Automatic reordering for dataflow safety of Datalog

Mistral Contrastin
University of Cambridge
Mistral.Contrastin@cl.cam.ac.uk

Dominic Orchard
University of Kent
d.a.orchard@kent.ac.uk

Andrew Rice
University of Cambridge
Andrew.Rice@cl.cam.ac.uk

ABSTRACT

Clauses and subgoals in a Datalog program can be given in any order without affecting program meaning. However, practical applications of the language require the use of built-in or external predicates with particular dataflow requirements. These can be expressed as input or output modes on arguments. We describe a static analysis of moding for Datalog which can transform an ill-moded program into a well-moded program by reordering clause subgoals, satisfying any dataflow requirements. We describe an incremental algorithm which efficiently finds a reordering if it exists. This frees the programmer to focus on the declarative specification of their program rather than on the implementation details of external predicates. We prove that our computed reorderings yield well-moded programs (soundness) and that if a program can be made well-moded, we compute a reordering to do so (completeness).

ACM Reference Format:
Mistral Contrastin, Dominic Orchard, and Andrew Rice. 2018. Automatic reordering for dataflow safety of Datalog. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Declarative languages aim to free the programmer from implementation details, allowing them to focus on the essence of a problem. However, in practice, implementation details often creep back in. In logic programming, one such implementation detail is *subgoal ordering* which rears its head once we start introducing external functions into pure logic programs or when performance becomes a concern. The aim of this paper is to push the implementation concern of subgoal ordering from the programmer back to the language. We consider Datalog, a syntactic subset of Prolog which has recently been (re)growing in popularity. Among other things, it is used for managing enterprise data [1] and as a language for concisely and efficiently expressing program analyses [10, 15].

Sentences in pure Datalog are Horn clauses, hence subgoals can be specified in *any order* where any ordering is trivially safe to invoke. However, this is not true in practice. Many systems allow useful external functionality such as arithmetic, comparison, and input/output functions. For example, a function for printing to the console might choose not to print unbound values or an external hashing function may require its first parameter to be strictly an input and its second to be an output. In large systems, ordering can also have performance implications, *e.g.*, a database query might make more efficient use of an index if an argument is ground.

The concept of *moding* [13] allows us to specify the dataflow requirements of predicates. For example, a programmer can specify via moding that a particular argument must be bound before the clause is executed as supported in Mercury [17]. Well-moded programs do not give invocation errors in the same way that well-typed programs do not go wrong.

We now describe a few examples of ill-moded programs which can be fixed using the information from our analysis. Consider the following Datalog clause with moding annotations as superscripts:

```
1 auth(U) :- hash+?(P,H), password(U,P), valid(U,H).
```

The superscript +? specifies that the `hash` predicate is safe only when the first argument is bound in an invocation and when the second is free or bound. This clause is not well-moded since `P` is not bound in the context of the first subgoal (it would need to be bound by the clause head), therefore `hash` is invoked with a free variable. However, in the absence of side-effects (discussed later), it is sound to reorder the subgoals to meet the moding constraints by swapping the first and second subgoals so that `password` is invoked first, binding `P` under the usual left-to-right semantics.

As an alternative, one might write this example as two clauses:

```
1 auth(U) :- check(U,P), password(U,P).
2 check(U,P) :- hash+?(P,H), valid(U,H).
```

The `hash` subgoal in `check` has a mode error again because `P` is unbound in the invocation of `check` in `auth` on line 1. However, the reordering to fix this is non-local: one must reorder the body of `auth` to make the invocation of `hash` safe. Searching all permutations of subgoals in all clauses in a program for valid orderings is infeasible. We instead propagate constraints to the callers.

We also provide information that can be used in conjunction with clause cloning. Imagine an interactive system with client- and server-side facilities for checking password strength:

```
1 client_check(P) :- weak(P,H).
2 server_check(H) :- weak(P,H).
3 weak(P,H) :- hash+?(P,H), rainbow+?(H,P).
```

The client side does not have access to the hash and the server side does not have access to the plaintext password. However both parties want to check if the password is compromised by looking up the hash in a rainbow table[1] and confirming that is indeed the password corresponding to the hash. There is no fixed-order of `weak`'s subgoals that satisfy moding requirements of both `hash` and `rainbow`. If we generate two versions of `weak` with different subgoal orderings and use the appropriate one according to the binding pattern at the call site, then queries involving both client- and server-side checks can be well-moded:

[1] A pre-computed reverse lookup table from hashes to plaintext.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
client_check(P) :- weak1(P,H).
server_check(H) :- weak2(P,H).
weak1(P,H) :- hash+?(P,H), rainbow+?(H,P).
weak2(P,H) :- rainbow+?(H,P), hash+?(P,H).
```

Cloning is not a feature of our moding system, but a by-product of an *adornment program transformation* (Section 3.2) which generate versions of each clause annotated with a variable binding pattern. The ordering information from our analysis guides the adornment procedure and ensures the subgoals used in the generated clauses are ordered such that they are safe to invoke.

We show how to statically check that Datalog programs are well-moded. Furthermore, we give an algorithm that computes orderings of subgoals that satisfy the moding constraints of programs, if such an ordering exists. This is valuable because it frees the programmer to focus on the specification of the higher-level goals rather than their syntactic order. Although orderings are computed statically, we do not prescribe a time of use. They can be used to transform the Datalog program statically (suitable for bottom-up or top-down evaluation) or reorder subgoals dynamically as they are evaluated (suitable for top-down evaluation).

The contributions of this paper are:

- a natural formalisation of well-modedness for Datalog in terms of existing adornment program transformation,
- a sound and complete incremental analysis algorithm for finding suitable subgoal orders of program clauses and checking well-modedness of programs.

The rest of the paper is structured as follows. We first state our assumptions and notation (Section 2) and give a formalisation of well-modedness based on the adornment program transformation for Datalog (Section 3). This leads us to our mode analysis algorithm establishing well-modedness and producing subgoal orderings for binding patterns. This is presented in intra-clausal (Section 4) and inter-clausal (Section 5) stages along with properties of the algorithm. We modify the intra-clausal analysis to accommodate some Datalog extensions (Section 6). Finally, we discuss previous approaches to mode analysis (Section 7) and conclude (Section 8). Proofs omitted from the main text are in Appendix A.

2 NOTATION AND ASSUMPTIONS

Definition 2.1 (Datalog program structure). A Datalog program is a set of *clauses* of the form p `:-` $s_1, \ldots, s_n$`.`, where:

- p is the clause *head*;
- s_i collectively forms the *body* of the clause, where each s_i is individually called a *subgoal*;
- A clause head and subgoals are *atomic formulae*;
- An *atomic formula* is a predicate symbol applied to a tuple of terms, *e.g.*, `p(X,Y)`.

We assume subgoals are executed left-to-right and variables are bound to their values whenever possible. Since there are no function symbols in Datalog, binding the value of a variable is same as grounding it.

We consider programs that come with a *query*. A query is of the form `?-` $s_1, \ldots, s_n$. This gets expanded into a normal clause with a fresh head predicate. Moreover, the head of this clause has all variables in the body of the query as its formal parameters.

We use a function *vars* to map the syntax of a logical formula or clause to a set of its variables. Functions *head* and *body* map a clause to its head and the set of its subgoals respectively.

A predicate symbol and its arity uniquely identifies a predicate. For presentation purposes, we assume the function symbol alone uniquely identifies the predicate. A predicate may have multiple occurrences in the body of a clause, *e.g.*, `p(X) :- q(X), q(X).`, has two distinct subgoals using the same predicate. The function *pred* maps a subgoal to its predicate and *arity* maps predicates to their arity. If p is a predicate and Pr is a program, then Pr_p is the set of clauses with p in their head.

Throughout, the "min" operator refers to the minimal elements of a set of subsets under the partial order defined by subset relation. For example, $\min\{\{1,2\},\{2,3\},\{1\},\{1,2,3\}\}$ is $\{\{1\},\{2,3\}\}$.

We assume mode requirements of predicates are available and do not give a syntax for their declarations. This information may be hard-coded in the case of built-in predicates or supplied through mode declarations akin to type declarations.

3 ADORNMENT AND WELL-MODEDNESS

Informally, a well-moded program does not produce runtime errors arising from insufficient variable binding. In this section, we introduce the definitions needed to formalise this notion for Datalog.

3.1 Mode annotations and constraints

Each variable only ever needs one of two modes in Datalog. The mode + indicates that the argument should be bound at the time of invocation and ? indicates it can either be bound or free.

As introduced above, we use *mode vectors* as superscripts to indicate the mode requirements of a predicate. Though these superscripts are placed on subgoals in our examples, they should be considered as global specifications for the predicate in the whole program. If the subgoal has no superscripts, this means the underlying predicate has no mode requirements (equivalent to having ? for all variables). If it has a set of mode vectors that means any one of them can be used to satisfy the dataflow requirements of the predicate, *e.g.*, $\mathtt{p}^{\{+?,?+\}}\mathtt{(X,Y)}$. This multiplicity may arise from multiple implementations backing the predicate or, as we explore below, due to different ordering of subgoals leading to different moding requirements for user-defined predicates.

Invocation safety is predicated on bound arguments, so instead of working with mode vectors directly, we use *constraints*.

Definition 3.1 (Constraint). For a predicate, p, any subset of its argument positions, $1 \leq i \leq \mathit{arity}(p)$, forms an *atomic constraint*. A set of atomic constraints which is minimal under the subset relation is a *constraint*. Thus, the domain of constraints for a predicate p is $D_p = \{\min S \mid S \subseteq \mathcal{P}(\{i \mid 1 \leq i \leq \mathit{arity}(p)\})\}$. Throughout this text C is used to range over constraints and AC over atomic constraints.

Definition 3.2 (Mode requirement semantics). A mode vector is translated into an atomic constraint by taking the set of indices for which the mode is +. A set of mode vectors is converted to a constraint by translating each mode vector and removing all super sets. This translation is done by $[\![-]\!]$. For example, $[\![\{\mathtt{++?}, \mathtt{?++}\}]\!]$ is $\{\{1,2\},\{2,3\}\}$. Given a *mode function*, mv, from predicates to a set of mode vectors, $[\![-]\!]_F$ is defined as $[\![-]\!] \circ mv$. We use mv and f to range over mode and constraint functions respectively.

Definition 3.3 (Ill constraint). A constraint holds alternative moding requirements and if this set is empty, there are no alternatives that can be used for safe predicate invocation. Hence, $\emptyset$ for any constraint domain is the unique *ill constraint*.

Definition 3.4 (Trivial constraint). The trivial constraint contains an atomic constraint that does not require any variables to be bound. So for any constraint domain, we say $\{\emptyset\}$ is the unique *trivial constraint*.

3.2 Adornments and ordering

Program adornment annotates the clauses of a program with binding patterns on their arguments. We use this to formalise well-modedness. We use a generalised form of adornment which is predicated on a subgoal reordering function. This generalisation allows different binding patterns to be produced for subgoals depending on the ordering and allows us to derive a versatile well-modedness definition. The following definitions define the transformation at a high-level as used in the rest of the paper.

Definition 3.5 (Adornment). An *adornment* associates to an argument/parameter of an atomic formula either f or b indicating the binding status of the argument/parameter: *free* or *bound*. An *adornment vector* (or a *binding pattern*) for an atomic formula is a vector of adornments whose size matches the predicate's arity. Throughout, $\mathbf{a}, \mathbf{b}$ range over adornment vectors which we index with natural numbers, *i.e.*, $\mathbf{a}_i$ is the i^{th} adornment in the vector.

For some subgoal *sub* of an adorned clause, we denote the adornment vector of *sub* as *adornment*(*sub*).

Definition 3.6 (Ordering). An *ordering* is a bijection between lists of subgoals. When applied to a list of subgoals it permutes them. We use σ to range over orderings.

Definition 3.7 (Generalised clause adornment). Let *adorn* be a function that takes a clause *cl*, a binding pattern $\mathbf{a}$, an ordering σ for the clause, and returns the adorned and reordered (according to σ) version of the clause.

A clause $adorn(cl, \mathbf{a}, \sigma)$ is calculated as follows: first, the clause head is assigned the binding pattern $\mathbf{a}$. Next, the body of the clause *cl* is reordered using σ. Finally, the list of subgoals is traversed left-to-right and adorned. For each argument that is a literal or a variable that is known to be bound, the argument receives the adornment b, otherwise it is given the adornment f. With each processed subgoal, we add all the variables of the subgoal to the list of variables known to be bound.

Example 3.8. Let *cl* be the clause:

```
p(X,Y) :- q(Y,Z), r(X,Y)
```

Given an adornment $\mathbf{a}$ = bf and a local ordering $\sigma(\mathsf{q},\mathsf{r}) = \mathsf{r},\mathsf{q}$, then $adorn(cl, \mathbf{a}, \sigma)$ produces:

```
p(X,Y)^bf :- r(X,Y)^bf, q(Y,Z)^bf
```

Note that now q has Y as bound since it is bound by p during the adornment procedure.

Definition 3.9 (Reordering functions). A *local reordering function* for a particular clause is a function mapping binding patterns (for the clause head) to orderings for that clause. A *global reordering function* maps clauses to local reordering functions.

We use *r* to range over local orderings and *gr* for the global ones.

The idea is that different binding patterns of the clause head can imply different reorderings for the clause. Later (Section 4 and Section 5), we will compute reordering functions such that, given a head binding pattern, adorning the reordered clause left-to-right with respect to this pattern yields subgoal binding patterns that are consistent with mode requirements.

Definition 3.10 (Generalised program adornment). Let *adornProgram* be a function that takes a program *Pr*, a query clause cl_q, and a global reordering function, *gr*.

An adorned version of the program is generated by invoking *adorn* on cl_q, with a binding pattern (adornment vector) with f for each parameter of the query clause and a reordering function $gr(cl_q)$.

For each subgoal in the body, we generate an adorned version of its predicate by using *adorn* on its clauses using the binding pattern given to the subgoal along with the corresponding local reordering from *gr*. This process is repeated for newly generated adorned clauses until no more clauses can be generated.

An adorned program is equivalent to the original program in the answers it computes [2]. When all the local reordering functions are the identity function (preserving source ordering), then *adornProgram* is the traditional adornment transformation.

Example 3.11. Consider the following program *Pr* where q is the query and hash is a built-in hash function and password is the external database predicate to look up a user's password.

```
q(H) :- hashByUser("Rebecca",H).
hashByUser(U,H) :- password(U,P), hash(P,H).
```

where $identity(cl) = \lambda\mathbf{a}.id$ is the global identity reordering, mapping every clause to the identity reordering function for every binding pattern. The adorned program, $adornProgram(Pr, cl_{\mathsf{q}}, identity)$, with the identity reordering contains q^{f} as it is the head of the query clause thus called by no subgoals and $\mathsf{hashByUser}^{\mathsf{bf}}$ since the first argument is bound in the query body. External predicates also receive adornments, but there are no associated clauses to generate. The hash subgoal's first argument is bound due to the earlier use of password which generates a binding for P. The adorned program in its entirety is:

```
q^f(H) :- hashByUser^bf("Rebecca",H).
hashByUser^bf(U,H) :- password^bf(U,P), hash^bf(P,H).
```

3.3 Well-modedness

We define *well-modedness* of a program in terms of adornment. It is defined in both clausal and program scopes.

Definition 3.12 (Mode & adornment consistency). An adornment, $\mathbf{a}$, is consistent with an atomic constraint, *AC*, when *AC* is a set of indices into $\mathbf{a}$ indicating bound adornments alone.

$$\mathbf{a} \blacktriangleleft AC \triangleq \forall i \in AC.\ \mathbf{a}_i = \mathsf{b}$$

The function *findAC* selects all atomic constraints in a constraint that are consistent with a given binding pattern:

$$findAC(\mathbf{a}, C) = \{AC \in C \mid \mathbf{a} \blacktriangleleft AC\}$$

A binding pattern, $\mathbf{a}$, is consistent with a constraint, *C*, when there are some atomic constraints in *C* consistent with $\mathbf{a}$.

$$\mathbf{a} \triangleleft C \triangleq findAC(\mathbf{a}, C) \neq \emptyset$$

Definition 3.13 (Clausal well-modedness). A clause cl is *well-moded* with respect to a constraint function f, a binding pattern $\mathbf{a}$, and a reordering σ, if all subgoal binding patterns of the clause after the adornment procedure are consistent with f.

$$wellModed(cl, \mathbf{a}, f, \sigma) \triangleq$$
$$\forall sub \in body(adorn(cl, \mathbf{a}, \sigma)).\ adornment(sub) \lhd f(pred(sub))$$

Definition 3.14 (Program well-modedness). A program Pr with a goal clause cl_q is *well-moded* with respect to a function f mapping predicts to constraints and a global reordering function gr, if all the subgoals in the adorned program have binding patterns consistent with the constraints in f.

$$wellModedProgram(Pr, cl_q, f, gr) \triangleq$$
$$\forall cl \in adornProgram(Pr, cl_q, gr), sub \in body(cl).$$
$$adornment(sub) \lhd f(pred(sub))$$

This definition of well-modedness permits ordering based transformation of clauses as well as retaining multiple versions of the same clause (with different subgoal orderings).

Somogyi [16] noted modes generalise adornments. This is indeed the case for Prolog which was the subject of their work. For Datalog, however, adornment precisely formalises well-modedness because Datalog does not deal with function symbols as Prolog does. Hence, a variable can only be instantiated to a value at the time of subgoal invocation or not at all, whereas in Prolog context, it is possible to partially instantiate variables, *e.g.* a list of variables, which calls for finer grained moding instead of binary adornments to fully express dataflow behaviour of predicates.

3.4 Properties of consistency

We briefly cover several results on the definition of mode consistency which will be of use in later results.

Lemma 3.15 (Ill and trivial constraints). *The trivial constraint $\{\emptyset\}$ is consistent with all binding patterns and the ill constraint $\emptyset$ is consistent with none.*

We define a partial order on constraints and functions that output constraints (constraint functions). Later, we establish monotonicity of various functions and operators to prove termination of the analysis in Theorem 5.11.

Definition 3.16. Let C_1 and C_2 be two constraints for the same predicate. If every adornment that is consistent with C_1 is also consistent with C_2 pointwise, we define a relation $\leq$ and say C_1 is less restrictive than C_2. Let $\preceq$ be the pointwise extension of $\leq$ to constraint functions.

$$C_1 \leq C_2 \triangleq \forall \mathbf{a}.\ \mathbf{a} \lhd C_2 \implies \mathbf{a} \lhd C_1$$
$$f \preceq g \triangleq \forall p.\ f(p) \leq g(p)$$

Lemma 3.17. *For a fixed predicate p, $\leq$ is a bounded partial order (PO) with $\emptyset$ as the top element and $\{\emptyset\}$ as the bottom element. For a fixed domain $\preceq$ is also a bounded partial order with constant functions $\emptyset$ and $\{\emptyset\}$ as top and bottom elements respectively.*

The partial order defined by consistency with binding patterns implies a subset relation between atomic constraints of two constraints.

Lemma 3.18 (PO to atomic constraint relation). *If a constraint C_2 is more restrictive than C_1, this means C_1 has a more relaxed atomic constraint for each atomic constraint in C_2.*

$$\forall C_1, C_2.\ C_1 \leq C_2 \implies \forall AC \in C_2\ \exists AC' \in C_1.\ AC' \subseteq AC$$

4 INTRA-CLAUSAL ANALYSIS

We start mode analysis by considering individual clauses of a predicate in isolation, deriving a moding constraint (Definition 3.1) for a clause in terms of its body and constraints of its subgoals alone.

A goal of this analysis is to perform better than brute-force search, we do this by following the path of least resistance, that is: as soon as we find some subgoal that can be scheduled without any constraints, we commit to it. This may discard some valid orderings of clauses but always leaves us at least one valid ordering that makes the program well-moded if such an ordering exists. This analysis is performed by a graph construction.

Before we explain the construction of this graph, we first characterise a more general graph structure and explain how orderings are stored in it.

4.1 Scheduling graph

A scheduling graph encodes orderings of a clause's subgoals. We work towards its formal definition and then explain how orderings can be retrieved from it. The purposes of introducing scheduling graphs rather than the specific construction used in the analysis are twofold. First, it assists in our proof of completeness (Lemma 4.34). Second, it provides a framework that abstracts most details of the construction and allows us to focus on subgoal choices alone.

Before we can construct a scheduling graph, we need to translate between a constraint which is given in terms of argument positions of predicates and a clause context which contains variables.

Definition 4.1 (Obligation). An *obligation*, as opposed to a constraint, is a set of variables that a subgoal is constrained upon. The empty set is the *trivial obligation.*

For example, given a subgoal `p(X,Y,Z)` constrained in its first and third arguments, then $\{1, 3\}$ is the atomic constraint and $\{\mathtt{X}, \mathtt{Z}\}$ is the obligation associated with it. We use two functions to go back and forth between sets of obligations and constraints (sets of atomic constraints): for an atomic formula p, $osToC_p$ maps a set of obligations to a constraint and $cToOs_p$ goes in the other direction.

Definition 4.2 (Scheduling graph). A *scheduling graph* g for a clause cl and a constraint function f over predicates in cl is a *directed acyclic graph* with a set of vertices $\mathbf{V}(g)$ and edges $\mathbf{E}(g)$.

An edge is a triple of a source vertex, label, and destination vertex. The label is a non-empty set of pairs comprising a subgoal from the body of cl and an obligation being discharged. We assume *src*, *label*, and *dst* helper functions to access components of edges. The *paths* function give paths of the graph.

A vertex is a tuple of the form (Alt, Acc). The set Alt stores *alternatives* given by a set of tuples of the form (s, Obg) where s is a subgoal of the clause and Obg is a set of obligations. The alternatives represent what can be scheduled after a given point in the graph. The set of obligations that are coupled with the subgoals represent the variables that can be bound to satisfy the moding constraints of this subgoal's predicate.

The second component of the vertex, *Acc*, is an accumulated obligation, keeping track of the variables that have to be bound at the head of the clause. As the head of the clause needs to bind these variables, the following constraint is imposed $Acc \subseteq vars(head(cl))$.

An example scheduling graph can be seen in Figure 1.

Furthermore, for every edge in a scheduling graph:

$$(Alt, Acc) \xrightarrow{l} (Alt', Acc')$$

the vertices and edge satisfy the following three properties which are jointly referred to as the *valid scheduling* property:

(1) The accumulator of the target vertex extends the accumulator of the source with the obligations of the label:

$$Acc' = nextAcc(Acc, l)$$
$$where \quad nextAcc(Acc, l) \triangleq Acc \cup \bigcup obgs(l)\}$$

where $obgs(l) \triangleq \{o \mid (s, o) \in l\}$ Thus scheduling adds a binding requirement that needs to be resolved in the head of the clause.

(2) Alternatives in the target are computed from source alternatives:

$$Alt' = nextAlt(Alt, l)$$

where

$$nextAlt(Alt, l) \triangleq \{(s, Obg) \in Alt \mid s \notin subs(l)\} \ominus \bigcup_{s \in subs(l)} vars(s)$$

with $subs(l) \triangleq \{s \mid (s, o) \in l\}$. The set of alternatives in the target has the labelling subgoals removed and variables in the subgoals within the label are "released", by the operator $\ominus$:

$$A \ominus Vars \triangleq \{(s, \min\{o \setminus Vars \mid o \in Obg\}) \mid (s, Obg) \in A\}$$

The release operator removes the variables it receives from the obligations of the alternatives and minimises the set of obligations to remove redundancies.

Example 4.3. Let `f(X,Y,Z)` be a subgoal and *Alt* be the alternative set $\{(\mathtt{f}, \{\{\mathtt{X},\mathtt{Y},\mathtt{Z}\}, \{\mathtt{X},\mathtt{Z}\}, \{\mathtt{Y},\mathtt{Z}\}\})\}$ be a set of alternatives. Releasing the obligation $\{\mathtt{Z}\}$ by $Alt \ominus \{\mathtt{Z}\}$ yields $\{(\mathtt{f}, \{\{\mathtt{X}\}, \{\mathtt{Y}\}\})\}$ which retains only the minimal obligations of `f(X,Y,Z)`.

(3) We require that the edge labels are selected from the alternatives of the preceding vertex:

$$\forall (s, o) \in l,\ \exists (s', Obg) \in Alt.\ s = s' \wedge o \in Obg$$

Finally, a scheduling graph has a root vertex, $Root_{cl,f}$, unique to the clause and the constraint function. It is a transcription of the predicate context into clause context coupled with an empty accumulator.

$$Root_{cl,f} \triangleq (\{(\{(s, \min cToOs_s(f(pred(s)))) \mid s \in body(cl)\}, \emptyset)$$

Although the constraint being translated is minimal by definition (Def. 3.1), we minimise the sets of obligations in the alternatives after translation because two indices may point to the same variable, hence creating a subset relation that did not exist between atomic constraints. For example, $\{\{1, 2\}, \{2, 3\}\}$ for subgoal `p(X,X,Y)` produces the alternative obligation $\min\{\{X\}, \{X, Y\}\}$, thus $\{\{X\}\}$.

If a path in a scheduling graph has labels covering every subgoal of a clause, then the path represents a clause ordering. Such paths are characterised by having a terminal vertex:

Definition 4.4 (Terminal vertex). A terminal vertex is of the form $(\emptyset, Acc)$ for some obligation *Acc*. The following predicate checks if a path has a terminal vertex:

$$terminal(p) \triangleq \exists Acc.\ (\emptyset, Acc) \in \mathbf{V}(p)$$

If a path in a scheduling graph has a terminal vertex, it has to be the last vertex on the path since edge labels are chosen from the alternatives. Further, all subgoals must have been scheduled in the edges preceding the terminal vertex.

Paths from the root of a scheduling graph to its terminal vertices represent subgoal orderings, which we will extract. Since the labels in the graph may have multiple subgoals, paths in the graph form *compact orderings*. Each edge expands to all permutations of its members. Permuting each edge on a path and concatenating the resulting ordering fragments lead to orderings of a whole clause. Let *orderings* be the function that returns all orderings stored in a path. For example, a sequence of first projections of labels (subgoals) $\{\mathtt{p}\}, \{\mathtt{q},\mathtt{r}\}, \{\mathtt{s},\mathtt{t}\}$ leads to orderings mapping the syntactic order of the subgoals to the following:

`p,q,r,s,t` `p,r,q,s,t` `p,q,r,t,s` `p,r,q,t,s`

Using these definitions, we define *well-modedness* of paths.

Definition 4.5 (Well-moded path). A path *p* in a scheduling graph constructed for clause *cl* is *well-moded* with respect to a binding pattern **a** and constraint function *f* when all orderings in the path make the clause well-moded. The path also has to be terminal as adornment has to know where to place *all* the subgoals.

$$wellModedPath(p, cl, \mathbf{a}, f) \triangleq$$
$$terminal(p) \ \wedge \ \forall \sigma \in orderings(p).\ wellModed(cl, \mathbf{a}, f, \sigma)$$

4.2 Minimal obligation graph

Due to the constraints of scheduling graphs, a valid graph is essentially determined by choices of which subgoals to schedule at each edge using a particular obligation from the preceding set of alternatives. Here, we describe the construction of a particular scheduling graph referred as *minimal obligation graph* (MOG). The property of this scheduling graph is that it *greedily* (Definition 4.11) chooses the paths with trivial obligations as soon as possible.

Definition 4.6 (Minimal obligation graph). For a clause *cl* and a constraint function *f* over predicates in *cl*, $\mathrm{mog}_{cl,f}$ is a *minimal obligation graph* sharing its label and edge structure with that of a scheduling graph.

We construct $\mathrm{mog}_{cl,f}$ in a breadth-first manner. To aid understanding of the formal definition, we start with a complete example of a MOG constructed for an example clause, and then use this to expound on the definition of the $\mathrm{mog}_{cl,f}$ algorithm.

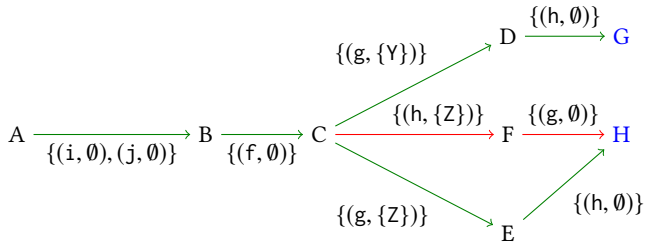
Example 4.7. Consider the following clause cl_r annotated with mode vectors:

`r(Y,Z) :-` $\mathtt{f}^{+}(\mathtt{X})$, $\mathtt{g}^{\{++?,+?+\}}(\mathtt{X},\mathtt{Y},\mathtt{Z})$, $\mathtt{h}^{+}(\mathtt{Z})$, `i(X), j(X,W).`

Subsequently, the moding annotations induce a constraint function *f* mapping predicate names to constraints defined:

$f(\mathtt{f}) = \{\{1\}\}$ $f(\mathtt{g}) = \{\{1,2\},\{1,3\}\}$ $f(\mathtt{h}) = \{\{1\}\}$ $f(\mathtt{i}) = f(\mathtt{j}) = \{\emptyset\}$

Fig. 1 then shows the MOG constructed by $\mathrm{mog}_{cl_r,f}$.



$A = (\{(\mathtt{f}, \{\{\mathtt{X}\}\}), (\mathtt{g}, \{\{\mathtt{X}, \mathtt{Y}\}, \{\mathtt{X}, \mathtt{Z}\}\}), (\mathtt{h}, \{\{\mathtt{Z}\}\}), (\mathtt{i}, \{\emptyset\}), (\mathtt{j}, \{\emptyset\})\}, \emptyset)$

$B = (\{(\mathtt{f}, \{\emptyset\}), (\mathtt{g}, \{\{\mathtt{Y}\}, \{\mathtt{Z}\}\}), (\mathtt{h}, \{\{\mathtt{Z}\}\})\}, \emptyset)$

$C = (\{(\mathtt{g}, \{\{\mathtt{Y}\}, \{\mathtt{Z}\}\}), (\mathtt{h}, \{\{\mathtt{Z}\}\})\}, \emptyset)$

$D = (\{(\mathtt{h}, \{\emptyset\})\}, \{\mathtt{Y}\})$ $E = (\{(\mathtt{h}, \{\emptyset\})\}, \{\mathtt{Z}\})$ $F = (\{(\mathtt{g}, \{\emptyset\})\}, \{\mathtt{Z}\})$

$G = (\emptyset, \{\mathtt{Y}\})$ $H = (\emptyset, \{\mathtt{Z}\})$

Clause:

`r(Y,Z) :-` $\mathtt{f}^{+}(\mathtt{X})$, $\mathtt{g}^{\{++?,+?+\}}(\mathtt{X},\mathtt{Y},\mathtt{Z})$, $\mathtt{h}^{+}(\mathtt{Z})$, `i(X), j(X,W).`

Figure 1: Minimal obligation graph for a clause of `r`

MOGs are constructed iteratively starting from a root node, we write $\text{mog}^{i}_{cl,f}$ for the i^{th} step of this process.

Base case. All MOGs stem from a graph of a root vertex (as in Definition 4.2) and no edges, defined:

$$\mathbf{V}(\text{mog}^{0}_{cl,f}) = \mathit{Root}_{cl,f} \qquad \mathbf{E}(\text{mog}^{0}_{cl,f}) = \emptyset$$

Thus $\text{mog}^{0}_{cl,f}$ initiates the breadth-first construction of the graph by adding no edges and one root. In the running example, the root vertex is called A. The alternative set of the root is a translation from predicate constraints to the clause context. For example, g has the constraint $\{\{1, 2\}, \{1, 3\}\}$ and thus `g(X,Y,Z)` is represented by $\{\{\mathtt{X}, \mathtt{Y}\}, \{\mathtt{X}, \mathtt{Z}\}\}$. The accumulator component of the vertex is empty.

Inductive step. We explore the MOG one unit distance from the root at a time. Say we are at distance $n + 1$, Vertices are derived from the destination vertices of the edges at distance $n + 1$. Edges are added for each vertex at distance n.

$$\mathbf{V}(\text{mog}^{n+1}_{cl,f}) = \{\mathit{dst} \mid (\mathit{src}, l, \mathit{dst}) \in \mathbf{E}(\text{mog}^{n+1}_{cl,f})\}$$

$$\mathbf{E}(\text{mog}^{n+1}_{cl,f}) = \bigcup_{v \in \mathbf{V}(\text{mog}^{n}_{cl,f})} \{\mathit{mkEdge}(v, l) \mid l \in \mathit{pickLabel}_{cl}(\pi_1\ v)\}$$

where *mkVertex* is defined via scheduling graph constructions (Def. 4.2):

$$\mathit{mkEdge}(\mathit{src}, l) \triangleq (\mathit{src}, l, \mathit{mkVertex}(\mathit{src}, l))$$

$$\mathit{mkVertex}((\mathit{Alt}, \mathit{Acc}), l) \triangleq (\mathit{nextAlt}(\mathit{Alt}, l), \mathit{nextAcc}(\mathit{Acc}, l))$$

From a given vertex, we decide what subgoals to schedule (what edge label to generate) using *pickLabel*. We give preference to scheduling of natural subgoals within the alternative set over the others. We explain what each case of these label choices means.

$$\mathit{pickLabel}_{cl}(\mathit{Alt}) \triangleq \begin{cases} \{\mathit{nats}(\mathit{Alt})\} & \mathit{nats}(\mathit{Alt}) \neq \emptyset \\ \mathit{nonNats}_{cl}(\mathit{Alt}) & \text{otherwise} \end{cases}$$

Case 1: Extending the graph with natural edges. The *natural subgoals* at a given vertex are those that are in the alternative set of the vertex and are paired with a singleton set of trivial obligation. This effectively means that the binding requirements of this subgoal are satisfied at this point.

$$\mathit{nats}(\mathit{Alt}) \triangleq \{(\mathit{sub}, \emptyset) \mid (\mathit{sub}, \{\emptyset\}) \in \mathit{Alt}\}$$

In the example, applying $\mathit{nats}(\pi_1\ A)$ yields: $\{(\mathtt{i}, \emptyset), (\mathtt{j}, \emptyset)\}$. This set is collectively used by *mkEdge* as a label. Thus, the subgoals `i` and `j` are scheduled between the nodes A and V.

One consequence of natural subgoals all having the trivial obligation in the edge label is that *nextAcc* does not add new obligations to the accumulator in the destination vertex generated. In the example, this is why the accumulator of B remains as the empty set.

Naturality of a subgoal is contextual. In the example, `f` is not natural in vertex A, but as a result of releasing variables of `i` and `j`, it becomes natural B. Hence, it is scheduled as a natural subgoal.

Case 2: Extending the graph with non-natural edges. If there are no natural subgoals to scheduled the *pickLabel* function tries alternatives with non-trivial obligations. Unlike the natural case, labels picked in this manner have a single subgoal obligation pair in them. Formally, they are selected as follows:

$$\mathit{nonNats}_{cl}(\mathit{Alt}) \triangleq \{\{(s, o)\} \mid (s, os) \in \mathit{Alt}, o \in os, o \subseteq \mathit{vars}(\mathit{head}(cl))\}$$

A side condition for selecting a label this way is to check if the obligation is a subset of the head variables. This ensures that it is possible to bind the variable in the head.

In the example, all edges from C are created in this manner as all the alternatives C have non-empty obligations.

Unlike the natural case, *nextAcc* potentially augments the accumulator in the destination vertex since labels of this kind have non-empty obligations; the caller has to remove these obligations. In the example, vertices D, F, and E are generated using this function and since at C the accumulator is empty, in these three vertices the accumulator is the single obligation specified in the label.

Building the whole graph. Full $\text{mog}_{cl,f}$ is given as a union of its components.

$$\mathbf{V}(\text{mog}_{cl,f}) = \bigcup_{n \geq 0} \mathbf{V}(\text{mog}^{n}_{cl,f}) \qquad \mathbf{E}(\text{mog}_{cl,f}) = \bigcup_{n \geq 0} \mathbf{E}(\text{mog}^{n}_{cl,f})$$

Lemma 4.8. *MOG construction leads to a scheduling graph.*

Lemma 4.9 (MOG termination). *For any clause cl, and any constraint function f, the MOG construction $\text{mog}_{cl,f}$ terminates.*

The only scenario where a clause cannot lift its constraints to the caller is if its subgoals have variables that do not appear in the head. MOG construction gets stuck in this case.

Example 4.10 (Stuck construction). Consider the following clause with the given moding requirements:

```
1  r(X) :- g+(Y), f+(X).
```

These moding requirements cannot be satisfied by any ordering of subgoals as the variable `Y` cannot be bound by just binding `X`. This is shown below by the MOG for this clause, which fails to contain any paths that schedule all the subgoals of the clause.

After the initial step, the root vertex contains the set of alternatives: $\{(\mathtt{f}, \{\{\mathtt{X}\}\}), (\mathtt{g}, \{\{\mathtt{Y}\}\})\}$. Since none of these are natural subgoals, *pickLabel* selects a non-natural edge which requires augmenting the accumulator. Only one edge is possible since only one obligation has all of its variables in the head of the clause, namely `f`. This yields the MOG:

$$(\{(\mathtt{f}, \{\{\mathtt{X}\}\}), (\mathtt{g}, \{\{\mathtt{Y}\}\})\}, \emptyset) \xrightarrow{\{(\mathtt{f}, \{\mathtt{X}\})\}} (\{(\mathtt{g}, \{\{\mathtt{Y}\}\})\}, \{\mathtt{X}\})$$

There are no further edges that can be generated (*i.e.*, $\mathbf{V}(\mathrm{mog}^2_{cl,f}) = \mathbf{E}(\mathrm{mog}^2_{cl,f}) = \emptyset$) and the MOG has no terminal vertices; there is no full ordering that makes this clause well-moded with respect to the given constraints.

A path is considered greedy if it schedules natural subgoals as soon as possible.

Definition 4.11 (Greedy paths). A path p is said to be *greedy* if for any edge p_i with a source vertex v and a label l, whenever v has a natural subgoal *sub* ($(sub, \{\emptyset\})$ in the alternative set), $(sub, \emptyset)$ appears in the label, l, of the edge or equivalently does not appear in the alternatives of the destination vertex.

$$greedy(p) \triangleq \forall i, s.\ (s, \{\emptyset\}) \in \pi_1\ p_i \implies \forall Alt.\ (s, Alt) \notin \pi_1\ p_{i+1}$$

Proposition 4.12. *A MOG is a scheduling graph where all paths are greedy.*

Proof. Requirements of a scheduling graph are satisfied trivially by the MOG construction. Natural edge generation by *pickLabel* takes precedence, therefore all paths satisfy greediness. □

4.2.1 Optimising MOG construction. MOG construction is described here in a breadth-first manner. A depth-first reconstruction is possible which, when accompanied with some global state, may prune the graph. First, we switch to depth first-construction by branching only when we reach a terminal vertex or there is no possible edge to add to a path. Second, we maintain a global list of accumulator components of terminal vertices we encounter so far. Each time we extend a path (via natural subgoals or otherwise), we check if the accumulator component of the newly created vertex is a superset of any member of the global list. If so, then we do not create that vertex at all as we already have a path that is at least as good as the one we are about to explore.

In Example 4.7, the red edges show the edges that would not have been constructed in this approach. If the terminal vertex G is discovered first, the global list contains $\{\mathtt{Y}\}$, therefore when we try to branch from C, we realise F's accumulator is a superset of $\{\mathtt{Y}\}$ and do not add the edge at all. E, on the other hand, is explored as before since its accumulator $\{\mathtt{Z}\}$ is not a superset of $\{\mathtt{Y}\}$.

The upside of this change is that we have fewer vertices to explore, the downside is, in the end, we will have fewer orderings to choose from for a given binding pattern and which orderings are retained depends on which terminal vertex is encountered first.

4.3 Extracting a clause constraint out of a MOG

The second component of a MOG vertex is an accumulated obligation. Using this, we can derive a new constraint for the given clause. Only terminal vertices contribute to constraint derivation.

The obligation in a terminal vertex is a viable option for the caller to resolve. This notion of choice between obligations must be expressed with no reference to concrete variable names as these are clause scoped (*i.e.*, the bound names are irrelevant outside the definition). Hence, we convert obligations back into singleton constraints and combine them using the $\otimes$ operator.

Definition 4.13. If A and B are in the constraint domain D_p. We define $A \otimes B$ as $\min(A \cup B)$.

We take the union of the two constraints and min eliminates the redundant constraints, *i.e.*, min eliminates more restrictive atomic constraints in the union.

Example 4.14. Consider the following mode-annotated clause:

```
r(X,Y) :- f++(X,Y), g+?(X,Y).
```

For a constraint function f derived from the moding, then $\mathrm{mog}_{\mathtt{r},f}$:

$$\begin{array}{l} \{(\mathtt{f}, \{\mathtt{X}, \mathtt{Y}\})\} \quad (\{\{(\mathtt{g}, \{\emptyset\})\}, \{\mathtt{X}, \mathtt{Y}\}) \xrightarrow[\{(\mathtt{g}, \emptyset)\}]{} (\emptyset, \{\mathtt{X}, \mathtt{Y}\}) \\ (\{(\mathtt{f}, \{\{\mathtt{X}, \mathtt{Y}\}\}), (\mathtt{g}, \{\{\mathtt{X}\}\})\}, \emptyset) \\ \{(\mathtt{g}, \{\mathtt{X}\})\} \quad (\{(\mathtt{f}, \{\emptyset\})\}, \{\mathtt{X}\}) \xrightarrow{\{(\mathtt{f}, \emptyset)\}} (\emptyset, \{\mathtt{X}\}) \end{array}$$

The clause `r` receives a different constraint depending on the order of the subgoals. If `f` is first (as it is in the source), `X` and `Y` together are the obligation, hence we constrain both argument positions and get the constraint {{1,2}}. If `g` is first, only `X` is the obligation as use of `g` binds `Y` and satisfies the moding requirement of `f`. Thus, for this ordering we have the constraint {{1}}. It is clear the latter ordering is more favourable as it leads to a more relaxed (smaller) constraint. Hence, the $\otimes$ operator eliminates the former.

$$\{\{1, 2\}\} \otimes \{\{1\}\} = \min(\{\{1, 2\}\} \cup \{\{1\}\}) = \{\{1\}\}$$

Definition 4.15 (Constraint generated by a MOG). The $\otimes$ operator combines translations of terminal vertices into constraints to derive an overall constraint for the clause from the MOG, defined as the following function *extract*:

$$extract_{cl}(mog) \triangleq \bigotimes_{(\emptyset, o) \in \mathbf{V}(mog)} osToC_{head(cl)}(\{o\})$$

Example 4.16. In the running example from Figure 1, the terminal vertices are $G = (\emptyset, \{\mathtt{Y}\})$ and $H = (\emptyset, \{\mathtt{Z}\})$ therefore:

$$extract_{cl_\mathtt{r}}(\mathrm{mog}_{cl_\mathtt{r},f}) = \{\{1\}\} \otimes \{\{2\}\} = \{\{1\}, \{2\}\}$$

i.e., either the first or second parameter of the head clause `r` should be bound.

Lemma 4.17. *The extract function is monotonically increasing.*

$$\forall cl, f, g.\ f \preceq g \implies extract_{cl}(\mathrm{mog}_{cl,f}) \leq extract_{cl}(\mathrm{mog}_{cl,g})$$

4.4 Extracting subgoal orders out of a MOG

Now that we have a constraint for the clause, we explain how to build a partial function that gives the subgoal order of a clause for a given adornment. It is a partial function as for some adornments no ordering of subgoals leads to a well-moded program. So it will only be defined for adornments that are consistent with the constraint of the clause.

We define a function $collect_{mog}$, parameterised by a MOG, which maps atomic constraints to sets of subgoal orderings. It does it in two steps. First, it finds all paths from the root to the terminal vertices with accumulated obligations such that these obligations correspond to the atomic constraint. Second, it expands the compact orderings represented by these paths into subgoal orderings.

In general, a MOG may provide multiple valid orderings for binding patterns as there may be multiple distinct atomic constraints consistent with a given adornment. We assume a *choose* function that selects from a set of orderings. This function can be based on a metric defined on orderings. For example, it may use a metric that measures the distance from the original source ordering and pick the one that is closest. This is useful for preserving some optimisations based on estimated relation sizes.

Equipped with these definitions, we build a partial function that gives an ordering of a clause for a particular adornment under a constraint function f:

$$reordering_{cl,f}(\mathbf{a}) \triangleq choose(\bigcup_{AC \in C} collect_{\mathrm{mog}_{cl,f}}(AC)) \\ where\ C = findAC(\mathbf{a}, extract_{cl}(\mathrm{mog}_{cl,f}))$$

4.5 Results

Lemma 4.18. *If any of the subgoals of a clause cl have the ill constraint according to constraint function f, the constraint for the clause is also ill.*

$$\forall sub \in body(cl).\ f(pred(sub)) = \emptyset \implies extract_{cl}(\mathrm{mog}_{cl,f}) = \emptyset$$

Lemma 4.19 (Intra-clausal soundness). *For a given clause, cl with head predicate q and a constraint function, f, if an adornment is consistent with the constraint generated by intra-clausal analysis, then the adornments of all the subgoals of the adorned clause (with the reordering function from the intra-clausal analysis) are also consistent with respect to f.*

$$\forall \mathbf{a}.\ \mathbf{a} \lhd extract_{cl}(\mathrm{mog}_{cl,f}) \implies wellModed(cl, \mathbf{a}, f, reorder_{cl,f}(\mathbf{a}))$$

Proof. Pick an arbitrary adornment $\mathbf{a}$ that is consistent with the extracted constraint from the MOG, *i.e.*, $\mathbf{a} \lhd extract_{cl}(\mathrm{mog}_{cl,f})$ (see Definition 3.12 for $\lhd$). This implies $reordering_{cl,f}$ is defined at $\mathbf{a}$, which can only happen if there is a path p in $\mathrm{mog}_{cl,f}$ leading to this ordering.

We proceed by establishing a contradiction if p represents an unsound path *i.e.* there is at least one subgoal on the path with an adornment inconsistent with the constraint function f.

Let $\mathtt{s}(\mathbf{X})$ to be the earliest subgoal (where $\mathbf{X}$ is a list of variables) that is inconsistent with its constraint after adorning the clause with $\mathbf{a}$ and p_i be the edge that contains this subgoal in its label. Let $\mathbf{b}$ be that inconsistent binding pattern of $\mathtt{s}$ and C be its constraint from f, then we have $\forall AC \in C.\ \mathbf{b} \not\blacktriangleleft AC$ by definition of consistency. This can only happen if for each atomic constraint there is at least one index j such that $\mathbf{X}_j$ is adorned free because all indices in ACs are bound by definition of $\blacktriangleleft$. Let F be the set of all such offending variables. Since all of F has to be free at the point $\mathtt{s}$ is scheduled, none of F can appear in the head of the clause as bound or as an argument to any subgoals before p_i.

There are two ways $\mathtt{s}$ could have been scheduled. Either in an edge with (possibly) other natural subgoals (extension by natural subgoals) or by extending the accumulated obligation (extension by non-natural subgoals).

Consider extension by natural subgoals, requiring an alternative in the source of p_i of the form $(\mathtt{s}, \{\emptyset\})$. At the root vertex, the alternatives include $(\mathtt{s}, \min\ cToOs_s(C))$. Each obligation in this alternative must have at least one element from F as they are generated from C. This cannot be the case as none of F appeared as an argument before p_i and hence these variables cannot be released with $\ominus$, thus we cannot obtain the alternative $(\mathtt{s}, \{\emptyset\})$ or schedule $\mathtt{s}$ at p_i.

Consider extension by non-natural subgoals. As before we know that the alternatives at the source of p_i each contain at least one member of F. This means regardless of which alternative is used, the accumulated obligation at the destination of p_i must contain a member of F. Hence, the terminal vertex's accumulator must contain at least one member of F. All variables in this accumulator must be *bound* in the head due to the fact that the reordering is generated using a terminal vertex with an accumulated obligation corresponding to an atomic constraint of the head. This contradicts our assumption that the offending variable is *free* in the head.

Since having an inconsistent subgoal in the path contradicts the formation of the path, all subgoals must be consistent with their constraints when the head is adorned with $\mathbf{a}$. Since $\mathbf{a}$ was arbitrary, the lemma holds. □

Intra-clausal completeness is more involved. It states that:

$$\forall \mathbf{a}.\ (\exists r.\ wellModed(cl, \mathbf{a}, f, r(\mathbf{a}))) \implies \mathbf{a} \lhd extract_{cl}(\mathrm{mog}_{cl,f})$$

(full statement in Lemma 4.34). That is, if there exists a local ordering r that makes a clause well-moded with respect to head binding pattern $\mathbf{a}$, then $\mathbf{a}$ is consistent with the constraint computed by our analysis. We prove completeness by showing than an arbitrary ordering r is captured (in some way) by our MOG construction, *i.e.*, that $\mathrm{mog}_{cl,f}$ is complete. This involves first converting arbitrary orderings to scheduling graphs, and showing that paths in such a graph can be transformed into effectively equivalent paths in our MOG. Key to this is the property that MOG paths are *greedy*.

Definition 4.20 (Ordering to scheduling graph). Given a clause cl with n subgoals, a constraint function f defined for all predicates invoked in cl, a binding pattern $\mathbf{a}$, and an ordering σ of cl giving an adorned and reordered clause $cl' = adorn(cl, \mathbf{a}, \sigma)$, then there is a scheduling graph $g = schedule(cl, \mathbf{a}, f, \sigma)$, where:

$$\mathbf{V}(g) = \bigcup_{0 \le i < n} V_i \qquad \mathbf{E}(g) = \bigcup_{0 \le i < n} E_i$$

$$V_0 = Root_{cl,f} \qquad E_0 = \emptyset \qquad V_i = dst(E_i)$$

$$E_i = \{mkEdge(v, (s, o)) \mid v \in V_{i-1}, (s, obgs) \in v, s = {cl'}_i, \\ o \in obgs, adornment(s) \lhd osToC_s(\{o\})\}$$

Lemma 4.21 (Well-moded ordering to terminal path). *For a constraint function f, clause cl, binding pattern $\mathbf{a}$, and ordering σ where $wellModed(cl, \mathbf{a}, f, \sigma)$, then a generated scheduling graph $g = schedule(cl, \mathbf{a}, f, \sigma)$ has a path $p \in paths(g)$ where $terminal(p)$.*

Definition 4.22 (Conversion). We convert scheduling graph paths, where each edge has a singleton set label, into greedy paths.

Find the earliest vertex, p_i, that has a subgoal with the trivial obligation in its alternatives. For all such subgoals in p_i, use the *swap* operation (below) to place them in adjacent edges starting from p_i in any order. Use *merge* repeatedly to merge all these edges. Repeat this process until the path cannot be changed anymore.

Definition 4.23 (Swap). The *swap* operation on a path p in a scheduling graph takes an index i and assuming p_i and p_{i+1} exist, produces a new path where the subgoals in the edge p_i comes before the subgoals in the edge p_{i+1}. The operation is applied when all the subgoals in the edge p_{i+1} have trivial obligations in p_i and consequently at p_{i+1}.

Let L be the edge label at p_i and R be the edge label at p_{i+1}.The new path is q is as follows:

$$\begin{aligned}\forall j < i.\ q_j &= p_j\\ q_j &= mkEdge(src(p_j), R)\\ q_{j+1} &= mkEdge(dst(q_j), \{(s, o \setminus \bigcup_{(s',o)\in R} vars(s')) \mid (s, o) \in L\})\\ \forall j > i+1.\ q_j &= mkEdge(dst(q_{j-1}), label(p_j))\end{aligned}$$

Lemma 4.24 (Trivial obligation consistency). *If a scheduling graph vertex v has a subgoal s with obligation $\{\emptyset\}$ in its alternative set, any ordering with a partial ordering derived from the root to v has the predicate s consistent with respect to any binding pattern.*

Lemma 4.25 (Swap well-modedness). *For a path p well-moded with respect to a binding pattern $\mathbf{a}$ and some constraint function the swap operation preserves well-modedness.*

Lemma 4.26 (Swap scheduling path). *If a path p is in a scheduling graph before swapping, it remains to be in one after.*

Definition 4.27 (Merge). Let p be a path in a scheduling graph, a *merge* of edges p_i and p_{i+1} removes them both and replaces it with a single edge with a label that is the union of all the labels. The operation is applied if and all subgoals in both of these edges have the trivial obligation at the source of p_i. Let q be the resulting path with the following specification:

$$\begin{aligned}\forall j < i.\ q_j &= p_j\\ q_i &= (src(p_i), label(p_i) \cup label(p_{i+1}), dst(p_{i+1}))\\ \forall j > i.\ q_j &= p_{j+1}\end{aligned}$$

Lemma 4.28 (Merge well-modedness). *If a path in a scheduling graph is well-moded with respect to some adornment and a constraint function, then the path produced by a merge is also well-moded.*

Lemma 4.29 (Merge scheduling graph). *If p is a path in a scheduling graph, a merge performed somewhere on this path produces another path the is also in a scheduling graph.*

Lemma 4.30 (Conversion greed). *If a path is in a scheduling graph, then its conversion is a greedy path in a scheduling graph.*

Proof. Swap and merge preserve scheduling graph structure. By construction subgoals in edges are positioned such that they follow the trivial obligation, since a subgoal cannot appear in more than one edge, greediness requirement is satisfied. □

Lemma 4.31 (Conversion preserves well-modedness). *If the path is well-moded, then so is the converted path.*

Thus, we have shown that conversion creates greedy scheduling paths, and preserves well-modedness of paths. We then show such paths are in the MOG (Lemma 4.32) and are consistent with adornment (Lemma 4.33), finally leading to completeness.

Lemma 4.32 (Greedy path completeness). *For a clause cl, every greedy scheduling path ending in a terminal vertex and conforming to a constraint function is present in the MOG determined by cl and the constraint function f.*

$$\forall p, \mathbf{a}.\ greedy(p) \wedge wellModedPath(p, \mathbf{a}, cl, f) \implies p \in paths(\mathrm{mog}_{cl,f})$$

Lemma 4.33 (Path extract connection). *For a fixed binding pattern $\mathbf{a}$, existence of a well-moded path in a MOG implies consistency of $\mathbf{a}$ with the constraint extracted from the MOG.*

$$\forall \mathbf{a}\ p.\ wellModedPath(p, \mathbf{a}, cl, f) \wedge p \in paths(\mathrm{mog}_{cl,f}) \implies \mathbf{a} \lhd extract_{cl,f}(\mathrm{mog}_{cl,f})$$

Lemma 4.34 (Intra-clausal completeness). *For a given clause, cl, a constraint function, f, and an adornment $\mathbf{a}$ for the head of cl, if there is a local reordering that makes the adornment of the subgoals consistent with their constraints, the head adornment is consistent with the constraint extracted from the MOG. That is:*

$$\forall \mathbf{a}.\ (\exists r.\ wellModed(cl, \mathbf{a}, f, r(\mathbf{a}))) \implies \mathbf{a} \lhd extract_{cl}(\mathrm{mog}_{cl,f})$$

Proof. Fix an arbitrary adornment $\mathbf{a}$ and assume the antecedent. We need to show that $\mathrm{mog}_{cl,f}$ contains a path that ends in a terminal vertex leading to an atomic constraint consistent with $\mathbf{a}$.

By Definition 4.20, we convert the ordering for $\mathbf{a}$ into a scheduling graph $g = schedule(cl, \mathbf{a}, f, r(\mathbf{a}))$. Since, $wellModed(cl, \mathbf{a}, f, r(\mathbf{a}))$, there exists at least one path p which is terminal in g (Lemma 4.21). From Definition 4.22 (with Lemma 4.30) we convert this path p into a greedy path p', which is terminal and well moded (Lemma 4.31, Lemma 4.30). Since the path p' is well moded and terminal we have that, $wellModedPath(p, \mathbf{a}, cl, f)$ (Definition 4.5).

Using Lemma 4.32 (greedy path completeness), it then follows that $p' \in paths(\mathrm{mog}_{cl,f})$, *i.e.*, that p' is constructed by our MOG-based analysis. Combined with Lemma 4.33 (well-moded MOG path implies extract consistency) then $\mathbf{a} \lhd extract_{cl,f}(\mathrm{mog}_{cl,f})$. □

5 INTER-CLAUSAL ANALYSIS

Having defined how to determine a moding constraint for a given clause, we are ready to find a constraint for a given program comprising multiple predicates each comprising one or more clauses.

Any of the clauses of a predicate can be used to evaluate a subgoal invoking that predicate. This means that for a subgoal invocation to be safe, the bodies of clauses of the invoked predicate must be safe. This implies the constraint of a predicate is a combination of the constraints of its clauses, which we capture with the $\oplus$ operator:

Definition 5.1. If A and B are in the constraint domain D_p, we define $A \oplus B$ as $\min\{a \cup b \mid a \in A, b \in B\}$.

This captures the idea of joint constraints because it produces an atomic constraint for each possible pair of atomic constraints and union ensures the requirements of both clauses are reflected

in the newly generated atomic constraint. As with the $\otimes$ operator, min eliminates the redundancies.

Example 5.2. Consider the following clauses belonging to a predicate `r`, each with the same head:

```
r(X,Y,Z) :- f+??(X,Y,Z).
r(X,Y,Z) :- g{++?,?++}(X,Y,Z).
r(X,Y,Z) :- h??+(X,Y,Z).
```

Use of `r` must reflect the constraints of all these clauses. Individually, the constraints for each clause are $\{\{1\}\}$, $\{\{1,2\},\{2,3\}\}$, and $\{\{3\}\}$ respectively. The only way these three constraints are satisfied is if all three arguments are constrained; $\oplus$ computes this:

$$\{\{1\}\} \oplus \{\{1,2\},\{2,3\}\} \oplus \{\{3\}\} =$$
$$\min \{\{1,2\},\{1,2,3\}\} \oplus \{\{3\}\} = \{\{1,2\}\} \oplus \{\{3\}\} = \{\{1,2,3\}\}$$

Lemma 5.3 ($\oplus$ consistency homomorphism). *A binding pattern is consistent with two constraints combined with $\oplus$ iff that binding pattern is consistent with each constraint individually,* i.e.*:*

$$\forall \mathbf{a}, C_1, C_2.\ \mathbf{a} \lhd (C_1 \oplus C_2) \iff \mathbf{a} \lhd C_1 \wedge \mathbf{a} \lhd C_2$$

Definition 5.4 (Whole-program analysis). Whole-program analysis is then computed as a fixpoint computation over functions from predicates to constraints. In each iteration, the constraints of the clauses with a shared head are combined with the $\oplus$ operator and constraints belonging to predicates without clauses (such as built-in predicates) are preserved without modification.

$$analyse_{Pr}(f) \triangleq \begin{cases} f & \text{if } f = step_{Pr}(f) \\ analyse_{Pr}(step_{Pr}(f)) & \text{otherwise} \end{cases}$$

$$step_{Pr}(f)(p) \triangleq \begin{cases} \bigoplus_{cl \in Pr_p} extract_{cl}(\mathrm{mog}_{cl,f}) & \text{if } Pr_p \neq \emptyset \\ f(p) & \text{otherwise} \end{cases}$$

In the definition of *step*, we use the $\oplus$ operator over constraints for each clause belonging to predicate p, extracted from the MOG by *extract* (Definition 4.15) which is defined in terms of $\otimes$ over constraints.

Note that if $Pr_p = \emptyset$ this implies that the predicate is externally defined, so we default to extracting its constraint $f(p)$ coming from some external mode declaration.

Lemma 5.5. *Every constraint function generated from a moding function gets more restrictive when step is applied to it.*

$$\forall mv.\ [\![mv]\!]_F \preceq step_{Pr}([\![mv]\!]_F)$$

Lemma 5.6 (Step monotonicity). *The step function is monotonically increasing (making constraints more restrictive):*

$$\forall Pr, f, g.\ f \preceq g \implies step_{Pr}(f) \preceq step_{Pr}(g)$$

Proof. Follows from monotonicity of *extract* (Lemma 4.17) and $\oplus$ consistency (Lemma 5.3). □

Lemma 5.7. *Every constraint function generated from a moding function gets more restrictive by application of analyse.*

$$\forall mv.\ [\![mv]\!]_F \preceq analyse_{Pr}([\![mv]\!]_F)$$

Proof. Observe that *analyse* is simply repeated application of *step*. The lemma follows from Lemma 5.5 and Lemma 5.6. □

A necessary property is that $\oplus$ is closed on constraints. This is indeed the case and is implied by the following semiring structure which relates the two operators $\oplus$ and $\otimes$.

Proposition 5.8. *$(D_p, \oplus, \otimes, \{\emptyset\}, \emptyset)$ is an idempotent commutative semiring where $\{\emptyset\}$ is the additive identity and $\emptyset$ is the multiplicative identity.*

This is known as Martelli's semiring, originally used to compute cutsets of a graph [11, 12].

Proof. Straightforward, previously given by Martelli [11]. □

In order to transform the program during adornment we need a consistent reordering function. This can be constructed as a higher-order function from clauses to functions that map adornments to orderings. The inner function is constructed using the MOGs of the predicate as in Section 4.4.

$$reorderProgram_f(cl) \triangleq reordering_{cl,f}$$

5.1 Results

We now show that inter-clausal analysis (our full analysis) is fast-failing, terminating, sound, and complete.

Fast failure for ill-moded predicates is a strength of our analysis. Ill-moded constraints quickly propogate using the *step* function.

Proposition 5.9 (Fast failure). *After a single application of step, the ill constraint propagates from the body of a clause to the entire head predicate constraint.*

$$\forall f, cl, sub \in cl.\ f(pred(sub)) = \emptyset \implies step_{Pr}(f)(pred(head(cl))) = \emptyset$$

Corollary 5.10. *The number of step applications it takes to converge to the ill constraint is bounded by the static call distance between two predicates.*

Proof. Follows immediately from Proposition 5.9. □

Theorem 5.11. *For all Datalog programs, Pr, and mode functions, mv, inter-clausal analysis, $analyse_{Pr}([\![mv]\!])$, terminates.*

Proof. We know that *step* function terminates because intra-clausal analysis is a function of MOG construction which terminates by Lemma 4.9. All there is left to show is that *analyse* always reach a fixpoint. This is the case as *step* forms a chain (Lemma 5.6, Lemma 5.5) and $\preceq$ is bounded (Lemma 3.17). □

Theorem 5.12 (Inter-clausal soundness). *Given a program Pr containing a query cl_q with head predicate q and a mode function mv, if the analysis yields the trivial constraint for q, Pr is well-moded with respect to query clause cl_q and mode function mv.*

$$analyse_{Pr}([\![mv]\!]_F)(q) = \{\emptyset\}$$
$$\implies wellModedProgram(Pr, cl_q, [\![mv]\!]_F, reorderProgram_f(cl))$$

Proof. Let f and af be the constraint functions $[\![mv]\!]_F$ and $analyse_{Pr}([\![mv]\!]_F)$ respectively.

We know by (Lemma 5.7) $f \preceq af$, that is for any predicate p and adornment $\mathbf{a}$ being if $\mathbf{a}$ is consistent with $af(p)$, then it is also consistent with $f(p)$.

Assume the antecedent of the proposition. Recall that $\{\emptyset\}$ is the trivial constraint with which all binding patterns are consistent.

This means under all binding patterns, there is an ordering for the subgoals of cl_q, where the binding patterns derived for the subgoals are consistent with the constraints of af and by $\preceq$ also with f. Otherwise, we would contradict intra-clausal soundness (Lemma 4.19).

The predicate constraints of subgoals in the body of cl_q may arise from two sources. If the predicate in question is external, we know by assumption it is consistent with af and hence with f. If it is an internal predicate, then it is a combination of clause constraints via $\oplus$. We know by Lemma 5.3 that each of the clause constraints are consistent with the binding pattern given to the subgoal in cl_q. These clause constraints can only be generated by intra-clausal analysis. We can apply the same reasoning recursively to the body of these clauses to show that all constraints of the external predicates in the body are satisfied. Hence, all external predicate constraints according to f are satisfied as required.

The reason we need a fixpoint rather than a single application of *step* is that intra-clausal soundness ensures soundness with respect to the input constraint function but intra-clausal analysis potentially produces a more strict restrictive constraint function due to (mutual) recursion of clauses. At the fixpoint the output constraint function and the input constraint function are one and the same, hence all dataflow constraints are satisfied. □

Theorem 5.13 (Inter-clausal completeness).

$$\exists gr.\ wellModedProgram(Pr, cl_q, [\![mv]\!]_F, gr) \implies analyse_{Pr}([\![mv]\!]_F)(q) = \{\emptyset\}$$

Proof. We proceed with a proof-by-contradiction: we start by assuming that the antecedent is true and that $analyse_{Pr}([\![mv]\!]_F)(q) \neq \{\emptyset\}$. Since a query only has one clause, $\oplus$ is never invoked in $analyse_{Pr}$ for q and thus each *step* only ever extracts a constraint for the query from a single MOG. For the constraint to end up non-trivial, the MOG construction for cl_q must on all paths generate a non-trivial obligation in the accumulator (since a trivial obligation would dominate via the definition of $\otimes$). Therefore, we must need to bind a variable, say X, in the head of the query q. However, by the antecedent and the definition of *adornProgram*, there is an ordering gr such that all variables in the head of query can be adorned with f (free) and all subsequent clauses are consistent with $[\![mv]\!]_F$. Thus, we have reached a contradiction. Therefore, the statement of completeness holds.

□

Corollary 5.14 (Global reordering existence). *For every program that can be well-moded with reordering, we can construct a global reordering function.*

Proof. By Theorem 5.13, we know that if a global reordering function exists, analysis will find the trivial constraint for the head. This means for each clause as a part of the analysis we construct local reordering functions that are defined for all relevant binding patterns. By combining these local reordering functions, we can construct the desired global reordering function. □

Our analysis algorithm preserves the work that has been done on a program if it is extended by additional clauses.

Theorem 5.15 (Incremental analysis). *For a given program Pr, an arbitrary clause cl, and a mode function mv defined on all predicates appearing in Pr and cl, inter-clausal analysis can be incrementally computed by computing a constraint function for Pr first and using this as a basis for computing constraints for $Pr \cup \{cl\}$.*

$$analyse_{Pr\cup\{cl\}}([\![mv]\!]_F) = analyse_{Pr\cup\{cl\}}(analyse_{Pr}([\![mv]\!]_F))$$

Proof. By (Lemma 5.7), we have $[\![mv]\!]_F \preceq analyse_{Pr}([\![mv]\!]_F)$ We also have the following inequality:

$$analyse_{Pr}([\![mv]\!]_F) \preceq analyse_{Pr\cup\{cl\}}([\![mv]\!]_F)$$

This holds because if the head of cl is not a head in Pr, the constraint of the head of cl is $\{\emptyset\}$ which is the bottom. If it appears as a head, this means at each application of *step* there will be an additional constraint that needs to be combined using $\oplus$. We have $C_1 \preceq C_1 \oplus C_2$ by Lemma 5.3 and Definition 3.16.

By applying $analysis_{Pr\cup\{cl\}}$ to both inequalities we obtain:

$$\begin{aligned} &analyse_{Pr\cup\{cl\}}([\![mv]\!]_F) \\ \preceq\ &analyse_{Pr\cup\{cl\}}(analyse_{Pr}([\![mv]\!]_F)) \\ \preceq\ &analyse_{Pr\cup\{cl\}}(analyse_{Pr\cup\{cl\}}([\![mv]\!]_F)) \end{aligned}$$

Additionally, *analyse* reaches a fixpoint (Theorem 5.11), thus

$$\begin{aligned} &analyse_{Pr\cup\{cl\}}([\![mv]\!]_F) \\ \preceq\ &analyse_{Pr\cup\{cl\}}(analyse_{Pr}([\![mv]\!]_F)) \\ \preceq\ &analyse_{Pr\cup\{cl\}}([\![mv]\!]_F) \end{aligned}$$

Since $\preceq$ is anti-symmetric, the lemma holds. □

We achieve a stronger incremental computation result if the clause extending the program has a fresh head. It allows us to perform the analysis without performing intra-clausal analysis on the original program.

Corollary 5.16 (Fresh head incremental). *If additionally, we know that the head predicate of cl does not feature in Pr, we need not consider the clauses of Pr when extending the constraint function. That is:*

$$analyse_{Pr\cup\{cl\}}([\![mv]\!]_F) = analyse_{\{cl\}}(analyse_{Pr}([\![mv]\!]_F))$$

A stronger result still is achieved when the extending clause is non-recursive. Despite the restrictions on the clause, this captures queries in an interactive system. We can determine well-modedness of a query using a single application of the intra-clausal analysis.

Corollary 5.17 (Fast convergence). *If cl is also non-recursive, analyse converges to a fixpoint in a single step.*

$$analyse_{Pr\cup\{cl\}}([\![mv]\!]_F) = step_{\{cl\}}(analyse_{Pr}([\![mv]\!]_F))$$

6 EXTENDING DATALOG

The analysis so far accommodates Datalog programs with external predicates that are assumed to have no side-effects. However, most implementations extend Datalog in various ways. In this section, we explore how to accommodate extensions in our mode analysis. Namely, we discuss preserving order of effectful predicates, negated subgoals, and wildcards.

Additionally, we note that aggregate computations [8, Chapter 2] despite affecting flow of values are compatible with our analysis without any modifications.

6.1 Preserving order of effectful predicates

Since this work is motivated by incorporating external predicates, effectful predicate evaluation is a natural extension. This poses a problem for our analysis, since we reorder subgoals to achieve well-modedness. It is unsound for effectful predicates *e.g.* we may reorder the subgoals so that the subgoal that reads a file precedes the one that opens it.

We assume a coarse-grained effect system where each effectful predicate can interfere with any other, so the syntactic order of all effectful predicates need to be preserved. For each clause assume a list of the effectful subgoals, l, matching their syntactic order. We modify the rules for generating new edges in the MOG construction so that l appears in order in the edges.

For non-natural edges, if the subgoal selected for labelling is in source vertex list l, then it has to be the head of l otherwise we do not generate the edge. For natural edges, the set of subgoals in the label can contain at most one element from l and if it does it must be the head of l.

As this modification forces the edges to respect the syntactic order of effectful subgoals, we no longer produce unsound orderings.

6.2 Negated subgoals

Negation allows a subgoal to hold when it is not satisfied. There are various ways of accommodating it in Datalog with radical effects on language semantics but dataflow-wise they behave identically.

Example 6.1. If we evaluate the following program naïvely, the values for `outOfStock` will be the set of all finite strings except "Milk". This is undesirable as it compromises termination of bottom-up semantics.

```
1 inStock("Milk").
2 outOfStock(X) :- not(inStock(X)).
```

The usual solution to this problem is to require the variables inside a negated subgoal to be bound. This is easy to express within our analysis by changing the generation of alternatives at the root of the MOG. Say we have a negated subgoal n with predicate p. If the constraint function has $\emptyset$ for p hence it cannot be evaluated, then the alternative at the root node is $(n, \emptyset)$ as before. Otherwise, we require all the variables to be bound, so the alternative is $(n, \{vars(n)\})$.

6.3 Wildcards

Wildcards allow the value of a subgoal argument to be ignored. For example, `p(X,_)` ignores its second argument. This is equivalent to using an existential variable *i.e.* one that appears once in the body.

If a wildcard is used as an argument to a subgoal with mode + at that argument position, the dataflow requirements for that predicate cannot be satisfied. If the wildcards are eliminated through introduction of a fresh existential variable for each wildcard, no modification to our analysis is needed. However, if the analysis is performed without assigning fresh variables, we need two adjustments to the intra-clausal analysis. First, *vars* should ignore wildcards. Second, *cToOs* should add a special wildcard variable each time an atomic constraint indexes a wild card variable in the subgoal. Since *vars* ignores wildcards and head cannot have a wildcard, there will not be any way of scheduling these alternatives.

7 RELATED WORK

Mellish [13] introduces mode inference through abstract interpretation for Prolog programs. Debray and Warren [6] improves on this work by precise handling of aliasing. Both perform inference on the programs as they are written without reordering of subgoals.

Mercury [14] and HAL [5] mode systems are closest to ours in spirit. They both reorder subgoals to satisfy mode restrictions and both use constraint-based analysis. Both of these are higher-order languages and allow function symbols. Hence, they provide more sophisticated modes that express partial instantiations of variables *e.g.* a list with unbound variables is more instantiated than just a variable and less instantiated than a list with ground elements. Much of the analysis is thus concerned with precise aliasing tracking. By contrast, lack of function symbols simplify our mode analysis. In particular, Overton et al. [14] reports their constraint-based analysis is 10 to 100 times slower compared to their previous brute force search based algorithm on benchmark programs. Additionally, HAL only reorders subgoals during mode checking with mode specifications whereas we also do reordering in the absence of specifications. Another difference is that both of these are typed languages and their analyses rely on type of predicates for mode analysis. This is not possible for untyped Datalog.

More recently, Yedalog [4] and Dyna [7] were developed, inspired by Datalog. They both add function symbols and face the same aliasing problems described above. Both provide static mode systems and refer to Mercury as inspiration but without an explicit account of the underlying algorithm.

In addition to the order preservation method for subgoals with side-effects in Section 6.1, there is an alternative involving modes. Henderson et al. [9, Chapter 5] reify the external world as a value to be passed around. Use of mode constraints on external world arguments establishes mode dependencies between effectful clauses which would allow our analysis to remain sound without modification. This is similar to use of phantom types [3] in typed languages. The downside of this approach is that the external world has to be shuffled manually or a variable inserting transformation is needed.

Overall, we differ from the literature by targeting Datalog in its standard form without function symbols and types. This simplifies the analysis and allowed us to prove soundness and completeness. Additionally, unlike other approaches, our analysis is incremental allowing performant mode checking in interactive systems.

8 CONCLUSIONS

We presented a static mode analysis for Datalog to allow programs to be well-moded through reordering whenever possible. The combinatorial explosion of global permutation search is tackled by exploiting dataflow restrictions within the clauses and allowing incremental analysis of programs particularly for interactive systems. We showed that the algorithm is terminating, sound, and complete with respect to exhaustive global order search.

ACKNOWLEDGMENTS

We thank Alan Mycroft for his suggestions on the terminology and Tim Griffin for spotting errors in the earlier iterations of this paper. This work was supported by the EPSRC [grant number EP/M026124/1]

REFERENCES

[1] Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. Design and implementation of the LogicBlox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1371–1382. ACM, 2015.

[2] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In *Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 269–284. ACM, 1987.

[3] James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[4] Brian Chin, Daniel von Dincklage, Vuk Ercegovac, Peter Hawkins, Mark S Miller, Franz Och, Christopher Olston, and Fernando Pereira. Yedalog: Exploring knowledge at scale. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[5] M Garcıa de la Banda, Peter J Stuckey, Warwick Harvey, Kim Marriott, et al. Mode checking in HAL. *Lecture notes in computer science*, pages 1270–1284, 2000.

[6] Saumya K Debray and David S Warren. Automatic mode inference for logic programs. *The Journal of Logic Programming*, 5(3):207–229, 1988.

[7] Jason Eisner and Nathaniel W Filardo. Dyna: Extending datalog for modern AI. In *Datalog Reloaded*, pages 181–220. Springer, 2011.

[8] Todd J Green, Shan Shan Huang, Boon Thau Loo, Wenchao Zhou, et al. Datalog and Recursive Query Processing. *Foundations and Trends® in Databases*, 5(2): 105–195, 2013.

[9] Fergus J Henderson et al. Strong modes can change the world. In *Technical Report 93/25*. Citeseer, 1993.

[10] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: on synthesis of program analyzers. In *International Conference on Computer Aided Verification*, pages 422–430. Springer, 2016.

[11] Alberto Martelli. *An application of regular algebra to the enumeration of cut sets in a graph.* 1974.

[12] Alberto Martelli. A gaussian elimination algorithm for the enumeration of cut sets in a graph. *Journal of the ACM (JACM)*, 23(1):58–73, 1976.

[13] Christopher S Mellish. Abstract interpretation of Prolog programs. In *International Conference on Logic Programming*, pages 463–474. Springer, 1986.

[14] David Overton, Zoltan Somogyi, and Peter J Stuckey. Constraint-based mode analysis of Mercury. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 109–120. ACM, 2002.

[15] Yannis Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis. In *Datalog Reloaded*, pages 245–251. Springer, 2011.

[16] Zoltan Somogyi. A System of Precise Models for Logic Programs. In *ICLP*, pages 769–787. Citeseer, 1987.

[17] Zoltan Somogyi, Fergus J Henderson, and Thomas Charles Conway. Mercury, an efficient purely declarative logic programming language. *Australian Computer Science Communications*, 17:499–512, 1995.

A OMITTED PROOFS

Lemma 3.15 (Ill and trivial constraints). *The trivial constraint $\{\emptyset\}$ is consistent with all binding patterns and the ill constraint $\emptyset$ is consistent with none.*

Proof. The $\blacktriangleleft$ relation is universally quantified so it holds trivially for the atomic constraint $\emptyset$, hence *findAC* is never empty for the trivial constraint. On the other hand, for the ill constraint, *findAC* is always the empty set as there are no atomic constraints. □

Lemma 3.17. *For a fixed predicate p, $\leq$ is a bounded partial order (PO) with $\emptyset$ as the top element and $\{\emptyset\}$ as the bottom element. For a fixed domain $\leq$ is also a bounded partial order with constant functions $\emptyset$ and $\{\emptyset\}$ as top and bottom elements respectively.*

Proof. The fact that the relation $\leq$ is a partial order follows from properties of implication. Nothing is consistent with $\emptyset$ which confirms the top element and any adornment is consistent with $\emptyset$ which confirms the bottom element (Lemma 3.15). Pointwise extension preserves all of these properties. □

Lemma 3.18 (PO to atomic constraint relation). *If a constraint C_2 is more restrictive than C_1, this means C_1 has a more relaxed atomic constraint for each atomic constraint in C_2.*

$$\forall C_1, C_2.\ C_1 \leq C_2 \implies \forall AC \in C_2\ \exists AC' \in C_1.\ AC' \subseteq AC$$

Proof. Unfolding definitions of $\lhd$ and $\blacktriangleleft$, it is readily seen that the more restrictive constraint requires more arguments at particular locations to be *bound* than the less restrictive one, which establishes the subset relation. □

Lemma 4.8. *MOG construction leads to a scheduling graph.*

Proof. The first two properties of the valid scheduling property is satisfied trivially as new vertices use *nextAcc* and *nextAcc* as defined in property statements. The edge labels are selected using *pickLabel* which satisfies the third condition as it selects subgoals from the source vertex alternatives as well as the obligation it is coupled with.

Labels must not have obligation variables that do not appear in the head. When scheduling natural subgoals as they have empty obligations, that is the case. When scheduling non-natural ones, the condition is embedded inside *nonNats* selector used by *pickLabel*, hence that too is satisfied.

The root vertex is also chosen as it is defined in the scheduling graph.

All conditions of a scheduling graph are satisfied. □

Lemma 4.9 (MOG termination). *For any clause cl, and any constraint function f, the MOG construction $\mathrm{mog}_{cl,f}$ terminates.*

Proof. Apart from the root, the vertices are generated through edges. So it is sufficient to show that we can only generate finitely many edges. As a finite union of finite sets is finite, it suffices to show there is an n such that for all m bigger than to n, $\mathbf{E}(\mathrm{mog}^m_{cl,f})$ is empty.

This is indeed the case since the edges can only be added using *pickLabel* and *nextAlt* from a given vertex. The *pickLabel* function can only select labels from the alternatives of the preceding vertex and *nextAlt* removes the subgoals involved in edge labels from the succeeding alternative set. Hence, together they produce alternative sets strictly smaller than the edge's source vertex. Since we start with finite number of alternatives, we cannot generate infinite number of vertices. □

Lemma 4.17. *The extract function is monotonically increasing.*

$$\forall cl, f, g.\ f \preceq g \implies \mathit{extract}_{cl}(\mathrm{mog}_{cl,f}) \leq \mathit{extract}_{cl}(\mathrm{mog}_{cl,g})$$

Proof. Fix a clause cl and assume $f \preceq g$ for some f and g. This implies at each point p in these functions we have $f(p) \leq g(p)$. It is sufficient to show that every terminal path that can be reached via g is also reachable with f (in the sense that it has the same number of edges sharing the same subgoals on the edges but paired possibly with different obligations) and the accumulator at the end of each these paths is a subset of that of g.

By Lemma 3.18, we know for each predicate p, for all atomic constraints of $g(p)$, there is an atomic constraint in $f(p)$ that is smaller. This property trivially transfers to obligations in clause context. This means each time we can extend a vertex using g we can extend it with f and the obligation is at most as big as that of come from g. Hence, the accumulated obligation yields the desired subset property for each path.

There may be other terminal paths generated by f. If any of those leads to smaller accumulators, then the property still holds. If they do not, it does not matter because atomic constraints are minimised, so the redundant atomic constraints are removed. □

Lemma 4.18. *If any of the subgoals of a clause cl have the ill constraint according to constraint function f, the constraint for the clause is also ill.*

$$\forall sub \in body(cl).\ f(pred(sub)) = \emptyset \implies extract_{cl}(\mathrm{mog}_{cl,f}) = \emptyset$$

Proof. The *extract* function derives constraints using terminal vertices. However, $\mathrm{mog}_{cl,f}$ cannot have a terminal vertex because the alternatives contain a subgoal *sub* with empty set of obligations. This obligation is not $\{\emptyset\}$ so we cannot schedule *sub* using the first case of *pickLabel* for natural subgoals. We also cannot use the other case of *pickLabel* which uses *nonNats*, as the side condition for the new edge includes $o \in os$ and *os* in this case is empty. Since there is no way of scheduling *sub*, there is no way of reaching a terminal vertex. □

Lemma 4.21 (Well-moded ordering to terminal path). *For a constraint function f, clause cl, binding pattern* **a***, and ordering σ where wellModed$(cl, \mathbf{a}, f, \sigma)$, then a generated scheduling graph $g = schedule(cl, \mathbf{a}, f, \sigma)$ has a path $p \in paths(g)$ where terminal(p).*

Proof. For at least one path to be terminating, it follows that E_i must be non-empty for every $0 \leq i < n$. By the above construction and the *well scheduling* properties, this would then imply that $\{(\emptyset, Acc)\} \in E_n$ for some *Acc*, *i.e.* that *terminal*(p) for some path p.

Since the root node fills all the obligations for each subgoal from f, *i.e.*, $V_0 = \{(\{(s, \min cToOs_s(f(pred(s))))\ |\ s \in body(cl)\}$ then each singleton obligation $\{o\}$ for a subgoal s is derived from $f(pred(s))$.

From well-modedness, we have that $adornment(s) \lhd f(pred(s))$ for each subgoal $s \in body(adorn(cl, \mathbf{a}, \sigma))$, therefore, where we let $\mathbf{b} = adornment(s)$:

$$\begin{aligned}
& \mathbf{b} \lhd f(pred(s)) \\
\iff\ & \mathbf{b} \lhd osToC_s(cToOs_s(f(pred(s)))) \\
\implies\ & \mathbf{b} \lhd osToC_s(\{o\})\ \textit{where}\ \exists o \in cToOs_s(f(pred(s))) \\
\iff\ & \mathbf{b} \lhd osToC_s(\{o\})\ \textit{where}\ \exists o \in \min cToOs_s(f(pred(s)))
\end{aligned}$$

Therefore, from well-modedness, we can always satisfy the conditions of the E_i set comprehension for at least one pair (s, o). Thus, $E_i \neq \emptyset$ for all $0 \leq i < n$. Therefore, there is at least one terminal path $p \in schedule(cl, \mathbf{a}, f, \sigma)$. □

Lemma 4.24 (Trivial obligation consistency). *If a scheduling graph vertex v has a subgoal s with obligation $\{\emptyset\}$ in its alternative set, any ordering with a partial ordering derived from the root to v has the predicate s consistent with respect to any binding pattern.*

Proof. Due to the invariants of the scheduling graph, the only way v has s with the trivial obligation in its alternative set is if it had it that way at its root vertex or subgoals scheduled on p before v released variables in its obligation.

Since the variables in the obligation correspond to variables that needs to be bound to be consistent with the predicate constraint, any ordering that follows any one of the partial orderings up to v has s consistent with its constraint regardless the binding pattern of the clause head. □

Lemma 4.25 (Swap well-modedness). *For a path p well-moded with respect to a binding pattern* **a** *and some constraint function the swap operation preserves well-modedness.*

Proof. Let the sets of subgoals in the edges p_i and p_{i+1} be P and Q respectively. Consistency depends on the adornment of the subgoal which depends on the variables bound before adornment. We already fixed the head adornment **a** and that does not change with subgoal swapping.

We proceed by considering different portions of the path:

$q_{j<i}$ The set of bound variables are same as before, so the subgoal prior to this point remain consistent.

$q_{j>i+1}$ When regarded atomically, P and Q together bind the same set of variables regardless the order they are scheduled in. Hence, the bound variables after the vertex q_{i+1} remain the same.

q_{i+1} The set of variables there were sufficient to make subgoals in P remain bound when the subgoals in P are moved to the right. They might be augmented by the addition of variables in Q but additional bound variables do not compromise consistency.

q_i Because we have the side condition on swap that all subgoals of Q must have the trivial obligation at p_i, we already know all that needs to be bound to achieve consistency is bound at p_i (Lemma 4.24).

□

Lemma 4.26 (Swap scheduling path). *If a path p is in a scheduling graph before swapping, it remains to be in one after.*

Proof. Structurally, swap does not change the vertices or the edges.

The vertices conform with the properties of a scheduling path as they use *mkEdge* to construct the vertices. It is used in Definition 4.6 and Lemma 4.8 establishes that the vertices produced by it are those expected by a scheduling graph.

The edges also conform with scheduling graph requirements. The obligations within the labels being subset of the head variables is satisfied trivially as changing the position of the label has no effect on this.

The final requirement is that the elements of the labels (subgoal obligation pairs) have to be chosen from the alternatives. By assumption the label moved to the left, has the trivial obligations in the alternatives of the preceding vertex, so the requirement is satisfied. In the new intermediate vertex the alternatives may include shrank obligations due the variables bound the subgoals moved to the left. But the label is modified to exclude these variables from the obligations, so the property holds for this new label as well. □

Lemma 4.28 (Merge well-modedness). *If a path in a scheduling graph is well-moded with respect to some adornment and a constraint function, then the path produced by a merge is also well-moded.*

Proof. Let p be the path in question and p_i & p_{i+1} be the edges being merged. Fix the binding pattern **a** that the path is consistent with.

A merge only enables new orderings. We need to show each of these new orderings are still consistent with all of the predicate constraints when their binding patterns are derived from **a**.

In the new orderings, the subgoals before p_i appear in the positions they did in the old orderings, in which they were already consistent. The subgoals after p_{i+1} also appear in the same positions as before and changing the positions of the subgoals in p_i and p_{i+1} do not affect the variables bound during the adornment of these subgoals, hence they too remain consistent.

The new locations subgoals of p_i can appear in the new orderings are still after the points we established their consistency, hence their consistencies are preserved.

Those in p_{i+1} can appear at locations before the points we established their consistencies, but by assumption we only merge if those subgoals appear with trivial obligations inside the source of p_i. By Lemma 4.24, we know that these predicates also retain their consistencies. □

Lemma 4.29 (Merge scheduling graph). *If p is a path in a scheduling graph, a merge performed somewhere on this path produces another path the is also in a scheduling graph.*

Proof. We start with a path in the scheduling graph and make no modification to the vertices before and after the edges being merged. Hence, it is enough to show that the invariants are satisfied for the edges being merged.

Accumulator related invariants are trivially satisfied as the obligations on the labels of both of these edges are $\emptyset$ (due to third invariant of the valid scheduling property), hence they do not change the accumulator (as they did not before). They subset restriction on the label obligations are also satisfied, as $\emptyset$ is trivially a subset of the head variables.

The alternative set of the destination of the merged edge is also the same as before since exactly the same subgoals are removed from the alternative set and hence the same variables (of these subgoals) are released. □

Lemma 4.31 (Conversion preserves well-modedness). *If the path is well-moded, then so is the converted path.*

Proof. Swap and merge preserves well-modedness of a path. □

Lemma 4.32 (Greedy path completeness). *For a clause cl, every greedy scheduling path ending in a terminal vertex and conforming to a constraint function is present in the MOG determined by cl and the constraint function f.*

$$\forall p, \mathbf{a}.\ \mathit{greedy}(p) \wedge \mathit{wellModedPath}(p, \mathbf{a}, cl, f) \implies p \in \mathit{paths}(\mathrm{mog}_{cl,f})$$

Proof. We consider a more general property: that for a greedy, well-moded path p, a prefix of p of length n is a prefix path of $\mathrm{mog}_{cl,f}$, where an empty path comprises just the root vertex. We assume a path p satisfying the antecedent of the lemma, and proceed by induction on the length n of the prefix path:

- $n = 0$. By the definition of scheduling graphs, the root node V_0 is fixed, therefore $\{V_0\} = \mathbf{V}(\mathrm{mog}^0_{cl,f})$ trivially; a zero-length path comprises just the start vertex.
- $n = k + 1$. Let $(s, o) = \mathit{src}(p_{k+1})$ and assume the inductive hypothesis: the path $p_0 \ldots p_k$ is a prefix path of $\mathrm{mog}_{cl,f}$.
  We consider then two cases:
  - $(\mathit{nats}(\mathit{src}(p_{k+1})) \neq \emptyset)$ therefore by greediness and the well-scheduling property, $\mathit{trg}(p_{k+1}) = \mathit{mkVertex}(\{(\mathit{nats}(\mathit{Alt}), \emptyset)\})$ which is equal to the edge constructed by $\mathrm{mog}_{cl,f}$ give the vertex $\mathit{src}(p_{k+1})$;
  - $(\mathit{nats}(\mathit{src}(p_{k+1})) = \emptyset)$ therefore by well-modedness on the subgoal s we have that $\mathit{adornment}(s) \triangleleft f(\mathit{pred}(s))$. For the computed adornment to be consistent with the constraints of f, it follows that for every variable X in this clause which is bound in subgoals adornment, its corresponding index i in the constraint due to f. For X to be bound, it follows that it was bound earlier on the path, or is bound in the clause head. The former cannot be true, as if it was bound earlier on the path it would have been released from the alternative set via $\ominus$. Subsequently, it must be bound in the head and therefore its obligation $o \subseteq \mathit{vars}(\mathit{head}(cl))$. Therefore, by well-scheduling, we have a vertex which satisfies the requirements of *nonNats* in *pickLabel*, thus the edge p_{k+1} is equal to that constructed by $\mathbf{E}(\mathrm{mog}^k_{cl,f})$ at this point given vertex $\mathit{src}(p_{k+1})$.

  Therefore, $p_0 \ldots p_k p_{k+1}$ is a prefix path of $\mathrm{mog}_{cl,f}$.

Therefore $p \in \mathit{paths}(\mathrm{mog}_{cl,f})$. □

Lemma 4.33 (Path extract connection). *For a fixed binding pattern* **a**, *existence of a well-moded path in a MOG implies consistency of* **a** *with the constraint extracted from the MOG.*

$$\forall \mathbf{a}\ p.\ \mathit{wellModedPath}(p, \mathbf{a}, cl, f) \wedge p \in \mathit{paths}(\mathrm{mog}_{cl,f}) \implies \mathbf{a} \triangleleft \mathit{extract}_{cl,f}(\mathrm{mog}_{cl,f})$$

Proof. Fix a binding pattern **a** and a path p and assume the antecedent. To show the consequent, it is enough to show a stronger statement: the atomic constraint AC extracted from any well-moded MOG path p is consistent with the binding pattern **a**. This generalisation is valid since atomic constraints are combined via $\otimes$ in *extract* which is monotonically decreasing (getting less restrictive wrt. consistency).

Assuming an arbitrary index $i \in AC$ constraining variable X in the head, it follows that X is in the terminal accumulator and was scheduling at some point in the path p by extending the accumulator with this variable. Let s be the subgoal that causes this augmentation. Since we know by the premise that p is well-moded with respect to **a** and f, we also know the constraint of s in f is satisfied by the binding pattern at s derived from head binding pattern **a** by *adorn*. Since scheduling s augmented the accumulator, X could not have appeared in the previous subgoals as the release operator $\ominus$ would have eliminated X from the obligation of the alternative. Hence, X must be bound in the head. This is exactly what is required for the consistency with AC. As there is nothing particular about i and X, all indices in AC are similarly bound, hence the atomic constraint is consistent with **a**.

If there are no indices in the accumulator at the end of the path, the extracted constraint has to be trivial as $\otimes$ takes the minimal elements. Every binding pattern is consistent with the trivial constraint (Lemma 3.15), so the lemma holds. □

Lemma 5.3 (⊕ consistency homomorphism). *A binding pattern is consistent with two constraints combined with ⊕ iff that binding pattern is consistent with each constraint individually,* i.e.*:*

$$\forall \mathbf{a}, C_1, C_2.\ \mathbf{a} \lhd (C_1 \oplus C_2) \iff \mathbf{a} \lhd C_1 \wedge \mathbf{a} \lhd C_2$$

Proof. (⇒) Assume $\mathbf{a} \lhd (C_1 \oplus C_2)$. This means that for every element AC of $C_1 \oplus C_2$, we have $\mathbf{a} \blacktriangleleft AC$. Unfolding definition of $\oplus$, we have X and Y that are subsets of AC such that AC is $X \cup Y$. Unfolding definition of $\blacktriangleleft$, we have $\forall i \in AC.\ \mathbf{a}_i = \mathrm{b}$. This certainly holds for all subsets of AC, so we have $\forall i \in X.\ \mathbf{a}_i = \mathbf{b}$ and similarly for Y. C_1 must be a set of such X by definition of $\oplus$, thus $\mathbf{a} \lhd C_1$ holds. Same argument applies to C_2 as $\oplus$ is commutative.

(⇐) Assume $\mathbf{a} \lhd C_1$ and $\mathbf{a} \lhd C_2$. $C_1 \oplus C_2$ is a subset of $C_1 \times C_2$, so it suffices to show $\mathbf{a} \lhd C_1 \times C_2$. This requires showing $\mathbf{a}$ is consistent with every atomic constraint in this set. Since each of these atomic constraints can be represented as a union of an element from C_1 and another element from C_2 and that we know $\mathbf{a}$ is consistent with each of these elements, it also has to be consistent with the union. □

Lemma 5.5. *Every constraint function generated from a moding function gets more restrictive when step is applied to it.*

$$\forall mv.\ [\![mv]\!]_F \preceq step_{Pr}([\![mv]\!]_F)$$

Proof. Unfolding definition of $\preceq$, it is sufficient to show the equivalent pointwise property holds:

$$\forall mv, p.\ [\![mv]\!]_F(p) \preceq step_{Pr}([\![mv]\!]_F)(p)$$

By assumption mv only has mode requirements for external predicates. Let p be an arbitrary predicate within the domain of $[\![mv]\!]_F$. Now we consider effect of *step* depending on whether p is an internal or an external predicate.

If p is an external predicate, *step* does nothing, so the lemma holds by reflexivity of $\preceq$ (Lemma 3.17).

If p is an internal predicate, by assumption $[\![mv]\!]_F(p)$ is $\{\emptyset\}$, which is the bottom for $\preceq$ (Lemma 3.17). □

Proposition 5.9 (Fast failure). *After a single application of step, the ill constraint propagates from the body of a clause to the entire head predicate constraint.*

$$\forall f, cl, sub \in cl.\ f(pred(sub)) = \emptyset \implies step_{Pr}(f)(pred(head(cl))) = \emptyset$$

Proof. Fix f, cl, and sub assume the antecedent. By Lemma 4.18, we know the clause constraint has the ill constraint. By Lemma 5.3, we know any $\mathbf{a}$ that is consistent with a constraint combined using $\oplus$ with $\emptyset$ must have $\mathbf{a} \lhd \emptyset$. There is no such $\mathbf{a}$, thus the overall constraint for the predicate at the head of the clause is $\emptyset$ as required. □

Corollary 5.16 (Fresh head incremental). *If additionally, we know that the head predicate of cl does not feature in Pr, we need not consider the clauses of Pr when extending the constraint function. That is:*

$$analyse_{Pr \cup \{cl\}}([\![mv]\!]_F) = analyse_{\{cl\}}(analyse_{Pr}([\![mv]\!]_F))$$

Proof. Constraints of clauses are obtained by ⊕ operator and intra-clausal analysis which is a function of the clause body and the constraints for those subgoals. Clauses apart from cl are unaffected by the constraint of cl as the head predicate for this clause by assumption does not feature in other clauses, hence cannot affect the overall predicate constraints. Since the *step* function preserves constraints that are not mentioned in the set of clauses it is parametarised over, the lemma holds. □

Corollary 5.17 (Fast convergence). *If cl is also non-recursive, analyse converges to a fixpoint in a single step.*

$$analyse_{Pr \cup \{cl\}}([\![mv]\!]_F) = step_{\{cl\}}(analyse_{Pr}([\![mv]\!]_F))$$

Proof. We know the constraints of predicates appearing in Pr are all stable and body of cl can only feature them and external predicates for which the constraints do not change. Since the clause constraint is a function of its body and the constraints at this point, single iteration is sufficient. □