Mutual Exclusion by Interpolation

Jael Kriener and Andy King

University of Kent, Canterbury, CT2 7NF, UK

Abstract. The question of what constraints must hold for a predicate to behave as a (partial) function, is key to understanding the behaviour of a logic program. It has been shown how this question can be answered by combining backward analysis, a form of analysis that propagates determinacy requirements against the control flow, with a component for deriving so-called mutual exclusion conditions. The latter infers conditions sufficient to ensure that if one clause yields an answer then another cannot. This paper addresses the challenge of how to compute these conditions by showing that this problem can be reformulated as that of vertex enumeration. Whilst directly applicable in logic programming, the method might well also find application in reasoning about type classes.

1 Introduction

Given two sets of constraints, C_1 and C_2, defined over the same vector of variables $\boldsymbol{x}$, it is interesting to know exactly which variables of $\boldsymbol{x}$ are sufficient to distinguish between C_1 and C_2. That is, which elements of $\boldsymbol{x}$ need to be ground (fixed by being fully instantiated) so as to ensure that if C_1 is satisfiable then C_2 is unsatisfiable, and vice versa. We call a groundedness condition on a subset of $\boldsymbol{x}$ that has this property a mutual exclusion condition for C_1 and C_2.

These conditions have clear applications in logic programming [1, 2] and possible applications in functional programming, as is explained in Section 2. The problem is how to compute these conditions simply and do so in a way that is efficient. If C_1 and C_2 are systems of Herbrand constraints, such as the constraints $C_1 = (\mathsf{X} = [\mathsf{Y}|\mathsf{Ys}], \mathsf{Z} = [\mathsf{Y}|\mathsf{Zs}])$ and $C_2 = (\mathsf{X} = [], \mathsf{Z} = [])$, then the problem can be made more tractable by applying abstraction. In particular, observe that the linear constraints $S_1 = (\mathsf{X} = 1 + \mathsf{Ys}, \mathsf{Z} = 1 + \mathsf{Zs}, \mathsf{Ys} \geq 0, \mathsf{Zs} \geq 0)$ and $S_2 = (\mathsf{X} = 0, \mathsf{Z} = 0)$ describe the sizes of the terms that satisfy C_1 and C_2 respectively, where term size is quantified using the list-length norm [3, 4]. For instance, C_1 has as one its solutions $\mathsf{X} = [\mathsf{a}, \mathsf{b}, \mathsf{c}], \mathsf{Y} = \mathsf{a}, \mathsf{Ys} = [\mathsf{b}, \mathsf{c}], \mathsf{Z} = [\mathsf{a}], \mathsf{Zs} = []$ which is described by the size constraints $\mathsf{X} = 3, \mathsf{Ys} = 2, \mathsf{Z} = 1, \mathsf{Zs} = 0$ which, in turn, satisfies S_1. The value of abstraction is that a mutual exclusion condition for S_1 and S_2, for instance the groundness condition $\mathsf{X} \vee \mathsf{Z}$, is also a mutual exclusion condition for C_1 and C_2: if X is fixed and S_1 (resp. C_1) is satisfiable then S_2 (resp. C_2) is unsatisfiable and vice versa, and likewise for Z.

However, even whithin the simpler domain of linear constraints, it was not obvious how these conditions can be found except by repeated application of

existential quantifier elimination [1, 2], an operation that is fundamentally exponential [5]. In this paper we address this problem by providing an alternative and elegant way for computing mutual exclusion conditions over linear constraints by applying Craig interpolation [6], a topic that has attracted much interest recently in program verification [7]. In doing so, we make the following contributions:

- we explain how a variant of Craig interpolation for linear systems [8] can be generalised to the problem of inferring mutual exclusion conditions, thereby avoiding quantifier elimination entirely;
- we argue that through an application of Farkas' Lemma [9] the problem can be solved using lexicographical reverse search (*lrs*) [10], an algorithm that is renowned for it space and time efficiency [11];
- we show how the resulting algorithm is anytime, which is always desirable from an algorithmic perspective [12], that is, an algorithm that enumerates mutual exclusion conditions one by one, for instance X then Z or vice versa, the join of which constitutes the most general condition, in this case $\mathsf{X} \vee \mathsf{Z}$.
- we justify the approach with correctness arguments and experimental results.

The following presentation is structured as follows: Section 2 motivates the study with problems drawn from logic and functional programming. Section 3 outlines three techniques that are ultimately combined to provide a way of computing mutual exclusion conditions. Section 4 explains the combination and argues correctness. Section 5 presents an implementation and experimental data. Sections 6 reviews the wider context whilst Section 7 outlines directions for future work. Section 8 concludes.

Finally, a note on presentation: As mentioned above, our original motivation for this work stems from our interest in determinacy inference for logic programs. Therefore, we approach the central question from the logic programming point of view - where we use code, we will use Prolog syntax and conventions. We hope readers from other contexts will forgive this presentational bias.

2 Motivation

Mutual exclusion is of interest in both functional and logic programming.

2.1 Applications in the Functional Setting

In the functional setting, the issue of mutual exclusion between constraints over the same variables arises in the context of type classes and overlapping instances thereof. A simple example of this in Haskell is the `Show` class and following pair of instance declarations:

```
class Show t where show :: t -> String

instance Show a => Show [a] where ...
instance Show [Char] where ...
```

The first instance is a general instance of show for lists of as, the second is one specifically for strings. There is no way of guaranteeing that the `show` functions defined in these two instances behave equivalently for strings. Unless a deterministic choice between the two instances can be made, this overlap introduces a non-determinacy into a functional program which, of course, is highly undesirable. Peyton-Jones et al. [13] propose a framework that accepts some overlap, provided that a unique most specific instance can be found for each point of use. This allows, for instance, the example above, since at each point where the two instances of `Show` are both applicable, the second is more specific than the first. There are, however, cases of overlap which cannot be handled adequately by this requirement (see [14] for a more detailed discussion and example).

Currently the Hugs interpreter simply rejects overlapping instances. The Glasgow Haskell Compile (GHC), by default, does the same. Yet, GHC gives the user the option of allowing overlap, in which case it attempts to find the most specific applicable instance at all points of use (see [15, Section 7.2]). Even so, GHC only detects overlap when it poses a problem. Namely when a multiply overloaded method is called with a combination of arguments that does not allow the compiler to chose deterministically between the available instances.

In general, a type class C is defined by a declaration of the following form, where t_1 to t_n are type parameters, m is a method of type $t'_1 \to \ldots \to t'_i$ and each t'_j depends on zero or more of t_1 to t_n:

$$\mathit{class}\ \mathsf{C}\ t_1 \ldots t_n\ \mathit{where}\ \mathsf{m}\ ::\ t'_1 \to \ldots \to t'_i$$

The choice between two instances of C, I_1 and I_2, is determined by the constraints, c_1 and c_2, over $\langle t_1, \ldots, t_n \rangle$ imposed in the declarations of I_1 and I_2. These constraints can take one of the many forms like $t_k = [a]$, $t_k = (a, b)$, etc for some types a, b, and arise from partial instantiations (e.g. '`[a]`' in the example above) in the type declaration itself. Or they can be of the form $\mathsf{D}\ t_k$, for some type class D, and be propagated from a so-called context (e.g. '`Show a`' in the example above), prefixed to the type declaration of the instance by '`=>`'.

By finding all and only those parameters on which c_1 and c_2 are inconsistent, mutual exclusion inference could improve the situation in two ways:

- *Inference of safe use:* Suppose a method m is implemented twice in a pair of instances overlapping I_1 and I_2, I_1 and I_2 are mutually exclusive on a parameter t_k and the type of m depends on t_k. Then m can be used safely, if it is called in such a way that t_k is ground (i.e. m's use is not polymorphic at t_k). In this fashion, mutual exclusion inference can determine safe 'modes', i.e. patterns of polymorphism, of multiply instantiated methods.
- *Systematic detection:* Two instances I_1 and I_2 overlap on those parameters, that do not occur in the mutual exclusion condition between their respective constraints. If no other way of choosing between the two instances can be found (e.g. by finding a most specific one), the overlap is irreconcilable and the program potentially non-deterministic. Mutual exclusion inference can identify at least some overlap at the point of declaration, rather than the point of use of the instances, allowing for systematic detection of overlap and

providing appropriate guidance on how to avoid it. (To identify all overlap, the inference would have to be complete. See Section 7.)

2.2 Applications in the Logical Setting

In the logical setting, the issue arises in the context of determinacy or backward correctness [16, Section 1.7], i.e. whether a goal generates an answer once rather than multiply. An example is `subset`, which given two sets, represented as ordered, duplicate-free lists, checks whether the first is a subset of the second:

```
subset(    [],      _).
subset([X|L1], [X|L2]) :-        subset(L1, L2).
subset([X|L1], [Y|L2]) :- X > Y, subset([X|L1], L2).
```

It is highly desirable that `subset` behaves as a function when called with the both arguments ground - a Boolean test should not succeed more than once. On the other hand, when called with only the second argument ground, `subset` can be used to enumerate all subsets of a given set. To be utilisable thus, its behaviour needs to be relational, not functional, in this latter mode.

The problem of inferring functional modes for a predicate goes by the name of determinacy inference [1]. A form of determinacy inference for Prolog with cut has recently been presented [2] that applies backward analysis [17] to reduce the problem to the task of finding groundness conditions on a set of variables under which two systems of constraints are mutually exclusive, that is, not simultaneously satisfiable. However, the derivation of these conditions proved to be by far the most time consuming component of the whole implementation [2].

3 Theoretical Components

In this section we present the theoretical foundations for the mutual exclusion inference presented below. We introduce the three components of the inference mechanism - interpolation, Farkas' lemma and the *lrs*-algorithm - individually, and in section 4 establish the connections between these components.

3.1 Interpolation

Craig's interpolation theorem [6] has recently received considerable attention in verification. The reader is referred to [7] for a survey and discussion. In its original form the theorem states the following property of first-order logic (where $\mathsf{comps}(F)$ denotes the set of predicate- and constant-symbols in a formula F):

Theorem (Craig 1957). *Given two formulae A and B, such that $A \models B$, there is a formula I, such that $A \models I$ and $I \models B$ and $\mathsf{comps}(I) \subseteq \mathsf{comps}(A) \cap \mathsf{comps}(B)$. I is called an 'interpolant' of A and B.*

The interpolation theorem is frequently stated in the following form (see [18] for a discussion of the consequences of this change in formulation):

Theorem (Craig 1957 - Variation). *Given two formulae A and B, such that $A \wedge B \models \bot$ (their conjunct $A \wedge B$ is infeasible), there exists a formula I, such that $A \models I$ and $I \wedge B \models \bot$ and $\mathsf{comps}(I) \subseteq \mathsf{comps}(A) \cap \mathsf{comps}(B)$. Koács and Voronkov [18] introduce the term 'reverse interpolant' for this formula and we follow their terminology.*

Krajíček [8] translates this result to the theory of linear inequalities over non-negative variables thus (where $\mathsf{vars}(E)$ is the set of variables occurring in E and $\wedge\boldsymbol{A}$ is the conjunction of all elements of $\boldsymbol{A}$):

Theorem (Krajíček 1997). *Let $\boldsymbol{A} = \{A_1, \ldots, A_m\}$ and $\boldsymbol{B} = \{B_1, \ldots, B_\ell\}$ denote systems of linear inequalities of form $\sum_{i=1}^{n} a_i x_i \geq b$ where $a_i, b \in \mathbb{Z}$ and $x_i \geq 0$. If $(\wedge\boldsymbol{A}) \wedge (\wedge\boldsymbol{B}) \models \bot$, then there exists a reverse interpolant I such that $\wedge\boldsymbol{A} \models I$ and $I \wedge (\wedge\boldsymbol{B}) \models \bot$ where $\mathsf{vars}(I) \subseteq \mathsf{vars}(\boldsymbol{A}) \cap \mathsf{vars}(\boldsymbol{B})$.*

For the purposes of mutual exclusion between systems of linear constraints, we are interested in reverse interpolants of this form. To see this, observe that, given a reverse interpolant I between two clauses of constraints $\boldsymbol{A}$ and $\boldsymbol{B}$, the choice between $\boldsymbol{A}$ and $\boldsymbol{B}$ depends on the truth value of I: If I is satisfied, $\boldsymbol{B}$ cannot be. If it is not, neither is $\boldsymbol{A}$. Since grounding $\mathsf{vars}(I)$ is sufficient to fix I's truth value, it is sufficient to choose deterministically between $\boldsymbol{A}$ and $\boldsymbol{B}$. Hence, $\mathsf{vars}(I)$ defines a mutual exclusion condition betewen $\boldsymbol{A}$ and $\boldsymbol{B}$.

Note that in contrast to the original theorem, its variation, and therefore Krajíček's result, are completely symmetric and can be generalised to the following corollary.

Corollary 1 (Generalisation of Krajíček 1997). *Given n systems of linear inequalities $\boldsymbol{A}_1, \ldots, \boldsymbol{A}_n$, such that $\wedge_{i=1}^{n}(\wedge\boldsymbol{A_i}) \models \bot$, there is an n-tuple of reverse interpolants $\langle I_1, \ldots, I_n\rangle$, such that:*

(a) $(\wedge\boldsymbol{A}_k) \models I_k$ *for each* k,
(b) $\bigwedge_{i=1}^{k-1}(\wedge\boldsymbol{A}_i) \wedge I_k \wedge \bigwedge_{i=k+1}^{n}(\wedge\boldsymbol{A}_i) \models \bot$ *for each* k,
(c) $\mathsf{vars}(I_k) \subseteq \mathsf{vars}(A_k) \cap \left(\bigcup_{i=1}^{k-1} \mathsf{vars}(\boldsymbol{A}_i) \cup \bigcup_{i=k+1}^{n} \mathsf{vars}(\boldsymbol{A}_i)\right)$ *for each* k *and*
(d) $\bigwedge_{k=1}^{n} I_k \models \bot$.

We refer to $\langle I_1, \ldots, I_n\rangle$ as an n-tuple of 'generalised reverse interpolants' (GRIs) between the formulae $\boldsymbol{A}_1$ to $\boldsymbol{A}_n$.

Note that (c) is weaker than what may be expected. However, it is strong enough for our purposes, since it ensures that a generalised reverse interpolant will not constrain variables other than those shared between the system it is derived from and at least one other system in the original problem. In the context of constraints arising from Prolog predicates, that is sufficient to guarantee that the n-tuple of GRIs will contain constraints on the head-variables only. Observe also that this generalised problem subsumes the binary case: Given, for example, three systems $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$, such that $(\wedge\boldsymbol{A}) \wedge (\wedge\boldsymbol{B}) \models \bot$, there will be a triple of GRIs $(I_a, I_b, \top)$ between the three formulae where $\top$ denotes the

vacuous inequality. In fact, by enumerating all n-tuples of GRIs between a set of n formulae, one can decide which of them are pairwise inconsistent, by observing a tuple of GRIs for which exactly two interpolants are not $\top$.

For the present purpose of mutual exclusion inference, an enumeration of tuples of GRIs is therefore informative in the following two ways:

- Given a set of n clauses defining some predicate (or a set of instances of a type class), an enumeration of all n-tuples of GRIs decides which clauses are mutually exclusive (which instances potentially overlap).
- Furthermore, such n-tuples contain information sufficient to determine on which variables two clauses are mutually exclusive and thereby allow the inference of groundedness conditions that ensure that not more that one of the clause can ever succeed.

3.2 Farkas' Lemma

Farkas' Lemma [9] is a result in linear algebra which, since its first publication over a century ago, has been cited and used in a vast variety of equivalent formulations. One of these is the following (stated in [19]):

Theorem (Farkas 1907). *Let $\boldsymbol{M}$ be a matrix of dimensions $m \times n$ and $\boldsymbol{b}$ a column-vector of length m. **Exactly one** of the following statements holds:*

1. *There exists a column-vector $\boldsymbol{x}$ of length n, such that $\boldsymbol{Mx} \leq \boldsymbol{b}$.*
2. *There exists a row-vector $\boldsymbol{v} \geq \boldsymbol{0}$ of length m, such that $\boldsymbol{vM} = \boldsymbol{0}$ and $\boldsymbol{vb} < 0$, where $\boldsymbol{0}$ represents the null-matrix of appropriate dimensions.*

We shall use the following straightforward corollary of the above:

Corollary 2 (Corollary of Farkas 1907). *Let $\boldsymbol{M}$ be a matrix of dimensions $m \times n$ and $\boldsymbol{b}$ a column-vector of length m. If there is no column-vector $\boldsymbol{x}$ of length n, such that $\boldsymbol{Mx} \leq \boldsymbol{b}$, then there exists a column-vector $\boldsymbol{p} \geq \boldsymbol{0}$ of length m, such that $\boldsymbol{M}^T\boldsymbol{p} \geq \boldsymbol{0}$ and $\boldsymbol{b}^T\boldsymbol{p} < 0$, where $\boldsymbol{A}^T$ denotes the transpose of $\boldsymbol{A}$ and $\boldsymbol{0}$ represents the null-vector of appropriate dimensions.*

To put this into the present context, Farkas' Lemma states that every system of m linear inequality constraints over n variables has a dual system of $n+1$ linear inequalities over m variables such that either the original system has a feasible solution, $\boldsymbol{x}$, or the dual system has a feasible solution, $\boldsymbol{p}$, which exhibits some inconsistency in the original system. It does so by defining a way of combining the original system as a weighted sum which reduces to the inconsistent constant constraint $1 \leq 0$ or some simple equivalent thereof.

Example 1 (A Farkas' Witness for Inconsistency). Consider the following unfeasible system of four linear inequalities: $x \leq 0$, $y + z \geq 1$, $2x \geq 1$, $z \leq 1$, represented by the matrices $\boldsymbol{M}$ and $\boldsymbol{b}$ below. The column-vector $\boldsymbol{p}$ ($\geq \boldsymbol{0}$) is a solution to the dual system $\boldsymbol{M}^T\boldsymbol{p} \geq \boldsymbol{0} \wedge \boldsymbol{b}^T\boldsymbol{p} < 0$ defined by Corollary 2:

$$\boldsymbol{M} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & -1 \\ -2 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \boldsymbol{b} = \begin{bmatrix} 0 \\ -1 \\ -1 \\ 1 \end{bmatrix} \qquad \boldsymbol{p} = \begin{bmatrix} 2 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

When interpreting $\boldsymbol{p}$'s elements as weights on the original inequalities, their weighted sum becomes $(2x - 2x) \leq (2 \times 0 - 1)$ or equivalently $0 \leq -1$.

The relevance for mutual exclusion now becomes apparent. Given a system that is known to be unfeasible, Farkas' Lemma provides a witness for that fact, $\boldsymbol{p}$, which extracts an unfeasible sub-system by means of discarding irrelevant constraints, which are simply weighted by zero.

Moreover, the reader will have observed the elegance in the fact that the problem of finding a suitable $\boldsymbol{p}$ is formulated in the same framework as the original problem it is derived from. Identifying an inconsistent core of a linear system requires no new machinery or meta-reasoning over and above what is in place to try and solve the original system in the first place.

3.3 The *lrs*-algorithm

The final component is the *lrs*-algorithm, developed and implemented by Avis [10]. To establish the necessary connections, we refer to some concepts from the theory of linear optimisation [20], in particular polyhedra [19].

The Main Theorem for polyhedra [19, Theorem 1.2] states that any polyhedron in n-dimensional space has two equivalent representations: one as a system of linear inequalities over n variables (the H-representation), the other as a set of vertices and rays in n-dimensional space (the V-representation). Each vertex in the V-representation defines a so-called basic feasible solution of the H-representation. To explain, let $\boldsymbol{A}$ denote a system of linear inequalities of the form $\boldsymbol{c} \cdot \boldsymbol{x} \leq b$ over n variables $\boldsymbol{x}$, where $\boldsymbol{c}$ is a vector of coefficients. A basic feasible solution of $\boldsymbol{A}$ is a vector $\boldsymbol{x}$ that satisfies all inequalities in $\boldsymbol{A}$ and, in addition, $\boldsymbol{x}$ saturates n linearly independent inequalities in $\boldsymbol{A}$. (A vector $\boldsymbol{x}$ is said to saturate an inequality $\boldsymbol{c} \cdot \boldsymbol{x} \leq b$ if $\boldsymbol{c} \cdot \boldsymbol{x} = b$.) Any solution to $\boldsymbol{A}$ can be derived from as a linear combination of vertices, or basic feasible solutions, in the V-representation of polyhedron.

The main function of the *lrs*-algorithm is vertex enumeration, that is, compute the V-representation of a polyhedron from its H-representation (though conversion in the other direction is also possible). As we have seen, vertex enumeration is equivalent to enumerating the basic feasible solutions of a system of linear inequalities. The *lrs*-algorithm is based on the reverse search algorithm presented by Avis and Fukuda [21]. Again, space-constraints prohibit a detailed discussion of its functionality, except to say that the method is based on repeated pivoting, that is, moving from one basic feasible solution to another by replacing one of the saturated inequalities with another [10]. Among various correctness results, Avis [10] states the following performance result:

Theorem (Avis 2000). *The lrs-algorithm finds all vertices of a polyhedron defined by m linear inequalities over d variables in time $O(md^2)$ per vertex and $O(md)$ space.*

Note that, crucially, the time required for the computation depends on the output, that is the number of vertices in the V-representation. Moreover, lrs does

not require any auxiliary data-structures to store a network of vertices, or even require a stack to realise recursion, which is reflected in its low space complexity. This motivates reformulating the problem of inferring mutual exclusion conditions as a vertex enumeration problem.

4 Inferring Interpolants for Mutual Exclusion

This section uses two running examples to illustrate the inference of mutual exclusion condictions: One very accessible (A), the other designed to demonstrate more interesting features of the method (B). Correctness is also addressed.

The method for inferring tuples of GRIs presented below is the direct application of the following two observations:

Observation 1 (Farkas' Lemma is an Interpolation Result). Corollary 2 of Farkas' lemma has almost the form of an interpolation result. The difference lies in the fact that, given n systems $\{\boldsymbol{A}_1, \ldots, \boldsymbol{A}_n\} = \boldsymbol{A}$ such that $\bigwedge_{i=1}^{n}(\wedge \boldsymbol{A}_i)$ is inconsistent, the interpolation results stated in Corollary 1, stipulate the existence of a reverse interpolant for each subsystem $\boldsymbol{A}_k$, which encapsulates the inconsistency between that $\boldsymbol{A}_k$ and $\boldsymbol{A} \setminus \boldsymbol{A}_i$. Whereas Corollary 2 proves the existence of a single vector $\boldsymbol{p}$ containing information about the entire problem. The crucial insight is that this $\boldsymbol{p}$ can be treated as fully compositional and be divided into subvectors $\boldsymbol{p}_k$ for each original $\boldsymbol{A}_k$. Calculating the weighted sums defined by these subvectors results in a tuple of GRIs. That is seen by noting that any weighted sum over a subsystem $\boldsymbol{A}_k$ is in fact entailed by $\boldsymbol{A}_k$. Moreover, since addition is associative, these entailed inequalities are inconsistent, that is to say, their sum is equivalent to $1 \leq 0$, provided the original weighted sum is. Every Farkas witness for inconsistency $\boldsymbol{p}$, as defined by Corollary 1, therefore corresponds to an n-tuple of GRIs between the subsystems of $\boldsymbol{A}$.

Observation 2 (lrs Efficiently Computes Farkas-interpolants). A second difference between the interpolation results in section 3.1 and Farkas' lemma is the fact that the latter is constructive as opposed to the former. That is to say Corollary 1 asserts the existence of *some* tuple of GRIs between inconsistent systems of linear inequalities, but does not prescribe how to derive these tuples. Corollary 2, on the other hand, provides sufficient information about the vector $\boldsymbol{p}$ to allow its construction as any solution of a given system of linear inequalities. The *lrs*-algorithm is designed to find *all* basic feasible solutions of such a system and therefore enumerate the $\boldsymbol{p}$ vectors.

Combining these two observations, we obtain a way to efficiently compute tuples of GRIs between inconsistent systems of linear inequalities. The inference proceeds in the following four simple stages: (a) Normalisation, (b) Applying Farkas' Lemma, (c) Applying *lrs* and (d) Interpreting the result. To demonstrate, consider the following two examples:

Example 2. Below are argument size relationships over non-negative $\langle x, y, z\rangle$ that can be derived [22] to describe following two predicates append and multiplex:

```
app( [], L, L).
app( [H|L1], L2, [H|L]) :-
    app( L1, L2, L).
```

```
multiplex( [], _, []).
multiplex( [A], [B], [A,B,A]).
multiplex( [A], [B1,B2], [A,B1,A,B2]).
multiplex( [A|As], B, C) :-
    append( [A|As], [A|As], Anew),
    zip( Anew, Bnew, C).
```

$\mathsf{append}(x, y, z)$:- $x = 0, y = z.$
$\mathsf{append}(x, y, z)$:- $x \geq 1, y \leq z - 1.$

$\mathsf{multiplex}(x, y, z)$:- $x = 0, z = 0.$
$\mathsf{multiplex}(x, y, z)$:- $x = 1, y = 1, z = 3.$
$\mathsf{multiplex}(x, y, z)$:- $x = 1, y = 2, z = 4.$
$\mathsf{multiplex}(x, y, z)$:- $x \geq 2, z = 2x + y.$

Normalisation To apply linear programming, the size relations are normalised by substituting strict equalities $x = y$ with two inequalities $x - y \leq 0$ and $y - x \leq 0$ and transforming inequalities into the form $a_1x_1 + \ldots + a_nx_n \leq a_0$:

Example 3. The size relationships that hold for the i^{th} clause of append can be represented by the system $\boldsymbol{A}_i = M_{A_i}\boldsymbol{x} \leq b_{A_i}$ where $\boldsymbol{x} = \langle x, y, z\rangle$. Likewise $\boldsymbol{B}_i = M_{B_i}\boldsymbol{x} \leq b_{B_i}$ characterises the i^{th} clause of multiplex.

$$M_{A_1} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & -1 & 1 \end{bmatrix} \quad b_{A_1} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad M_{A_2} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix} \quad b_{A_2} = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$$

$$M_{B_1} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{bmatrix} \quad b_{B_1} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad M_{B_2} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{bmatrix} \quad b_{B_2} = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \\ 3 \\ -3 \end{bmatrix}$$

$$M_{B_3} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{bmatrix} \quad b_{B_3} = \begin{bmatrix} 1 \\ -1 \\ 2 \\ -2 \\ 4 \\ -4 \end{bmatrix} \qquad M_{B_4} = \begin{bmatrix} -1 & 0 & 0 \\ -2 & -1 & 1 \\ 2 & 1 & -1 \end{bmatrix} \quad b_{B_4} = \begin{bmatrix} -2 \\ 0 \\ 0 \end{bmatrix}$$

Next we combine the matrices M_{A_i} (resp. M_{B_i}) and vectors b_{A_i} (resp. b_{B_i}) thus:

Example 4.

$$M_A = \begin{bmatrix} \frac{M_{A_1}}{M_{A_2}} \end{bmatrix} \quad b_A = \begin{bmatrix} \frac{b_{A_1}}{b_{A_2}} \end{bmatrix} \qquad M_B = \begin{bmatrix} \frac{M_{B_1}}{M_{B_2}} \\ \frac{M_{B_3}}{M_{B_4}} \end{bmatrix} \quad b_B = \begin{bmatrix} \frac{b_{B_1}}{b_{B_2}} \\ \frac{b_{B_3}}{b_{B_4}} \end{bmatrix}$$

Applying Farkas' Lemma Applying the Corrollary of Farkas' Lemma, we know that each of these systems is infeasible, iff there is a corresponding column-vector $\boldsymbol{p} \geq 0$, such that: $M^T\boldsymbol{p} \geq \mathbf{0}$ and $b^T\boldsymbol{p} < 0$. We are looking, therefore, for column-vectors $\boldsymbol{p}$, which satisfies the following constraints:

Example 5.

$$\begin{bmatrix} 1 & -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & -1 & 0 & 1 \\ 0 & 0 & -1 & 1 & 0 & -1 \end{bmatrix} \boldsymbol{p_A} \geq \mathbf{0} \quad \wedge \quad \begin{bmatrix} 0 & 0 & 0 & 0 & -1 & -1 \end{bmatrix} \boldsymbol{p_A} < 0$$

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & -1 & -2 & 2 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 \end{bmatrix} \boldsymbol{p_B} \geq \mathbf{0} \quad \wedge$$
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 3 & -3 & 1 & -1 & 2 & -2 & 4 & -4 & -2 & 0 & 0 \end{bmatrix} \boldsymbol{p_B} < 0$$

Applying lrs These systems of constraints define vertex enumeration problems. Running *lrs* on these problems, we obtain exactly two possible solutions in example A and 35 in example B, 12 of which are shown below:

Example 6.

$$\begin{aligned} \boldsymbol{p}_{A_1} &= \langle 1, 0, 0, 0 \,|\, 1, 0 \rangle \\ \boldsymbol{p}_{A_2} &= \langle 0, 0, 0, 1 \,|\, 0, 1 \rangle \end{aligned}$$

$$\begin{aligned} \boldsymbol{p}_{B_1} &= \langle 0,\ 0, 0, 0, \,|\, 0, 0, 0, 0, 0, 0, \,|\, 0, 0, 0, \tfrac{1}{2}, \tfrac{1}{2}, 0, \,|\, 1,\ 0, \tfrac{1}{2} \rangle \\ \boldsymbol{p}_{B_2} &= \langle 0,\ 0, 0, 0, \,|\, 0, 0, 0, 0, 0, 0, \,|\, 1, 0, 0, 0, 0, 0, \,|\, 1,\ 0, 0 \rangle \\ \boldsymbol{p}_{B_5} &= \langle 0,\ 0, 0, 0, \,|\, 0, 0, 0, 0, 1, 0, \,|\, 0, 0, 0, 0, 0, 0, \,|\, 2,\ 0, 1 \rangle \\ \boldsymbol{p}_{B_6} &= \langle 0,\ 0, 0, 0, \,|\, 1, 0, 0, 0, 0, 0, \,|\, 0, 0, 0, 0, 0, 0, \,|\, 1,\ 0, 0 \rangle \\ \boldsymbol{p}_{B_{11}} &= \langle 0,\ 0, 0, 0, \,|\, 0, 0, 1, 0, 0, 0, \,|\, 0, 0, 0, 1, 0, 0, \,|\, 0,\ 0, 0 \rangle \\ \boldsymbol{p}_{B_{12}} &= \langle 0,\ 0, 0, 0, \,|\, 0, 0, 0, 0, 1, 0, \,|\, 0, 0, 0, 0, 0, 1, \,|\, 0,\ 0, 0 \rangle \\ \boldsymbol{p}_{B_{16}} &= \langle 0,\ 0, \tfrac{1}{4}, 0, \,|\, 0, 0, 0, 0, 0, 0, \,|\, 0, 0, 0, 0, 0, 0, \,|\, \tfrac{1}{2}, 0, \tfrac{1}{4} \rangle \\ \boldsymbol{p}_{B_{29}} &= \langle \tfrac{1}{2}, 0, 0, 0, \,|\, 0, 0, 0, 0, 0, 0, \,|\, 0, 0, 0, 0, 0, 0, \,|\, \tfrac{1}{2}, 0, 0 \rangle \\ \boldsymbol{p}_{B_{19}} &= \langle 0,\ 0, \tfrac{1}{4}, 0, \,|\, 0, 0, 0, 0, 0, 0, \,|\, 0, 0, 0, 0, 0, \tfrac{1}{4}, \,|\, 0,\ 0, 0 \rangle \\ \boldsymbol{p}_{B_{32}} &= \langle 1,\ 0, 0, 0, \,|\, 0, 0, 0, 0, 0, 0, \,|\, 0, 1, 0, 0, 0, 0, \,|\, 0,\ 0, 0 \rangle \\ \boldsymbol{p}_{B_{21}} &= \langle 0,\ 0, \tfrac{1}{3}, 0, \,|\, 0, 0, 0, 0, 0, \tfrac{1}{3}, \,|\, 0, 0, 0, 0, 0, 0, \,|\, 0,\ 0, 0 \rangle \\ \boldsymbol{p}_{B_{35}} &= \langle 1,\ 0, 0, 0, \,|\, 0, 1, 0, 0, 0, 0, \,|\, 0, 0, 0, 0, 0, 0, \,|\, 0,\ 0, 0 \rangle \end{aligned}$$

Interpreting the Result Observe that, unsurprisingly, each of these has exactly one row for every constraint in the respective normalised system. Interpreting the elements of each $\boldsymbol{p}$ as multipliers and calculating the thus weighted sum of the original constraints, we obtain from each $\boldsymbol{p}$ an n-tuple of GRIs:

Example 7.

$$\begin{aligned} I_{A_1} &= \langle \quad x \leq 0, \quad -x \leq -1 \rangle \\ I_{A_2} &= \langle\, z - y \leq 0, \, y - z \leq -1 \rangle \end{aligned}$$

$$
\begin{array}{rl}
I_{B_1} = \langle & \top \ , \ \top \ , \ \frac{z}{2} - \frac{y}{2} \leq 1 \ , \ \frac{z}{2} - \frac{y}{2} \geq 2 \ \rangle \\
I_{B_2} = \langle & \top \ , \ \top \ , \ x \leq 1 \ , \ x \geq 2 \ \rangle \\
I_{B_5} = \langle & \top \ , \ z \leq 3 \ , \ \top \ , \ z - y \geq 4 \ \rangle \\
I_{B_6} = \langle & \top \ , \ x \leq 1 \ , \ \top \ , \ x \geq 2 \ \rangle \\
I_{B_{11}} = \langle & \top \ , \ y \leq 1 \ , \ y \geq 2 \ , \ \top \ \rangle \\
I_{B_{12}} = \langle & \top \ , \ z \leq 3 \ , \ z \geq 4 \ , \ \top \ \rangle \\
I_{B_{16}} = \langle & \frac{z}{4} \leq 0 \ , \ \top \ , \ \top \ , \ \frac{z}{4} \geq 1 \ \rangle \\
I_{B_{29}} = \langle & \frac{x}{2} \leq 0 \ , \ \top \ , \ \top \ , \ \frac{x}{2} \geq 1 \ \rangle \\
I_{B_{19}} = \langle & \frac{z}{4} \leq 0 \ , \ \top \ , \ \frac{z}{4} \geq 1 \ , \ \top \ \rangle \\
I_{B_{32}} = \langle & x \leq 0 \ , \ \top \ , \ x \geq 1 \ , \ \top \ \rangle \\
I_{B_{21}} = \langle & \frac{z}{3} \leq 0 \ , \ \frac{z}{3} \geq 1 \ , \ \top \ , \ \top \ \rangle \\
I_{B_{35}} = \langle & x \leq 0 \ , \ x \geq 1 \ , \ \top \ , \ \top \ \rangle
\end{array}
$$

The reader will have observed, that these are in fact n-tuples of GRIs between the clauses of the original systems.

Example 8. Consider $I_{B_{11}}$ and $I_{B_{12}}$. Since each has $\top$ in the first and fourth position, they each show that $\boldsymbol{B}_2$ and $\boldsymbol{B}_3$ are pairwise inconsistent. The former exhibits their inconsistency on y, the latter that on z. Hence the groundness condition $y \vee z$ is sufficient for clauses 2 and 3 of multiplex to be mutually exclusive (if one clause is satisfiable then the other is not). Likewise the groundness condition $x \vee z$ is sufficient to distinguish between clause pairs 1 and 2 (indicated by $I_{B_{21}}$ and $I_{B_{35}}$), 1 and 3 (by $I_{B_{19}}$ and $I_{B_{32}}$), 1 and 4 (by $I_{B_{16}}$ and $I_{B_{29}}$), and lastly 2 and 4 (by I_{B_5} and I_{B_6})[1]. Finally for clause pair 3 and 4 the condition is $x \vee (y \wedge z)$ (by I_{B_1} and I_{B_2}). Thus an overall condition sufficient to ensure that not more than one clause can ever succeed is $(y \vee z) \wedge (x \vee z) \wedge (x \vee (y \wedge z)) = (y \wedge z) \vee (x \wedge (y \vee z))$.

By similar reasoning, though on a smaller scale, a condition of $x \vee (y \wedge z)$ is inferred for append from I_{A_1} and I_{A_2}.

4.1 Correctness

Lemma 1 (Soundness). *Let $\boldsymbol{A}$ be an inconsistent system of linear inequalities composed of n subsystems. If the above method finds an n-tuple $\boldsymbol{I}$ of entailed constraints from $\boldsymbol{A}$, then $\boldsymbol{I}$ is indeed an n-tuple of GRIs between $\boldsymbol{A}$.*

Proof. By Observation 1 and soundness of *lrs* (Theorem - Avis 2010).

Lemma 2 (Completeness). *Let $\boldsymbol{A}$ be a system of linear inequalities, composed of n subsystems. If $\boldsymbol{A}$ is inconsistent, i.e. there is an n-tuple of GRIs between the subsystems of $\boldsymbol{A}$, then the above method finds one.*

Proof. By Corollary 1 and completeness of *lrs* (Theorem - Avis 2010).

[1] Since all values are required to be non-negative, the constraint $z - y \geq 4$ implies $z \geq 4$; when derriving a groundedness condition from a tuple where the GRIs do not share all variables, the intersection generally suffices.

Minimality Given a *set* of tuples of GRIs, $\boldsymbol{I}$, this construction renders a necessary, not just sufficient, mutual exclusion condition between two subsystems $\boldsymbol{A}_i$ and $\boldsymbol{A}_j$ of a system of linear inequalities $\boldsymbol{A}$, if the following holds. To express the condition, let $\boldsymbol{I} \setminus (i,j)$ denote the subset of $\boldsymbol{I}$ containing all those tuples which have $\top$ in all positions other than the ith and jth. Then the condition is that $\boldsymbol{I} \setminus (i,j)$ elements cannot be linearly combined into a tuple I_x, such that (a) $I_x \notin \boldsymbol{I} \setminus (i,j)$, and (b) $\nexists I_k \in \boldsymbol{I} \setminus (i,j)$ such that $\mathsf{vars}(I_k) \subset \mathsf{vars}(I_x)$.

To see this, recall that all non-basic solutions of a system of linear inequalities can be derived by linear combination of basic feasible solutions. Further, though each $I_k \in \boldsymbol{I}$ defines a disjunct in the condition, I_k defines a redundant disjunct, if there is another tuple I_l, such that $\mathsf{vars}(I_l) \subseteq \mathsf{vars}(I_k)$. Given, therefore, a condition derived in the manner described above on the basis of a set of tuples of GRIs, $\boldsymbol{I}$, it can only be weakened by an additional non-redundant disjunct, if there is a tuple of GRIs which fulfils *(a)* and *(b)*.

Even though the sets of n-tuples of GRIs inferred by the above method generally possess that property, there are exceptions to that rule. They arise in situations where two clauses each contain more than one constraints on a common variable and that variable occurs with both a negative and a positive coefficient in each clause. If that variable occurs in more than one n-tuple of GRIs, then linear combination of these tuples may result in the cancellation, and therefore disappearance of that variable.

Example 9. To demonstrate, consider the following two systems:

$$\begin{array}{rr} \boldsymbol{A}_1 & \boldsymbol{A}_2 \\ x - z \leq 0 & -x + z \leq -1 \\ -x + y \leq 0 & x - y \leq -1 \end{array}$$

Each system contains more than one constraint on x and in each system, x occurs with both a positive and a negative coefficient.

Our method constructs two pairs of GRIs, $\langle x - z \leq 0, -x + z \leq -1 \rangle$ and $\langle -x + y \leq 0, x - y \leq -1 \rangle$, and thus infers $(x \wedge y) \vee (x \wedge z)$ as a mutual exclusion condition between $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$. However, it does not find $\langle y - z \leq 0, -y + z \leq -2 \rangle$, which is a linear combination of the first two. If this third pair were constructed, the inferred condition would be weakened to $(x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$.

As it stands, our method therefore cannot guarantee minimality of the mutual exclusion conditions inferred. In the (uncommon) case described above, the conditions inferred are stronger than necessary. We discuss a possible solution to that problem in section 6.

5 Implementation and Experimental Results

In our previous implementation of a determinacy inference [2], the time required to infer the mutual exclusion conditions dominated all other aspects of the analysis. We thus compare how the new approach compares against the time required

for argument size analysed based on polyhedra [22]. To make the comparison as fair as possible, the argument size analysis uses Octagons [23], which is a template domain that is particularly efficient, rather than general polyhedra [22].

file	size	lfp (ms)	mux (ms)	file	size	lfp (ms)	mux (ms)
browse	27	620	0	qsort	8	620	0
crypt_wamcc	20	670	24	queens	11	110	0
dialog	35	640	0	ronp	34	650	0
disj_r	59	1010	0	rotate	3	30	0
fastcolor	8	60	0	semi	25	3040	0
gabriel	31	650	0	serialize	17	930	4
ime_v2-2-1	44	19910	3	shape	10	90	0
music	11	290	0	treeorder	13	610	1

The column *size* is the number of predicates in the benchmark; *lfp* is the time required to compute the argument size relations; and *mux* is the time required to compute the mutual exclusion conditions from the argument sizes. Timings were performed on 2.53 GHz Macbook Pro. The analyser was implemented in SICStus 4.0.1 using its linear constraints for realising Octagons and lrs (`http://cgm.cs.mcgill.ca/~avis/C/lrs.html`) for computing the mutual exclusion conditions. We conclude that *mux* is no longer the computational bottleneck.

6 Related Work

Overlapping Type Class Instances Morris *et al* [14] discuss the problem of overlapping between instances of the same type class in Haskell. A similar problem arises in Coq [24]. We are not aware of any existing approaches to this problem that are oriented towards static analysis, but we believe that mutual exclusion inference may provide the right basis for such an approach.

Determinacy Checking The earliest framework for determinacy analysis of logic programs we are aware of is Sahlin's work [25] which geared towards partial evaluation and detects whether there are none, one or more than one way in which a goal can succeed. This approach is further developed by Mogensen [26], who presents a formal semantics to exhibits its correctness. Dawson et al. [27] present a bottom-up framework for adding success patterns to predicates so as to improve determinacy behaviour and promote early failure in unsuccessful computations. More recently, López-García et al. [28] present a framework which detects both fully deterministic predicates and predicates with mutually exclusive clause tests. Their work is sufficiently general to be able to handle constraints from different domains, including both linear and Herbrand constraints, and as well as address the effects of the *cut* on the determinacy of a predicate.

Interpolation Interpolation has recently found application in model checking [29], but the majority of approaches of constructing interpolants are based on proof-analysis [7]. Rybalchenko and Sofronie-Stokkermans [30] discuss the limitations

of deriving interpolants from proofs and show how to reduce the problem of constructing interpolants in linear arithmetic extended by uninterpreted function symbols to, essentially, three constraint problems. Their reduction makes explicit use of Farkas' Lemma. On the one hand their approach is more general than the work presented here, since constraint solving is applicable in a variety of domains other than linear arithmetic. On the other hand, this work represent a generalisation of their approach, since it allows the simultaneous enumeration of several interpolants between two or more such systems of linear inequalities.

7 Future Work

Our method will not always find the weakest mutual exclusion condition. As example 9 illustrates, this arises when two n-tuples of GRIs are combined in such a way that a common variable is removed in the combination process. These combinations can be found by adding a single constraint to the lrs formulation that stipulates that the coefficient of the shared variable is zero. We believe that this refinement ensures optimality, whilst avoiding expensive quantifier elimination. However, as yet, we have neither proven that this guarantees optimality nor found a benchmark program that actually warrants this refinement.

8 Conclusions

We have presented an efficient method for constructing sets of interpolants from inconsistent systems of linear inequalities. Further, we have shown how to apply this method to construct mutual exclusion conditions between clauses of Prolog predicates by applying size abstractions. The conditions inferred are guaranteed to be sufficient, but they are not, at present, guaranteed to be necessary.

Acknowledgements We thank Samir Genaim and Lunjin Lu for their help with determinacy inference and Tom Schilling and Matthieu Sozeau for discussions on type classes. This work was funded by the Royal Society Joint Project JP101405.

References

1. Lu, L., King, A.: Determinacy Inference for Logic Programs. In: ESOP. Volume 3444 of LNCS., Springer-Verlag (2005) 108–123
2. Kriener, J., King, A.: RedAlert: Determinacy Inference for Prolog. TPLP **11** (2011) 537–553
3. Bossi, A., Cocco, N., Fabris, M.: Norms on Terms and their use in Proving Universal Termination of a Logic Program. Theoretical Computer Science **124** (1994) 297–328
4. Martin, J.C., King, A., Soper, P.: Typed Norms for Typed Logic Programs. In: LOPSTR. Volume 1207 of LNCS., Springer (1996) 224–238
5. Weispfenning, V.: The Complexity of Linear Problems in Fields. Journal of Symbolic Computation **5** (1988) 3–27

6. Craig, W.: Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. J. Symb. Log. **22** (1957) 269–285
7. Weissenbacher, G.: Program Analysis with Interpolants. PhD thesis, Magdalen College (2010) http://ora.ouls.ox.ac.uk/objects/uuid:6987de8b-92c2-4309-b762-f0b0b9a165e6.
8. Krajíček, J.: Interpolation Theorems, Lower Bounds for Proof Systems, and Independence Results for Bounded Arithmetic. Journal of Symbolic Logic **62** (1997) 457–486
9. Farkas, J.: Theorie der einfachen Ungleichungen. Journal für die Reine und Angewandte Mathematik **124** (1902) 1–27
10. Avis, D.: lrs: A Revised Implementation of the Reverse Search Vertex Enumeration Algorithm. In Kalai, G., Ziegler, G.M., eds.: Polytopes - Combinatorics and Computation. Birkhauser-Verlag (2000) 177–198
11. Goodman, J.E., O'Rourke, J., eds.: Handbook of Discrete and Computational Geometry. CRC Press (2004)
12. Read, R.C.: Everyone a Winner. Annals of Discrete Mathematics **2** (1978) 107–120
13. Peyton-Jones, S., Jones, M., Meijer, E.: Type Classes: an exploration of the design space. In: ACM SIGPLAN Haskell Workshop. (1997)
14. Morris, J.G., Jones, M.P.: Instance Chains: Type Class Programming Without Overlapping Instances. In Hudak, P., Weirich, S., eds.: ICFP, ACM (2010) 375–386
15. The GHC Team: The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.2.1 (2011) http://www.haskell.org/ghc/docs/latest/html/users_guide.
16. O'Keefe, R.A.: The Craft of Prolog. MIT Press (1990)
17. King, A., Lu, L.: Forward versus Backward Verification of Logic Programs. In: ICLP. Volume 2916 of LNCS., Springer-Verlag (2003) 315–330
18. Kovács, L., Voronkov, A.: Interpolation and Symbol Elimination. In Schmidt, R.A., ed.: CADE. Volume 5663 of LNCS., Springer (2009) 199–213
19. Ziegler, G.M.: Lectures on Polytopes. Springer, New York (1995)
20. Bertsimas, D., Tsitsiklis, J.: Introduction to Linear Optimization. 1st edn. Athena Scientific (1997)
21. Avis, D., Fukuda, K.: Reverse search for enumeration. Discrete Applied Mathematics **65** (1996) 21–46
22. Benoy, F., King, A.: Inferring Argument Size Relationships with CLP(R). In: LOPSTR. Volume 1207 of LNCS., Springer (1996) 204–223
23. Miné, A.: The Octagon Abstract Domain. HOSC **19** (2006) 31–100
24. Sozeau, M.: A New Look at Generalized Rewriting in Type Theory. Journal of Formalized Reasoning **2** (2009) 41–62
25. Sahlin, D.: Determinacy Analysis for Full Prolog. In: Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM (1991) 23–30
26. Mogensen, T.Æ.: A Semantics-Based Determinacy Analysis for Prolog with Cut. In: Ershov Memorial Conference. Volume 1181 of LNCS., Springer (1996) 374–385
27. Dawson, S., Ramakrishnan, C.R., Ramakrishnan, I.V., Sekar, R.C.: Extracting Determinacy in Logic Programs. In: Proceedings of the Tenth International Conference on Logic Programming, MIT Press (1993) 424–438
28. López-García, P., Bueno, F., Hermenegildo, M.V.: Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Analyses. New Generation Computing **28** (2010) 177–206

29. McMillan, K.L.: Applications of Craig Interpolants in Model Checking. In Halbwachs, N., Zuck, L.D., eds.: TACAS. Volume 3440 of Lecture Notes in Computer Science., Springer (2005) 1–12
30. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint solving for interpolation. Journal of Symbolic Computation **45** (2010) 1212–1233