8 are groundable. For rule $P1$, $G = \{flat\}$, $NG = \emptyset$, and $\mathsf{unusable} = \emptyset$. For rule $P2$, $G = \{up\}$, $NG = \{sg, \mathtt{down}\}$, and $\mathsf{unusable} = \emptyset$. □

EXAMPLE 23 The sips $s1$ chosen for rule $T2$ in Example 21, as implicit in the adornments used, is not groundable.

$(T2)$: $t^{bc}(X,Y) :- q(X,Z) \ \& \ t^{bc}(Z,Y)$.

The subgoal t^{bc} is recursive with the head, and therefore must be in the nongrounding set $NG(s1)$ (Condition 1 of Definition 11). The sips $s1$ passes the head condition on Y into the t^{bc} subgoal, violating Condition 2 of Definition 11.

An alternative sips $s2$ that does not pass the head condition into the t^{bc} subgoal, but is otherwise similar to $s1$, is groundable, with $G(s2) = \emptyset$, $NG(s2) = \{q, p^{cf}\}$, and $\mathsf{unusable}(s2) = \{Y\}$. Note that $s2$ is not a full sips, but it is a full groundable sips. The resulting adorned rule is:

$(T2)$: $t^{bc}(X,Y) :- q(X,Z) \ \& \ t^{bf}(Z,Y)$.

□

Lemma 3 *There exists a groundable sips for every rule r and every conditioned adornment α.* □

Proof: Given a rule r for head predicate p, any sips $s = sips(r,\alpha)$ with $G(s) = \emptyset$ and $\mathsf{unusable}(s) = \mathsf{cvar}(p_h^\alpha)$ is trivially groundable. □

However, a sips that declares every head condition $\mathsf{unusable}$ is usually not interesting. We prefer a groundable sips that has (1) few $\mathsf{unusable}$ variables, so that more head conditions are used, and (2) a small grounding set, so that fewer subgoals are copied into each magic rule. Algorithm 3 can be used to find such a desirable grounding sips if (1) we bias the next function towards choosing non-recursive subgoals that ground a head condition, and (2) the set $\mathcal{C}$ of conditioned variables at positions greater than j is determined as:

$$\mathcal{C} = \mathsf{cvar}(\{C, u_1, u_2, \ldots, u_{j-1}\}).$$

Theorem 1 *Consider a bcf adorned program P, a conditioned goal p^α in P, and a rule r for p^α adorned according to a groundable sips s. Treating the conditions as bindings, apply the magic-sets transformation to the program P. Let mr be a resulting (non-range-restricted) rule for m_p^α. The following steps will make rule mr range-restricted:*

1. *If a conditioned argument of the head p^α in rule r is equal to a constant c, add the subgoal $X = c$ to rule mr, where X is the variable in the corresponding argument of m_p^α.*
2. *Drop the arguments of m_p^α that correspond to unusable variables of rule r. Drop any conditions in the rule r over the arguments just dropped.*
3. *Move the grounding subgoals in set $G(s)$ to the body of rule mr, mapping variables so that the variables in $G(s)$ equal to bound and conditioned arguments in the head of rule r are equated to the corresponding variables in rule mr, and all other variables are renamed to unique variables.*

□

Proof: The magic predicate m_p^α has an argument for each conditioned column of p^α. Consider any such argument position k with variable X that is not *limited* in the body of rule mr. Since variable X is not *limited*, only the built-in predicates in the body of mr refer to X.

Let Y be the term in the k^{th} argument of the head literal p^α in rule r. We consider three cases:

1. Y is a constant c. Step 1 of Theorem 1 adds the subgoal $X = c$ to the body of rule mr, thereby limiting variable X.
2. Y is an unusable variable. Step 2 of Theorem 1 drops X entirely from rule mr.
3. If none of the above two cases applies, then Y is a variable and Y must appear in and be *limited* by the subgoals of the grounding set $G(s)$. Step 3 of Theorem 1 moves the grounding subgoals of $G(s)$ into rule mr, mapping variable Y to X, and thereby limiting X.

All variables in rule mr are now *limited*, so that the rule is range-restricted. □

The importance of Theorem 1 is that every adorned rule r for p^α has a unique grounding set $G(r)$, and a unique unusable set $\mathsf{unusable}(r)$, that can be used to ground every magic rule for m_p^α, regardless of the actual conditions in the body of the magic rule.

An adorned program A, adorned according to a groundable sips for every rule in the program, is said to be a *groundable adorned program.*

9 Ground Magic-sets Transformation: The Algorithm

We define the Ground Magic-sets Transformation. We give an outline of the algorithm in Section 9.1 and introduce notation in Section 9.2. The algorithm itself is the subject of Section 9.3. Section 9.4 applies the algorithm to an example.

9.1 Outline

GMST transforms a groundable adorned Datalog program in two logical steps. In the first step GMST treats conditions as if they were bindings and applies the magic-sets transformation. Step 1 produces magic rules that may be non-range-restricted. Step 2 of GMST is the grounding step, where all non-range-restricted rules are grounded to make them range-restricted.

The grounding step works as follows: Let there be two rules, $r1$ and $r2$ for a conditioned predicate p, and two rules, $m1$ and $m2$ for the magic predicate m_p. (We consider the case of two rules and ignore adornments and predicate arguments for simplicity. The GMST Algorithm 5 is defined for any number of rules).

$(r1)$: $p :- m\_p \ \& \ G_1 \ \& \ NG_1.$
$(r2)$: $p :- m\_p \ \& \ G_2 \ \& \ NG_2.$

$(m1)$: $m\_p :- B_1 \ \& \ C_1.$
$(m2)$: $m\_p :- B_2 \ \& \ C_2.$

We have partitioned subgoals of rule $r1$ into the grounding set G_1 that limits the conditioned variables of the head predicate p, and the nongrounding set NG_1 of the remaining subgoals. The subgoals of rule $r2$ are partitioned similarly. The subgoals of magic rule $m1$ have been partitioned into the set C_1 of built-in predicates over the conditioned variables, and the set B_1 of all other subgoals. The subgoals of rule $m2$ are partitioned similarly.

The grounding transformation rewrites the above rules as

$(gr1)$: $p :- s_{r1} \ \& \ NG_1.$
$(gr2)$: $p :- s_{r2} \ \& \ NG_2.$

$(gm1.1)$: $s_{r1} :- B_1 \ \& \ C_1 \ \& \ G_1.$
$(gm2.1)$: $s_{r1} :- B_2 \ \& \ C_2 \ \& \ G_1.$

$(gm1.2)$: $s_{r2} :- B_1 \ \& \ C_1 \ \& \ G_2.$
$(gm2.2)$: $s_{r2} :- B_2 \ \& \ C_2 \ \& \ G_2.$

Rules $m1$ and $m2$ are grounded by moving the grounding subgoals G_1 and G_2 from rules $r1$ and $r2$ into the bodies of $m1$ and $m2$. In the ground rule $gm1.1$, subgoals of G_1 limit the condition variables, thereby making $gm1.1$ range-restricted. The additional variables of G_1 are added to the magic predicate to form a supplementary predicate s_{r1} that replaces $m\_p \ \& \ G_1$ in rule $r1$.

The sips for the newly constructed supplementary rules $gm1.1$, $gm1.2$, $gm2.1$, and $gm2.2$ and for the rules $gr1$ and $gr2$ can be derived from the groundable sips for rules $r1$ and $r2$. Consider a grounding subgoal g in set G_1. If subgoal g got information from the magic literal m_p in rule $r1$, then subgoal g will get the same information from the literals $B_1 \ \& \ C_1$ in the supplementary rule $gm1.1$. If a nongrounding subgoal ng in set NG_1 got information from the magic literal m_p or from a grounding subgoal in rule $r1$, then subgoal ng will get the same information from subgoal s_{r1} in rule $gr1$.

Assuming that the rule $r1$ above is produced by Step 1 of GMST, then Step 1 would also have produced a magic rule from each grounding subgoal g in G_1:

(u): m_g :- m_p & ....

During the grounding phase, we move subgoal g into the rules for m_p. The rules for m_g should therefore be derived from rules $gm1.1$ and $gm2.1$, not from rule $r1$. Thus after grounding, rule u needs to be modified to:

$(u1)$: m_g :- B_1 & C_1 & ....
$(u2)$: m_g :- B_2 & C_2 & ....

Similarly, the magic rules derived from the nongrounding subgoals NG_1 in rule $r1$ during Step 1 also need to be modified after grounding to replace the reference to "m_p & G_1" by a reference to s_{r1}.

9.1.1 Interleaving Steps 1 and 2

Can we avoid modifying magic rules after deriving them? In the GMST algorithm, we interleave Steps 1 and 2 so as to generate the magic rules in a specific order that avoids the need to rewrite them later. We use the following idea:

Subgoals in a rule for predicate p belong to a lower or the same stratum (Section 2.3) as predicate p. As a result, the magic-sets transformation on a rule for predicate p cannot generate magic rules for a predicate of a higher stratum than the stratum of p. (By "magic rule for a predicate q" we mean a rule for the magic predicate m_q.) This suggests a GMST algorithm where we process a program P from the highest stratum (containing the query goal) to the lowest stratum (base predicates). When processing a stratum j, we generate magic rules from nongrounding subgoals, but do not generate magic rules from grounding subgoals. If we do it right, we can make sure that the magic rules from nongrounding subgoals are in their final form. After finishing magic-sets transformation on stratum j, we have generated all the magic rules for predicates of stratum j that we will ever generate, since a later magic-sets transformation on a lower stratum cannot generate magic rules for predicates of stratum j. At this point, we ground all magic rules for predicates of stratum j to derive supplementary rules, and then do magic-sets transformation on the supplementary rules to derive magic rules from the grounding subgoals.

9.2 Notation

Given a program P and a query Q:

(Q): ?- C & $q^\gamma(\bar{t})$.

where C is a set of conditions, we construct a stratification of program P and assign stratum numbers to predicates as discussed in Section 2.3. If k is the stratum number for the query predicate, the query rule Q is placed in stratum $k+1$. $P^1, P^2, \ldots, P^k$ is the set of rules for predicates of stratum $1, 2, \ldots k$, abbreviated as rules for stratum $1, 2, \ldots k$.

We use m_p^α for the magic predicate of p^α, and s_r for the supplementary predicate of rule r at the position after the grounding subgoals of rule r. Thus, s_r is the predicate formed by grounding a magic predicate with grounding subgoals from rule r. By "magic rules for p^α," we will mean the rules for m_p^α, and by "supplementary rules for r," we will mean the rules for s_r. By "magic rules for stratum j," we will mean the magic rules for all predicates in stratum j, and by "supplementary rules for stratum j," we will mean the supplementary rules for all rules in stratum j.

A magic rule for predicate p^α is generated when MST is applied to a rule r that has p^α as a subgoal. The magic rule derived in this manner will be referred to as the "magic rule derived from subgoal p^α in rule r." By "magic rules derived from rule r," we will mean all magic rules derived from subgoals of rule r, and by "magic rules derived from stratum j," we will mean all magic rules derived from rules in stratum j.

If predicate p^α is in stratum j, magic rules for p^α are derived from p^α subgoals in rules for strata higher than or equal to j. We denote the magic rules for p^α derived from rules in strata higher than the stratum of p^α by $M_h(p^\alpha)$. Magic rules for p^α derived from rules of the same stratum as p^α are denoted by $M_=(p^\alpha)$. Then $M(p^\alpha) = M_h(p^\alpha) \cup M_=(p^\alpha)$ represents the full set of magic rules for p^α.

$\mathcal{M}(j) = \bigcup_{p \in \mathsf{preds}(j)} M(p)$ is the set of magic rules for stratum j. $\mathcal{M}_h(j)$, the set of magic rules for stratum j derived from higher strata, and $\mathcal{M}_=(j)$, the set of magic rules for stratum j derived from stratum j, are the unions of $M_h(p)$ and $M_=(p)$ over predicates in stratum j.

9.3 GMST Algorithm

Algorithm 5

Input:

- A range-restricted, groundable adorned program P, and a query Q:

  (Q): ?- C & $q^\gamma(\bar{t})$.

  where C is a set of conditions and $\bar{t}$ is a list of constants and variables.

- For each rule r for predicate p^α in P, a groundable sips $s(r)$ (rule r is adorned, so the sips is a function of the rule alone), and if p^α is conditioned, the grounding set $G(r)$. Subgoals in $G(r)$ must be nonrecursive with p^α. The nongrounding set $NG(r)$, and the unusable set $\mathsf{unusable}(r)$ can be derived from the grounding set $G(r)$.

- A stratification of Program P, with query predicate q in stratum k, and the base predicates in stratum 0.

Output: A range-restricted magic transformed program GP = GMST(P, Q) that is query equivalent to program P with respect to the query Q.

Method:
Create the seed magic rule

(GQ): $m\_q^\gamma(\bar{t}^{bc})$:- C.

where $\bar{t}^{bc}$ denotes the bound and conditioned arguments of $q^\gamma(\bar{t})$ in query Q. Rule GQ belongs to the set $M_h(q)$, and is the only rule in $\mathcal{M}_h(k)$.

for ($j = k$ **to** 1)
  Process stratum j.

Processing a stratum j**:** Perform the following four steps:

(A): Form supplementary predicates. For every rule $r \in P^j$ (P^j is the set of rules for predicates in stratum j), if rule r is for a conditioned predicate p^α, do:

1. Create a supplementary predicate s_r.
2. Determine the arguments, $\bar{Y}$, of s_r.
   $\bar{Y} = (\mathsf{bvar}(p_h^\alpha) \cup \mathsf{var}(G(r))) \bigcap (\mathsf{var}(p^\alpha) \cup \mathsf{var}(NG(r)))$.
3. Replace the grounding subgoals $G(r)$ with the supplementary subgoal $s_r(\bar{Y})$ in the body of rule r.

(B): Magic-Transform P^j**.** Apply magic-sets transformation to every rule r for p^α in P^j, deriving magic rules from the subgoals of r. If p^α is a bound predicate, we apply the regular MST. If p^α is a conditioned predicate, then while applying MST we must (1) treat conditions as if they are bindings, (2) assume that the head bindings/conditions come from the supplementary subgoal s_r, and not from m_p^α, and (3) not derive a magic rule from the s_r subgoal. Since a conditioned rule r is modified in Step A, magic rules are derived only from the nongrounding subgoals, and they all refer to the supplementary predicate s_r.

The magic rules generated in this step include magic rules for predicates in stratum j. These go into $\mathcal{M}_=(j)$, completing construction of the set $\mathcal{M}(j)$. Magic rules for predicates in lower strata i, $i < j$ are also created in this step. These rules go into the set $\mathcal{M}_h(i)$.

(C): Create Supplementary Rules. For each supplementary predicate s_r created in Step A, construct the supplementary rules defining s_r. Let p^α be the head predicate of rule r. Then, one supplementary rule is created from each rule for m_p^α in the set $\mathcal{M}(j)$. Given a magic rule m, the supplementary rule $s_{r.m}$ is derived as follows:

1. Unify the head of rule m with the b and c adorned arguments in the head of rule r. Do the substitution implied by the unification into the body literals of m. Let B be the resulting subgoals in the body of rule m.
2. Create the supplementary rule:

   $(s_{r.m})$: $\quad s_r(\overline{Y})$:- B & $G(r)$.
3. From amongst the B subgoals of the rule created above, remove any conditions over variables in unusable(r).

(D): Magic-Transform Supplementary-Rules. Treating conditions as if they are bindings, apply the magic-sets transformation to each supplementary rule $s_{r.m}$-+ created in Step C to derive magic rules from the grounding subgoals. We must assume that no bindings/conditions come from the head s_r, and we must derive magic rules from the grounding subgoals only. As an example, for the supplementary rule $s_{r.m}$ above, we derive magic rules from subgoals in set $G(r)$, but not from subgoals in set B.

Since the grounding subgoals $G(r)$ are required to be in a lower stratum than p^α, all magic rules generated in this step are for a stratum i lower than j. These rules would go into set $\mathcal{M}_h(i)$.

The supplementary rules, magic rules for bound predicates, and the modified original rules of program P form the output program G. $\diamond$

If a predicate p^α is conditioned (bound), its magic predicate m_p^α is also said to be conditioned (bound).

Lemma 4 *The modified rules for original predicates, the supplementary rules, and the magic rules derived by GMST do not refer to any conditioned magic predicate in their body.* $\Box$

Proof: We prove, by *induction on the strata of the original program*, that for any stratum i, the modified original rules in P^i, the supplementary rules for stratum i, and the magic rules derived from stratum i, all do not refer to any conditioned magic predicate.
Basis $(i = k+1)$: Stratum $k+1$ consists of only the query rule Q. The query rule, and the magic rule MQ derived from it, do not refer to any magic predicate.
Inductive Step: The inductive hypothesis is that the claim is true for all strata greater than j.
Consider the behavior of GMST on stratum j.

- In Step A, the conditioned rules (rules for conditioned predicates) of P^j are modified to refer to supplementary predicates.
- In Step B we derive magic rules from P^j. If a rule is for a conditioned predicate p^α, then Step B ensures that the derived magic rules refer to the supplementary predicate, and not to the conditioned magic predicate m_p^α. If a rule is for a bound predicate, then the rule and derived magic rules refer to a bound magic predicate. If a rule is for a free predicate, then the rule and derived magic rules do not refer to a magic predicate. Thus, none of the magic rules derived from stratum j in Step B refer to a conditioned magic predicate.

- In Step C we derive supplementary rules for stratum j. Each supplementary rule is constructed from the body of a magic rule derived from a stratum higher than j, or derived from stratum j in Step B, and from the grounding set consisting of original predicates. From the inductive hypothesis, and from the result of Step B, the magic rules do not refer to a conditioned magic predicate. Thus, none of the supplementary rules for stratum j refer to a conditioned magic predicate.
- In Step D we derive magic rules from supplementary rules for stratum j. Information is not obtained from the supplementary predicate in the head, and the body doesn't refer to conditioned magic predicates. Thus, no magic rule derived in Step D refers to a conditioned magic predicate.

It follows that the claim is true for stratum j.

□

Theorem 2 *Let $GP = GMST(P, Q)$ be the result of applying Algorithm 5 on a range-restricted groundable adorned Datalog program P for a query Q. The GMST transformation has the following properties.*

1. *Program GP is range-restricted.*
2. *Program GP is query equivalent to program P with respect to the query Q.*

□

Proof:
Claim 1. Program GP is range-restricted: Program GP consists of three types of rules, (1) modified original rules, (2) supplementary rules, and (3) magic rules for bound predicates.

- The original rules are given to be range-restricted. An original rule r is modified by removing grounding subgoals and adding a supplementary predicate. Any variable that appears in the head of r or in a nongrounding subgoal, and is *limited* by a grounding subgoal, will be an argument of the supplementary predicate (Step A.2), and will therefore continue to be *limited* in the modified rule.
- The supplementary rules are obtained by grounding a non-range-restricted magic rule. The three grounding steps listed in Step C of GMST correspond to the three steps in Theorem 1 that make a magic rule range-restricted. Thus, Theorem 1 implies that the supplementary rules are range-restricted.
- The magic rules for bound predicates are derived by MST, without treating conditions as bindings, and are range-restricted.

Claim 2. Program GP is query equivalent to Program P with respect to query Q: Treating conditions as if they are bindings, apply magic-sets transformation to program P for query Q, and let MP be the resulting non-range-restricted program. Assuming we can support non-range-restricted rules, Ramakrishnan's work on magic templates [Ram91] implies that programs P and MP are query equivalent.

Let GP^+ be the program GP extended with the magic rules for conditioned predicates derived by Algorithm 5 during computation of program GP. These magic rules were later discarded as there are no references to the conditioned magic predicates (Lemma 4). Programs GP and GP^+ are query equivalent with respect to query Q.

We prove, by *induction on the strata of the original program P*, that for any stratum i, there exists an equivalence preserving transformation E_i such that programs $E_i(MP)$ and GP^+ are identical in all modified rules of strata i or greater, the supplementary rules for strata i or greater, and the magic rules derived from strata i or greater. Let $F(i)$ represent the above property for stratum i. Then, $F(1) \Rightarrow (\exists E_1)(E_1(MP) \equiv GP^+ \wedge (E_1(MP), Q) \equiv (MP, Q))$, so the query equivalence of GP and P is proven.

Basis ($i = k + 1$): Stratum $k + 1$ consists of only the query rule Q. The query rule and the magic rule MQ derived from it are identical in programs MP and GP^+. Thus E_{k+1} = the null transformation makes $F(k + 1)$ true.

Inductive Step: The inductive hypothesis is that $F(j + 1)$ is true. Let E_{j+1} be the equivalence preserving transformation that makes $F(j + 1)$ true. Hence, $(E_{j+1}(MP), Q) \equiv (MP, Q)$.

We will now apply two equivalence preserving transformations $T1$ and $T2$ on program $E_{j+1}(MP)$ to get a program S_2 that is identical to GP^+ in all modified rules of strata j or greater, the supplementary rules for strata j or greater, and the magic rules derived from strata j or greater.

Let p^α be a conditioned predicate in stratum j. The rule r for p^α in program $E_{j+1}(MP)$ looks like:

(r): p^α :- m_p^α & G & NG.

G is the grounding set that must be nonrecursive with p^α. For every conditioned rule r in stratum j, define the common subexpression:

$(s.r)$: s_r :- m_p & G.

T1: $E_{j+1}(MP) \rightarrow S_1$ We first eliminate all references to magic predicates of stratum j in the rules of stratum j, and in the magic rules derived from nongrounding subgoals of the rules of stratum j, in a manner that make these rules identical to those in GP^+.

- **Rules of stratum** j**:** Each rule of the form r is replaced by:

  (gr): p^α :- s_r & NG.

- **Magic rules derived from nongrounding subgoals of stratum** j**:**

  We first note that Condition 4 of a groundable sips implies that each magic rule derived from a nongrounding subgoal in rule r either refers to m_p and all grounding subgoals, or does not refer to any of these. In the former case, we substitute s_r for the reference to "m_p & G." In the later case, we do nothing.

Let S_1 be the program after the above substitutions. Program S_1 is equivalent to $E_{j+1}(MP)$ since each substitution is an application of the folding transformation [TS84]. Also, each rule obtained after folding is identical to the corresponding rules derived in program GP^+ at Steps A and B of GMST during processing of stratum j.

The program S_1 has the property that no magic rule for stratum j refers to a conditioned magic predicate. To see this, consider how the magic rule m was derived. If rule m was derived from a higher stratum, then the inductive hypothesis states that rule m must be identical to the corresponding magic rule gm in GP^+. By Lemma 4, rule gm doesn't refer to a conditioned magic predicate. If rule m was derived from stratum j, let it be derived from a subgoal t^θ in the rule r for p^α. When created, rule m may have a reference to the conditioned magic predicate m_p^α. However, predicate t^θ is recursive with p^α (both are in stratum j), so the subgoal t^θ must be in the nongrounding set NG, and we would have eliminated the reference to m_p^α in rule m when processing magic rules derived from nongrounding subgoals. As is evident, this property depends on the grounding set of a conditioned rule being nonrecursive with the head.

T2: $S_1 \rightarrow S_2$ In the second transformation, we eliminate all references to magic predicates of stratum j in the supplementary rules for stratum j, and in the magic rules derived from grounding subgoals in the rules of stratum j. The references can be eliminated by an unfolding step [TS84] that replaces an m_p^α subgoal by all the definitions of the magic predicate m_p^α (that is, by the bodies of the rules for m_p^α).

After copying the rule bodies of m_p^α into the rules for s_r and the magic rules derived from the grounding subgoals of rule r, we drop all the copied conditions that are unusable in rule r. We also drop the arguments of s_r corresponding to the unusable variables. We note that unusable conditions are not passed into the grounding or nongrounding subgoals, so their magic rules do not refer to the unusable variables in rule r. Consequently, dropping the unusable variables in the unfolded magic rules from grounding subgoals preserves equivalence.

Dropping an unusable condition c from the unfolded rules for s_r means that rule r will be evaluated without the condition c. Consequently, the extension of p^α in the transformed program may be a superset of extension of p^α in program S_1. However, the new program is still query equivalent to S_1, since the dropped condition c would be applied in the rules where p^α appears as a subgoal.

Let S_2 be the program after the above transformations. Program S_2 is equivalent to program S_1 since each unfolding step preserves equivalence, and as discussed, dropping unusable conditions also preserves query equivalence. Further, the supplementary rules for stratum j, and the magic rules derived from the grounding subgoals of stratum j are identical to the corresponding rules derived in program GP^+ at Steps C and D of GMST during processing of stratum j.

Let $E_j = T2(T1(E_{j+1}))$ be the concatenation of transformations $T2$, $T1$, and E_{j+1}. The transformation E_j makes property $F(j)$ true. □

9.4 An Example

EXAMPLE 24 Consider the query Q on adorned program P:

(Q): ?- $X > 10$ & $p^{cf}(X,Y)$.
$(P1)$: $p^{cf}(X,Y)$:- $u(X,Y)$.
$(P2)$: $p^{cf}(X,Y)$:- $U > 10$ & $q^{ccf}(X,U,V)$ & $W > V$ & $p^{cf}(W,Y)$.
$(P3)$: $q^{ccf}(X,Y,Z)$:- $q1(X,U)$ & $q2(W,Y)$ & $q3(U,W,Z)$.

The query predicate p^{cf} is in stratum $k = 2$, predicate q^{ccf} is in stratum 1, and the remaining predicates are base predicates in stratum 0.

Let the sips for the three rules be the full grounding sips corresponding to the subgoal order shown, with minimal grounding sets $G(P1) = \{u(X,Y)\}$, $G(P2) = \{U > 10, q^{ccf}(X,U,V)\}$, and $G(P3) = \{q1(X,U), q2(W,Y)\}$. The `unusable` sets are empty.

The initialization step of GMST derives the seed magic rule:

$(MP4)$: $m\_p^{cf}(X)$:- $X > 10$.

GMST now performs two iterations, first processing stratum 2, and then processing stratum 1. In the following discussion, a paragraph labeled **(jL)** describes the effect of Step **L** of GMST on stratum **j**. The paragraphs are listed in the order in which processing occurs.

(2A): $s_{P1}(X,Y)$ and $s_{P2}(X,V)$ are the supplementary predicates for rules $P1$ and $P2$. Rules $P1$ and $P2$ are modified to:

$(GP1)$: $p^{cf}(X,Y)$:- $s_{P1}(X,Y)$.
$(GP2)$: $p^{cf}(X,Y)$:- $s_{P2}(X,V)$ & $W > V$ & $p^{cf}(W,Y)$.

(2B): Magic transformation on rules $GP1$ and $GP2$ derives the second magic rule for m_p^{cf}.

$(MP5)$: $m\_p^{cf}(W)$:- $s_{P2}(X,V)$ & $W > V$.

(2C): We create supplementary rules for the predicates s_{P1} and s_{P2}. Since m_p^{cf} has two magic rules, each supplementary predicate must have two rules, one from each of the magic rules $MP4$ and $MP5$.

$(GP4.1)$: $s_{P1}(X,Y)$:- $X > 10$ & $u(X,Y)$.
$(GP5.1)$: $s_{P1}(X,Y)$:- $s_{P2}(X_1,V_1)$ & $X > V_1$ & $u(X,Y)$.
$(GP4.2)$: $s_{P2}(X,V)$:- $X > 10$ & $U > 10$ & $q^{ccf}(X,U,V)$.
$(GP5.2)$: $s_{P2}(X,V)$:- $s_{P2}(X_1,V_1)$ & $X > V_1$ & $U > 10$ & $q^{ccf}(X,U,V)$.

(2D): Magic transformation on the supplementary rules $GP4.1$, $GP5.1$, $GP4.2$, and $GP5.2$, generates two magic rules for m_q^{ccf}.

$(MP6)$: $m\_q^{ccf}(X,U)$:- $X > 10$ & $U > 10$.
$(MP7)$: $m\_q^{ccf}(X,U)$:- $s_{P2}(X_1,V_1)$ & $X > V_1$ & $U > 10$.

(1A): $s_{P3}(X,U,W,Y)$ is the supplementary predicate for rule $P3$. Rule $P3$ is modified to:

$(GP3)$: $q^{ccf}(X,Y,Z)$:- $s_{P3}(X,U,W,Y)$ & $q3(U,W,Z)$.

(1B): Magic transformation on rule $GP3$ does not derive any magic rule.

(1C): We create supplementary rules for the predicate s_{P3}. Each of the two magic rules $MP6$ and $MP7$ for m_q^{ccf} is grounded by the grounding set of rule $P3$ to generate two supplementary rules.

$(GP6.3)$: $s_{P3}(X, U, V, W)$:- $X > 10$ & $Y > 10$ & $q1(X, U)$ & $q2(W, Y)$.
$(GP7.3)$: $s_{P3}(X, U, V, W)$:- $s_{P2}(X_1, V_1)$ & $X > V_1$ & $Y > 10$ & $q1(X, U)$ & $q2(W, Y)$.

(1D): Magic transformation on the supplementary rules $GP6.3$ and $GP7.3$ does not create any magic rule.

GMST Terminates: The transformed program GP is given by the rules:

$(GP1)$: $p^{cf}(X, Y)$:- $s_{P1}(X, Y)$.
$(GP2)$: $p^{cf}(X, Y)$:- $s_{P2}(X, V)$ & $W > V$ & $p^{cf}(W, Y)$.
$(GP3)$: $q^{ccf}(X, Y, Z)$:- $s_{P3}(X, U, W, Y)$ & $q3(U, W, Z)$.
$(GP4.1)$: $s_{P1}(X, Y)$:- $X > 10$ & $u(X, Y)$.
$(GP5.1)$: $s_{P1}(X, Y)$:- $s_{P2}(X_1, V_1)$ & $X > V_1$ & $u(X, Y)$.
$(GP4.2)$: $s_{P2}(X, V)$:- $X > 10$ & $U > 10$ & $q^{ccf}(X, U, V)$.
$(GP5.2)$: $s_{P2}(X, V)$:- $s_{P2}(X_1, V_1)$ & $X > V_1$ & $U > 10$ & $q^{ccf}(X, U, V)$.
$(GP6.3)$: $s_{P3}(X, U, V, W)$:- $X > 10$ & $Y > 10$ & $q1(X, U)$ & $q2(W, Y)$.
$(GP7.3)$: $s_{P3}(X, U, V, W)$:- $s_{P2}(X_1, V_1)$ & $X > V_1$ & $Y > 10$ & $q1(X, U)$ & $q2(W, Y)$.
$\square$

10 Extensions

We have used the *condition* adornment to represent an independent condition due to a built-in Datalog predicate. What about dependent conditions? What about restrictions due to function symbols ($X = f(Y)$) or evaluable arithmetic operations ($X = Y + 5$) in an extension to Datalog? We would like to express a strategy to pass these restrictions in a rule, and then we would like to rewrite the program so that the restrictions can be evaluated early during query evaluation. To do this using our approach, we only need to extend the adornment class to represent the *more general restrictions*. [Mum91, MPR94] give examples of extensions to the *bcf* adornment class. Once we have an adorned program correctly showing how the restrictions are to be passed, GMST can be used to pass the restrictions. The grounding phase of GMST for general restrictions is the same as for independent conditions.

Top-down evaluation methods use bindings to limit evaluation of subgoals, but cannot use conditions. Is it possible to extend the top-down methods to be as efficient as a bottom-up evaluation after the GMST transformation?

A challenging problem is to extend GMST to the case where a head condition is passed into a recursive subgoal. We believe that combining GMST with techniques to commute selections and recursion will lead to a general technique to pass conditions. As an example, consider program T of Example 21. The second column of $t^{bc}(X, Y)$ is invariant through recursion. So the condition on Y can be pushed into the nonrecursive rule, generating the program:

(Q): ?- $u(W)$ & $Y > W$ & $t^{bc}(W, Y)$.
$(T1)$: $t^{bc}(X, Y)$:- $u(W)$ & $Y > W$ & $v(X, Y)$.
$(T2)$: $t^{bc}(X, Y)$:- $q(X, Z)$ & $t^{bc}(Z, Y)$.

We can now ignore the condition on the second column, and optimize program T so as to only pass the binding on the first column.

The ground magic-sets transformation has been implemented in the Starburst extensible relational database system [HCL+90], which is partially incorporated in IBM DB2 V2. [MP94] describes the Starburst implementation as a part of a larger *extended magic-sets transformation* that includes treatment of

SQL with duplicates, aggregation, null values, and subqueries, a cost-based heuristic for determining join orders (sips) before applying magic, and integration with other relational optimization techniques. The implementation demonstrates the feasibility of the magic-sets transformation for commercial relational systems, and provides a mechanism to implement magic as an integral part of a new database system, or as an add-on to an existing database system.

11 Related Work

This paper is an expanded version of [MFPR90a]. An extension of magic-sets to duplicate and aggregate semantics of SQL is discussed in [MPR90], a performance study showing the benefits of magic-sets on SQL queries appears in [MFPR90b], and an implementation of these techniques in Starburst is reported in [MP94]. [MPR94] formalizes adornment classes as an algebraic entity, and discusses the desirable properties of any adornment scheme. [Mum91] is a Ph.D. dissertation based on the work done on magic-sets in context of Starburst [HCL+90].

The magic template transformation [Ram91] is a generalization of MST to pass restrictions due to function symbols, or due to equality predicates between free arguments. The templates approach treats any of these restrictions as if it is a binding, generating a b adornment, and applies the magic-sets transformation producing non range-restricted rules whenever a binding is actually not present. Step 1 of GMST is a straightforward extension of templates to conditions.

Other efforts to optimize queries through early evaluation of constraints include predicate pushdown, predicate movearound [LMS94], the folding-unfolding approach of [BKMR89], the constraint propagation of [SR93], compiling query constraints of [SS94], and theta semi-joins of [SSS95].

Predicate pushdown [Ull89b] deals with local predicates in queries without subexpressions. Extensions of pushdown for recursive queries have also been proposed [AU79, KL86]. These techniques work by realising that an argument is invariant through recursion, and hence a predicate could be applied on it before the recursion starts. Predicate movearound is a generalization of pushdown that also works on local predicates, and is based upon a pullup phase followed by a pushdown phase in a query tree [LS92, LMSS93]. The folding-unfolding approach of [BKMR89] is discussed in detail later. Constraint propagation of [SR93] treats conditions semantically rather than syntactically, and further generalizes pushdown to propagate local predicates through arguments semantically invariant through recursion. This is then followed by a fold/unfold step similar to [BKMR89]. One of the contributions of [SR93] is to show that the grounding step of GMST can be understood as a sequence of folds and unfold transformations. [SS94] also treats constraints semantically, and introduces parametrized constraints that are expressed in terms of parameters to be specified at runtime. A magic-sets transformation using parametrized constraints is also described. [SSS95] introduces a theta semi-join operator to do a semi-join on a condition predicate rather than on an equality predicate, and shows that the effect of magic and GMST on nonrecursive SQL queries can be achieved using a sequence of semi-join transformations.

Recently, there has been much interest in *constraint logic programs*, that is, logic programs that allow arithmetic constraints in rules. Our work can be seen as an efficient evaluation approach for a class of constraint logic programs. Specifically, we consider programs that can be evaluated without storing constraints and variables explicitly with tuples, and address the database situation where there is a large number of facts.

Folding/Unfolding [BKMR89]

We illustrate the approach of Balbin, Kemp, Meenakshi, and Ramamohanarao [BKMR89] on the same generation program. The reader is referred to Tamaki and Sato [TS84] for a discussion of folding and unfolding transformations.

EXAMPLE 25 Consider query Q on program P:

(Q): ?- $X > 5$ & $sg(X,Y)$.
$(P1)$: $sg(X,Y)$:- $\texttt{flat}(X,Y)$.
$(P2)$: $sg(X,Y)$:- $\texttt{up}(X,U)$ & $sg(U,V)$ & $\texttt{down}(V,Y)$.

Balbin et. al. use the *bf* adornment class to generate the following adorned program:

(AQ): ?- $X > 5$ & $sg^{ff}(X,Y)$.
$(AP1)$: $sg^{ff}(X,Y)$:- $\texttt{flat}(X,Y)$.
$(AP2)$: $sg^{ff}(X,Y)$:- $\texttt{up}(X,U)$ & $sg^{bf}(U,V)$ & $\texttt{down}(V,Y)$.
$(AP3)$: $sg^{bf}(X,Y)$:- $\texttt{flat}(X,Y)$.
$(AP4)$: $sg^{bf}(X,Y)$:- $\texttt{up}(X,U)$ & $sg^{bf}(U,V)$ & $\texttt{down}(V,Y)$.

Two steps are needed after the adornment phase. First, Balbin et. al. fold the selection $X > 5$ & $sg^{ff}(X,Y)$ into a predicate $t^{ff}(X,Y)$, and unfold the defining rules for $t^{ff}(X,Y)$ to get the program FP:

(FQ): ?- $t^{ff}(X,Y)$.
$(FP1)$: $t^{ff}(X,Y)$:- $\texttt{flat}(X,Y)$ & $X > 5$.
$(FP2)$: $t^{ff}(X,Y)$:- $\texttt{up}(X,U)$ & $X > 5$ & $sg^{bf}(U,V)$ & $\texttt{down}(V,Y)$.
$(FP3)$: $sg^{bf}(X,Y)$:- $\texttt{flat}(X,Y)$.
$(FP4)$: $sg^{bf}(X,Y)$:- $\texttt{up}(X,U)$ & $sg^{bf}(U,V)$ & $\texttt{down}(V,Y)$.

Second, Balbin et. al. apply MST on program FP to get:

(CQ): ?- $t^{ff}(X,Y)$.
$(CP1)$: $t^{ff}(X,Y)$:- $\texttt{flat}(X,Y)$ & $X > 5$.
$(CP2)$: $t^{ff}(X,Y)$:- $\texttt{up}(X,U)$ & $X > 5$ & $sg^{bf}(U,V)$ & $\texttt{down}(V,Y)$.
$(CP3)$: $sg^{bf}(X,Y)$:- $m\_sg^{bf}(X)$ & $\texttt{flat}(X,Y)$.
$(CP4)$: $sg^{bf}(X,Y)$:- $m\_sg^{bf}(X)$ & $\texttt{up}(X,U)$ & $sg^{bf}(U,V)$ & $\texttt{down}(V,Y)$.
$(CP5)$: $m\_sg^{bf}(U)$:- $\texttt{up}(X,U)$ & $X > 5$.
$(CP6)$: $m\_sg^{bf}(U)$:- $m\_sg^{bf}(X)$ & $\texttt{up}(X,U)$.

Program CP is equivalent to program GP obtained in Example 20 by applying GMST on program P. □

As is evident from Example 25, the folding/unfolding algorithm and GMST achieve identical results on some inputs. However, there exist several differences between the two approaches.

- Balbin et. al. assume that the program is adorned using the *bf* adornment class, with the *f* adornment used for conditioned variables. The success of the folding/unfolding transformation can sometimes depend on the *bf* adornment algorithm being able to split a recursive predicate into two adorned versions. To illustrate, consider how program P in Example 25 is adorned. When adorning rule $P2$ for sg^{ff}, the adornment algorithm has two choices, rule $AP2$ in Example 25, and the following rule $A'P2$:

  $(A'P2)$: $sg^{ff}(X,Y)$:- $sg^{ff}(U,V)$ & $\texttt{up}(X,U)$ & $\texttt{down}(V,Y)$.

  Rule $A'P2$ is attractive since the adornment algorithm thinks we are evaluating sg^{ff} anyway. The algorithm doesn't know that we actually have a condition on the sg^{ff} goal. The folding/unfolding algorithm will fail if rule $A'P2$ is generated.

  We can use the *bcf* adornments to avoid such mistakes in the adornment phase. We believe that the performance of folding/unfolding transformation can be improved by using the *bcf* adornment class.

- Common subexpressions are handled differently by the two techniques. When a conditioned predicate p is used in two or more rules, folding/unfolding will replicate all the rules for p, while GMST will only replicate the grounding subgoals from the rules for p. GMST is therefore preferable.

- The folding/unfolding algorithm cannot push a condition due to a built-in predicate in the rule body into a recursive subgoal. To illustrate, the condition $W > V$ in Example 24 cannot be pushed into the $p(W, Y)$ subgoal.
- Folding/unfolding can push down dependent conditions. Due to the interpretation of a c adornment as an independent condition, GMST, as presented in this section, cannot push down dependent conditions. However, if the adornment system is appropriately extended, GMST will be able to push down dependent conditions.
- Folding/unfolding can correctly push conditions that would be rejected as unusable by GMST in some examples.

Thus, there are programs where GMST is applicable while folding/unfolding is not, other programs where folding/unfolding is applicable while GMST is not. When both techniques are applicable, we expect GMST to perform better if the program has common subexpressions. Without common subexpressions, and with the same *bcf* adorned programs, the two transformations derive similar programs.

12 Conclusions

In this paper we have shown that the magic-sets techniques can be extended to deal with conditions. We believe that the work described in this paper is important not only because it extends the magic-sets transformation but also because it leads to a unified framework to push down all types of predicates in a database query. Consequently the ground magic-sets transformation is useful in practical relational database systems, particularly those with the power of SQL [ISO93].

We have extended the idea of *bf* adornments and shown that other adornment classes can be defined, opening up new ways to describe passing of information from larger classes of restrictions.

13 Acknowledgements

We thank Katherine Morris and Jeffrey D. Ullman for discussions on *bcf* adornments. The Starburst project at IBM Almaden Research Center and the *NAIL!* project at Stanford University provided a stimulating environment for this work. Ashish Gupta, Yatin Saraiya, and Gio Wiederhold provided helpful comments on drafts of this paper. We also thank the referees for their constructive suggestions.

References

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[AU79] Alfred Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Proceedings of the Sixth Symposium on Principles of Programming Languages (POPL)*, pages 110–120, January 1979.

[BKMR89] Isaac Balbin, David B. Kemp, Krishnamurthy Meenakshi, and Kotagiri Ramamohanarao. Propagating constraints in recursive deductive databases. In *North American Conference on Logic Programming (NA-CLP)*, October 16-20 1989.

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth Symposium on Principles of Database Systems (PODS)*, pages 1–15, Cambridge, MA, March 1986.

[BR87] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In *Proceedings of the Sixth Symposium on Principles of Database Systems (PODS)*, pages 269–283, San Diego, CA, March 1987.

[BR88] Francois Bancilhon and Raghu Ramakrishnan. Performance evaluation of data intensive logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 439–518, Washington D.C., 1988. Morgan Kaufmann.

[CGT90] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Springer-Verlag, Berlin Heidelberg, 1990.

[Day87] Umeshwar Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of the Thirteenth International Conference on Very Large Databases (VLDB)*, pages 197–208, Brighton, England, September 1-4 1987.

[GW87] Richard A. Ganski and Harry K. T. Wong. Optimization of nested SQL queries revisited. In Umeshwar Dayal and Irv Traiger, editors, *Proceedings of ACM SIGMOD 1987 International Conference on Management of Data*, pages 23–33, San Francisco, CA, May 27-29 1987.

[HCL+90] L.M. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, pages 143–160, March 1990. Also available as IBM Research Report RJ7278, San Jose, CA, Jan. 1990.

[ISO93] ISO_ANSI. ISO-ANSI working draft: Database language SQL3, 1993.

[Kim82] Won Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, 7(3), September 1982.

[KL86] Michael Kifer and E. Lozinskii. Filtering data flow in deductive databases. In *Proceedings of the International Conference on Database Theory (ICDT '86), LNCS 243*, pages 186–202, 1986.

[KRVV93] Paris C. Kanellakis, Sridhar Ramaswamy, Darren E. Vengroff, and Jeffrey S. Vitter. Indexing for data models with constraints and classes. In *Proceedings of the Twelfth Symposium on Principles of Database Systems (PODS)* [POD93], pages 233–243.

[LMS94] Alon Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. Query optimization by predicate movearound. In Jorge Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proceedings of the 20^{th} International Conference on Very Large Databases*, pages 96–107, Santiago, Chile, September 12-15 1994.

[LMSS93] Alon Levy, Inderpal Singh Mumick, Yehoshua Sagiv, and Oded Shmueli. Equivalence, query-reachability, and satisfiability in Datalog. In *Proceedings of the Twelfth Symposium on Principles of Database Systems (PODS)* [POD93], pages 109–122.

[LS92] Alon Levy and Yehoshua Sagiv. Constraints and redundancy in datalog. In *Proceedings of the Eleventh Symposium on Principles of Database Systems (PODS)*, pages 67–80, San Diego, CA, June 2-4 1992.

[MFPR90a] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic conditions. In *Proceedings of the Ninth Symposium on Principles of Database Systems (PODS)*, pages 314–330, Nashville, TN, April 2-4 1990.

[MFPR90b] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is relevant. In *Proceedings of ACM SIGMOD 1990 International Conference on Management of Data*, pages 247–258, Atlantic City, NJ, May 23-25 1990.

[Mor88] Katherine A. Morris. An algorithm for ordering subgoals in NAIL!. In *Proceedings of the Seventh Symposium on Principles of Database Systems (PODS)*, pages 82–88, Austin, TX, March 1988.

[MP94] Inderpal Singh Mumick and Hamid Pirahesh. Implementation of magic in starburst. In Richard Snodgrass and Marianne Winslett, editors, *Proceedings of ACM SIGMOD 1994 International Conference on Management of Data*, pages 103–114, Minneapolis, MN, May 25-27 1994.

[MPR90] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *Proceedings of the Sixteenth International Conference on Very Large Databases (VLDB)*, pages 264–277, Brisbane, Australia, August 13-16 1990.

[MPR94] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. Adornments in database programs. *Computers and Artificial Intelligence*, 13(2-3):201–230, 1994. Preliminary version appeared in Proceedings of the International Workshop on Deductive Databases, International Conference on Logic Programming, June 1993.

[Mum91] Inderpal Singh Mumick. *Query Optimization in Deductive and Relational Databases*. PhD thesis, Stanford University, Stanford, CA 94305, USA, December 1991. Technical Report No. STAN-CS-91-1400. Also available from University Microfilms International, 300 N. Zeeb Road, Ann Arbor, MI 48106. (313)761-4700.

[POD93] *Proceedings of the Twelfth Symposium on Principles of Database Systems (PODS)*, Washington, DC, May 25-27 1993.

[Ram91] Raghu Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *Journal of Logic Programming*, 11(3):189–216, 1991. Earlier version appeared in International Conference and Symposium on Logic Programming 1988.

[SR93] Divesh Srivastava and Raghu Ramakrishnan. Pushing constraint selections. *Journal of Logic Programming*, 16(3–4):361–414, 1993.

[SS94] Peter Stuckey and S. Sudarshan. Compiling query constraints. In *Proceedings of the Thirteenth Symposium on Principles of Database Systems (PODS)*, pages 56–67, Minneapolis, MN, May 24-26 1994.

[SSS95] Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. The magic of theta-semijoins. Technical Report 311308-950302-02TM, AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[TS84] H. Tamaki and T. Sato. Unfold/fold transformations in logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming*, pages 127–138, Uppsala, Sweden, 1984.

[Ull85] Jeffrey D. Ullman. Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, 10(3):289–321, September 1985.

[Ull89a] Jeffrey D. Ullman. Bottom-up beats top-down for datalog. In *Proceedings of the Eighth Symposium on Principles of Database Systems (PODS)*, pages 140–149, Philadelphia, PA, March 1989.

[Ull89b] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes 1 and 2*. Computer Science Press, 1989.