RedAlert: Determinacy Inference for Prolog

JAEL KRIENER and ANDY KING
School of Computing, University of Kent, CT2 7NF, UK.

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

This paper revisits the problem of determinacy inference addressing the problem of how to uniformly handle *cut*. To this end a new semantics is introduced for *cut*, which is abstracted to systematically derive a backward analysis that derives conditions sufficient for a goal to succeed at most once. The method is conceptionally simpler and easier to implement than existing techniques, whilst improving the latter's handling of *cut*. Formal arguments substantiate correctness and experimental work, and a tool called 'RedAlert' demonstrates the method's generality and applicability.

KEYWORDS: abstract interpretation, backwards analysis, Boolean formulae, constraints, cut, determinacy inference, Prolog

1 Introduction

The question of determinacy is constantly on the mind of a good Prolog programmer. It is almost as important to know that a goal will not compute an answer multiply, as it is to know that it will compute the right answer. To this effect, Prolog programmers often use the *cut* to literally cut off all choice points that may lead to additional answers, once a goal has suceeded. A *cut* that is used to (brutely) enforce determinacy in this way is termed a "red cut" (O'Keefe, 1990). O'Keefe also distinguishes between further uses of *cut*, namely "green cut" and "blue cut", which are used to avoid repeating tests in clause selection and exploring clauses which would ultimately fail. Such classifications have been introduced to facilitate reasoning about the determinising effects of *cut* in different contexts. Since these issues are subtle, they motivate developing semantically justified tools which aid the programmer in reasoning about determinacy in the presence of *cut*.

In light of this close connection between determinacy and *cut*, it is clear that *cut* ought to play a prominent role in determinacy analysis. This was recognised by Sahlin (1991), twenty years ago, who proposed an analysis which checks whether a goal can succeed more than once. The analysis abstracts away from the instantiation of arguments within a call which weakens its applicability. Mogensen (1996) recognised the need to ground the work of Sahlin on a formal semantics, yet his work illustrates the difficulty of constructing and then abstracting a semantics for *cut*. Very recently Schneider-Kamp et al. (2010) have shown how a semantics, carefully crafted to facilate abstraction, can be applied to check termination of logic

programs with *cut* on classes of calls. This begs the question whether a semantics can be distilled which is ameniable to inferring determinacy conditions. A good answer to this question will provide the basis for a tool that supports the software development process by providing determinacy conditions in the presence of *cut*.

1.1 Existing methods for determinacy inference

The issue of inferring determinacy in logic programs has been considered before (Lu and King, 2005; King et al., 2006), though neither of the works adequately addressed the *cut*. King et al. (2006) for example present a method for infering determinacy conditions initially for *cut*-free Prolog programs by using suspension analysis in a constraint-based framework. Their motivation is to overcome a limitation of the method presented by Lu and King (2005) that arises from the way in which the order of the literals in the clause influences the strength of the determinacy conditions inferred. To demonstrate this problem, consider the following example:

```
diag([],[],_).
diag([(X,Y)|Xs],[(Y,X)|Ys],[_|Ds]) :- diag(Xs,Ys,Ds).

vert([],[],_).
vert([(X,Y)|Xs],[(X1,Y)|Ys],[_|Ds]) :- {X1 = -X}, vert(Xs,Ys,Ds).

rot(Xs,Ys) :- diag(Xs,Zs,Ys), vert(Zs,Ys,Xs).
```

(The constraint notation in the second clause of `vert` is needed to render the predicate multi-modal.) The method presented by Lu and King (2005) infers the groundness of `Xs` as a sufficient condition for the determinacy of `rot(Xs,Ys)`. It does not detect that the groundness of `Ys`, too, is sufficient for determinacy. This is because the method only considers the left-to-right flow of information from one goal to the next. For instance, if `rot(Xs,Ys)` is called with `Ys` ground, then when the call `diag(Xs,Zs,Ys)` is encountered, neither `Xs` nor `Zs` are ground, hence the call is possibly non-deterministic and therefore the method concludes that only groundness of `Xs` is sufficient for determinacy of `rot(Xs,Ys)`.

In response, King et al. (2006) propose a framework in which the order of the literals in a clause does not impose the implicit assumption that the determinacy of a goal is not affected by the bindings subsequently made by a later goal. To demonstrate, notice that if `Ys` is ground then the execution of `vert(Zs,Ys,Xs)` grounds `Zs`, which is sufficient for the earlier goal `diag(Xs,Zs,Ys)` to be deterministic as well. They achieve this by delaying execution of a goal until a mutual exclusion condition between its clauses is fulfilled and then using suspension inference (Genaim and King, 2008) to infer a determinacy condition for the goals that constitute the body of a clause. This allows them to infer the determinacy condition `Xs` $\vee$ `Ys` for the goal `rot(Xs,Ys)`. Notice, however, the irony in solving a problem that arises from the failure to abstract away from the temporal order of execution by adding temporal complexity into the program.

1.2 Limitations of existing methods

However, the limitations of (King et al., 2006) become sharply apparent when considering the way that the framework is extended to *cut*: Their method is extended by strengthening the determinacy condition for a predicate to ensure that calls before a *cut* are invoked with ground arguments only. While this treatment is sufficient to handle green and blue *cut*s, it means that a *cut* will invariably strengthen the determinacy conditions derived. This is unsatisfactory when considering red *cut*s, given that they are used to ensure determinacy. In that case, the presence of *cut* ought to have a weakening effect on determinacy conditions. To demonstrate, consider the following pair of predicates:

```
memberchk(X,L) :- member(X,L), !.
member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```

In the framework of King et al. (2006), `memberchk` inherits its determinacy conditions from `member` and (if necessary) strengthens them to ensure that the arguments in the call to `member` are ground. In this situation, the determinacy condition derived for `member` is *false*, which cannot be strengthened within the domain of boolean constraints. Therefore the determinacy condition derived for `memberchk` is *false* as well. However, it should be obvious that the effect of the red *cut* in this situation is to make `memberchk` deterministic *independently of the determinacy of* `member`. This example demonstrates that in the presence of *cut*, determinacy conditions on predicates cannot be derived by a straightforward compositional method where parent predicates inherit their conditions from their sub-predicates. Rather, the method needs to allow for weakening and disregarding of determinacy information in the transition from parent to sub-predicates. Aiming to develop a uniform technique for handling *cut* along these lines, this paper makes the following contributions:

- it presents a concise semantics for Prolog with *cut*, based on a *cut*-normal form, that constitutes the basis for a correctness argument (and as far as we are aware the sequence ordering underpinning the semantics is itself novel);
- it presents and proves correct a method for inferring determinacy conditions on Prolog predicates which abstracts over the order of their execution and is both conceptually simpler and easier to implement than previous techniques;
- it reports experimental work that demonstrates precision improvements over existing methods; correctness proofs are given in (Kriener and King, 2011).

2 Preliminaries

2.1 Computational domains

The basic domain underlying the semantics presented in the next section is the set of constraints, *Con*, containing diagonalization constraints of the form $\vec{x} = \vec{y}$, expressing constraints on and bindings to program variables. *Con* is pre-ordered by the entailment relation, $\models$, and closed under disjunction and conjunction. We assume the existence of an extensive projection of θ onto $\vec{x}$, denoted by $\overline{\exists}_{\vec{x}}(\theta)$.

2.1.1 $Con^{\downarrow}$

Our concrete domain is the set of closed non-empty sets of constraints ($Con^{\downarrow}$), which represent program states by capturing all possible bindings to the program variables consistent with a specific set of constraints on the same. The elements of $Con^{\downarrow}$ are constructed thus: For any set of constraints Θ, $\downarrow\{\Theta\} = \{\phi \mid \exists\theta \in \Theta.\phi \models \theta\}$, i.e. the set of all constraints that entail some constraints in Θ. (Observe that $\downarrow\{\mathit{false}\} = \{\mathit{false}\}$.) In this construction, unification is straightforwardly modeled by intersection: The result of unifying variable A with constant c at state $\downarrow\{\Phi\}$ is simply $\downarrow\{A = c\} \cap \downarrow\{\Phi\}$. $Con^{\downarrow}$ is partially ordered by $\subseteq$ and $\langle Con^{\downarrow}, \subseteq, \{\mathit{false}\}, \downarrow\{\mathit{true}\}, \bigcup, \bigcap\rangle$ is a complete lattice. (Notice that $\emptyset \notin Con^{\downarrow}$.) Two projections, one an over-, the other an under-approximation, are defined on $Con^{\downarrow}$ as follows: $\overline{\exists}_{\vec{x}}(\Theta) = \{\overline{\exists}_{\vec{x}}(\theta) \mid \theta \in \Theta\}$, $\overline{\forall}_{\vec{x}}(\Theta) = \{\psi \in \Theta \mid \overline{\exists}_{\vec{x}}(\psi) = \psi\}$. Notice that both projections on $Con^{\downarrow}$ are defined in terms of an arbitrary existential projection on the elements of Con. Each of these two is required later on to ensure soundness: The denotational and success set semantics (Sects. 3.1 and 3.2) need to be over-approximations to be correct. Intuitively, they need to capture *all* possible solutions, even at the cost of letting a few impossible ones slip in. The determinacy semantics (Sect. 3.3) needs to be an under-approximation, which in that context has the effect of strengthening the determinacy condition. Weakening would lead to a loss of soundness there. A renaming operator $\rho_{\vec{x},\vec{y}}$ is defined on $Con^{\downarrow}$ thus: $\rho_{\vec{x},\vec{y}}(\Theta) = \overline{\exists}_{\vec{y}}(\overline{\exists}_{\vec{x}}(\Theta) \cup \{\vec{x} = \vec{y}\})$. (Notice here that $\rho_{\vec{x},\vec{y}}(\Theta) = \rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}(\Theta))$.) For a single constraint θ, $\mathit{vars}(\theta)$ is the set of all variables occurring in θ.
Similar to the notion of definiteness defined by Baker and Søndergaard (1993), a constraint θ *fixes* those variables, in respect to which it cannot be strengthened:

$$\mathit{fix}(\theta) = \{y \mid \forall\psi.((\psi \models \theta \wedge \psi \neq \mathit{false}) \rightarrow \overline{\exists}_{\vec{y}}(\theta) \models \overline{\exists}_{\vec{y}}(\psi))\}$$

Put simply, $\mathit{fix}(\theta)$ is the set of variables that are fixed or grounded by θ.

In addition to these fairly standard constructions, we define two binary operators on $Con^{\downarrow}$ to express more complex relations between its elements: Given $\Theta_1, \Theta_2 \in Con^{\downarrow}$ their mutual exclusion (mux) is the union of all those $\phi \in Con$, which fix a set of variables, on which Θ_1 and Θ_2 are inconsistent:

$$\mathit{mux}(\Theta_1, \Theta_2) = \{\phi \mid \exists Y \subseteq \mathit{fix}(\phi).(\overline{\exists}_Y(\Theta_1) \cap \overline{\exists}_Y(\Theta_2) = \{\mathit{false}\})\}$$

For example, given two sets $\Theta_1 = \downarrow\{A = c, B = d\}$, $\Theta_2 = \downarrow\{A = e, B = d\}$, their mutual exclusion will contain all constraints which fix the variable A to any constant f: $\mathit{mux}(\Theta_1, \Theta_2) = \downarrow\{A = f\}$. Notice that, since Θ_1 and Θ_2 do not disagree on B, fixing B will not distinguish between them and B is therefore not constrained in $\mathit{mux}(\Theta_1, \Theta_2)$. Observe that for $\Theta_1, \Theta_2 \in Con^{\downarrow}$, $\mathit{mux}(\Theta_1, \Theta_2) \in Con^{\downarrow}$, i.e. the mux of two closed sets is closed and that $\mathit{mux}(\Theta_1, \Theta_2) = \downarrow\{\mathit{true}\}$ if Θ_1 or Θ_2 is $\{\mathit{false}\}$.

Given $\Theta_1, \Theta_2 \in Con^{\downarrow}$, their implication is defined as the union of all those elements of $Con^{\downarrow}$ which, when combined with Θ_1, form subsets of Θ_2:

$$\Theta_1 \rightarrow \Theta_2 = \bigcup\{\Phi \mid \Phi \cap \Theta_1 \subseteq \Theta_2\}$$

For example, given two sets $\Theta_1 = \downarrow\{B = d\}$ and $\Theta_2 = \downarrow\{A = c, B = d\}$, $\Theta_1 \rightarrow \Theta_2 = \downarrow\{A = c\}$. Notice that this construction mirrors material implication on boolean

formulae in that the following statements are true for any Θ: $\downarrow\{true\} \to \Theta = \Theta$, $\Theta \to \downarrow\{true\} = \downarrow\{true\}$, $\downarrow\{false\} \to \Theta = \downarrow\{true\}$, $\Theta \to \downarrow\{false\} = \downarrow\{false\}$. Notice also that it is possible to recover Θ_2 from $\Theta_1 \to \Theta_2$ by simply intersecting the latter with Θ_1: $\Theta_1 \to \Theta_2$ is, in a sense, a systematic weakening of Θ_2 by Θ_1.

2.1.2 $Con^{\downarrow}_{seq}$

To model the indeterministic behaviour of Prolog semantically, we extend $Con^{\downarrow}$ to finite sequences of its elements which do not contain the set $\{false\}$, the elements of which are denoted by $\vec{\Theta}$. Concatenation is denoted ':', e.g., $\Theta_1 : [\Theta_2, \Theta_3] = [\Theta_1, \Theta_2, \Theta_3]$. To obtain a top element we add a single infinite sequence, $\omega = [\downarrow\{true\}, \downarrow\{true\}, \ldots]$ and define $Con^{\downarrow}_{seq} = \{(Con^{\downarrow} - \{false\})^n \mid n \geq 0\} \cup \{\omega\}$. $Sub_\ell(\vec{\Theta})$ denotes the set of all subsequences of $\vec{\Theta}$ of length ℓ. Eg: $Sub_2([\Theta_1, \Theta_2, \Theta_3]) = \{[\Theta_1, \Theta_2], [\Theta_2, \Theta_3], [\Theta_1, \Theta_3]\}$. Given a sequence of elements of $Con^{\downarrow}$, Θ^*, $trim(\Theta^*)$ is the result of removing all instances of $\{false\}$ from Θ^*.

$Con^{\downarrow}_{seq}$ can be partially ordered by a prefix-ordering (as is done by Debray and Mishra (1988)). However, under that ordering, the presence of *cut* poses problems in defining suitable monotonic semantic operators. Therefore, we define a partial order on $Con^{\downarrow}_{seq}$ ($\sqsubseteq$) thus: $\forall\vec{\Theta}_1, \vec{\Theta}_2 \in Con^{\downarrow}_{seq}.(\vec{\Theta}_1 \sqsubseteq \vec{\Theta}_2)$ *iff* $\exists\vec{\Phi} \in Sub_m(\vec{\Theta}_2) \cdot (\vec{\Theta}_1 \subseteq_{pw} \vec{\Phi})$ where $|\vec{\Theta}_1| = m$ and $\subseteq_{pw}$ is point-wise comparison on sequences of equal length. The lattice $\langle Con^{\downarrow}_{seq}, \sqsubseteq, [], \omega, \bigsqcup, \bigsqcap\rangle$ is complete (see Appendix), with $\bigsqcap$ and $\bigsqcup$ defined as follows (note that $\bigsqcap$ is needed only to define the fixpoints):

$$\vec{\Theta}_1 \sqcap \vec{\Theta}_2 = \begin{cases} \vec{\Theta}_2 & \text{if } \vec{\Theta}_1 = \omega \\ \vec{\Theta}_1 & \text{if } \vec{\Theta}_2 = \omega \\ \vec{\Theta}_2 \sqcap \vec{\Theta}_1 & \text{if } n < m \\ trim(\bigcup_{pw}\{\vec{\Theta}_1 \cap_{pw} \vec{\Phi} \mid \vec{\Phi} \in Sub_m(\vec{\Theta}_2)\}) & \text{otherwise} \end{cases}$$

where $|\vec{\Theta}_1| = m$, $|\vec{\Theta}_2| = n$ and $\cup_{pw}$ and $\cap_{pw}$ are point-wise union and intersection, which require their operands to be equal length. $\bigsqcap S$ is defined as the lifting of $\sqcap$ to sets in the natural way. From this we can define $\bigsqcup S = \bigsqcap\{\vec{\Theta} \mid \forall\vec{\Phi} \in S.\vec{\Phi} \sqsubseteq \vec{\Theta}\}$ in the normal way. The operators $\downarrow$, $\overline{\exists}_{\vec{x}}$, $\overline{\forall}_{\vec{x}}$ and $\rho_{\vec{x},\vec{y}}$ are all lifted straightforwardly to the elements of $Con^{\downarrow}_{seq}$ as the results of applying the same operations to each member of a given $\vec{\Theta}$. Eg: $\downarrow\overline{\exists}_{\vec{x}}([\Theta_1, \Theta_2]) = [\downarrow\overline{\exists}_{\vec{x}}(\Theta_1), \downarrow\overline{\exists}_{\vec{x}}(\Theta_2)]$. $\bigcup\vec{\Theta}$ denotes the union of all the elements of $\vec{\Theta}$, which itself is an element of $Con^{\downarrow}$. Finally, to save some space in the presentation of the definition of $\mathcal{F}_G$ in Section 3.1, a mixed $\cap$ is defined thus: $(\Phi : \vec{\Phi}) \cap \Theta = (\Phi \cap \Theta) : (\vec{\Phi} \cap \Theta)$.

2.2 Cut normal form

To simplify the presentation of the semantics, we require each predicate in the analysed program to be defined in a single definition of the form $p(\vec{x}) \leftarrow G_1; G_2, !, G_3; G_4$. For example, the `memberchk` and `member` predicates can be transformed to:

```
memberchk(X, L) :- false; (member(X, L), !, true); false.
member(X, L) :- L = [X| _]; (false, !, true); (L = [_| L_1], member(X, L_1)).
```

where `true` and `false` abbreviate post(*true*) and post(*false*) respectively. This does not introduce a loss of generality. (For details on this transformation see Appendix.)

2.3 Syntax and stratification

Given this normal form, the syntax of our programs is defined as follows:

$$
\begin{array}{lll}
\mathit{Head} & ::= p(\vec{x}) & \text{(where } \vec{x} \text{ is a vector of distinct variables)} \\
\mathit{Goal} & ::= \mathsf{post}(\theta) \mid \mathit{Head} \mid \mathit{Goal}, \mathit{Goal} & \\
\mathit{Predicate} & ::= \mathit{Head} \leftarrow \mathit{Goal}\ ;\ \mathit{Goal}\ ,\ !\ ,\ \mathit{Goal}\ ;\ \mathit{Goal} & \\
\mathit{Program} & ::= \epsilon \mid \mathit{Predicate} \cdot \mathit{Program} &
\end{array}
$$

where $\mathsf{post}(\phi)$ indicates that ϕ is added to the current constraint store. Again, $\mathit{vars}(G)$ is the set of variables in a goal G. Further, $\mathit{heads}(P)$ contains the heads of the predicates defined in P.

One would expect that an off-the-shelf denotational semantics could be taken and abstracted to distill a form of determinacy inference. However, the non-monotonic nature of *cut* poses a problem for the definition of such a semantics. In particular, *cut* can be used to define inconsistent predicates, eg: $p \leftarrow \mathit{false}\ ;\ p,\ !,\ \mathit{false}\ ;\ \mathit{true}$. To construct a denotational semantics, we have to address the problem posed by predicates like p, which cannot be assigned a consistent semantics.

Apt et al. (1988) address a parallel problem in the context of negation by banning the use of such viciously circular definitions. To this end, they introduce the notion of stratification with respect to negation. In their view, negation is used 'safely', if all predicates falling under the scope of a negation are defined independently of the predicate in which that negation occurs. Given the similarity between *cut* and *not*, it is natural to adopt a similar approach towards our analogous problem. We define stratification with respect to *cut*, assuming that *cut* is used safely, if only predicates that are defined independently of the context of a *cut*, can decide whether it is reached or not: A program P is *cut*-stratified, if there exists a partition $P = P_1 \cup \ldots \cup P_n$ such that the following two conditions are met for all $1 \leq i \leq n$: 1. For all $p(\vec{x}) \leftarrow G_1; G_2, !, G_3; G4$ in P_i, all calls in G_2 are to predicates in $\bigcup_{j<i} P_j$. 2. For all $p(\vec{x}) \leftarrow G_1; G_2, !, G_3; G4$ in P_i, all calls in G_1, G_3 and G_4 are to predicates in $\bigcup_{j\leq i} P_j$. Henceforth, we shall simply write 'stratified' to mean '*cut*-stratified'. Notice that this restriction is almost purely theoretical. In the worst case, a *cut* after a recursive call produces a situation like or similar to that of the predicate p above, which has no stable semantics and in practice introduces an infinite loop. In the best case, such a *cut* is simply redundant. Either way, we have not been able to find such a *cut* in an actual Prolog program, nor have we been able to come up with an example in which such a *cut* is put to good use.

3 Semantics

Given these preliminaries, we can now define a denotational semantics for Prolog with *cut* (section 3.1), over $\mathit{Con}^{\downarrow}_{\mathit{seq}}$, which is expressive enough to capture multiple answers, and a determinacy semantics (section 3.3), over $\mathit{Con}^{\downarrow}$, suitable for abstraction to boolean conditions. The success set semantics presented in between these two (section 3.2) provides a link between them.

3.1 Denotational semantics

To establish a basis for arguing the determinacy semantics presented in the following sections correct, we define a denotational semantics for Prolog with *cut*. The driving intuition here is, that the semantics of a program P is a mapping from goals called in the context of P to sequences of possible answer substitutions. The context is provided by an environment (μ), henceforth called a success environment to distinguish it from other types of environments, which is a mapping from predicate heads and $Con^{\downarrow}_{seq}$ to $Con^{\downarrow}_{seq}$: $Env ::= Head \rightarrow Con^{\downarrow}_{seq} \rightarrow Con^{\downarrow}_{seq}$. The notation $\mu[p(\vec{y}) \mapsto \vec{\Theta}]$ denotes the result of updating μ with a new assignment from $p(\vec{y})$ to $\vec{\Theta}$. For a given program P, the set E_P of success environments is point-wise partially ordered by: $\mu_1 \sqsubseteq \mu_2$ *iff* $\forall p(\vec{y}), \vec{\Theta}.(\mu_1(p(\vec{y}))(\vec{\Theta}) \sqsubseteq \mu_2(p(\vec{y}))(\vec{\Theta}))$. For any program P the lattice $\langle E_P, \sqsubseteq, \mu_\perp, \mu_\top, \bigsqcup, \bigsqcap \rangle$ is complete, where:

$$
\begin{array}{lll}
\mu_\perp = \lambda p(\vec{y})\vec{\Theta}.[] & & \mu_\top = \lambda p(\vec{y})\vec{\Theta}.\omega \\
\mu_1 \sqcup \mu_2 = \mu_3 \; s.t. \; \forall \vec{\Theta}, p(\vec{y}) \in heads(P).(\mu_3(p(\vec{y}))\vec{\Theta} & = & \mu_1(p(\vec{y}))\vec{\Theta} \sqcup \mu_2(p(\vec{y}))\vec{\Theta}) \\
\mu_1 \sqcap \mu_2 = \mu_3 \; s.t. \; \forall \vec{\Theta}, p(\vec{y}) \in heads(P).(\mu_3(p(\vec{y}))\vec{\Theta} & = & \mu_1(p(\vec{y}))\vec{\Theta} \sqcap \mu_2(p(\vec{y}))\vec{\Theta})
\end{array}
$$

And $\bigsqcup$ and $\bigsqcap$ are lifted to sets of environments in the normal way.

Definition 1
For a given stratified program P, its semantics - μ_P - is defined as a fixpoint of $\mathcal{F}_P$:

$$
\begin{array}{lcl}
\mathcal{F}_P & :: & Program \rightarrow Env \rightarrow Env \\
\mathcal{F}_P[\![\epsilon]\!]\mu & = & \mu \\
\mathcal{F}_P[\![P \cdot Ps]\!]\mu & = & \mathcal{F}_P[\![Ps]\!](\mu[p(\vec{y}) \mapsto (\mathcal{F}_H[\![P]\!]\mu)(p(\vec{y}))]) \\
 & where & P = p(\vec{y}) \leftarrow B
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{F}_H & :: & Predicate \rightarrow Env \rightarrow Env \\
\mathcal{F}_H[\![p(\vec{y}) \leftarrow B]\!]\mu & = & \mu[p(\vec{y}) \mapsto \lambda\vec{\Theta}\cdot \downarrow \overline{\exists}_{\vec{y}}(\mathcal{F}_G[\![G_1]\!]\mu\vec{\Theta} : \vec{\Psi})] \\
 & where & \vec{\Psi} = \begin{cases} \mathcal{F}_G[\![G_3]\!]\mu[\Phi] & if \; \mathcal{F}_G[\![G_2]\!]\mu\vec{\Theta} = \Phi : \vec{\Phi} \\ \mathcal{F}_G[\![G_4]\!]\mu\vec{\Theta} & otherwise \end{cases} \\
 & and & B = G_1; G_2, !, G_3; G_4
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{F}_G & :: & Goal \rightarrow Env \rightarrow Con^{\downarrow}_{seq} \rightarrow Con^{\downarrow}_{seq} \\
\mathcal{F}_G[\![G]\!]\mu[] & = & [] \\
\mathcal{F}_G[\![\mathsf{post}(\phi)]\!]\mu(\Theta : \vec{\Theta}) & = & trim(\downarrow\{\phi\} \cap \Theta : \mathcal{F}_G[\![\mathsf{post}(\phi)]\!]\mu\vec{\Theta}) \\
\mathcal{F}_G[\![p(\vec{x})]\!]\mu(\Theta : \vec{\Theta}) & = & (\downarrow \rho_{\vec{y},\vec{x}}(\; \mu \; p(\vec{y}) \; (\downarrow \rho_{\vec{x},\vec{y}}([\Theta])))) \cap \Theta : \mathcal{F}_G[\![p(\vec{x})]\!]\mu\vec{\Theta} \\
 & where & p(\vec{y}) \in dom(\mu) \\
 & and & vars(\vec{x}) \cap vars(\vec{y}) = \emptyset \\
\mathcal{F}_G[\![G_1, G_2]\!]\mu(\Theta : \vec{\Theta}) & = & \mathcal{F}_G[\![G_2]\!]\mu(\mathcal{F}_G[\![G_1]\!]\mu(\Theta : \vec{\Theta}))
\end{array}
$$

Observe that given a stratified program $P = P_1 \cup \ldots \cup P_n$, $\mathcal{F}_P$ is monotonic, under our sub-sequence order, within each stratum P_i. By Tarski's theorem, $\mathcal{F}_P[\![P_i]\!]$ has a least fixed point. μ_P can therefore be defined as the result of evaluating all strata in order from lowest to highest, starting with $\mu_\perp$ and then taking the least fixed point of the previous stratum as input to the evaluation of the next stratum.

The crucial part is in $\mathcal{F}_H$, which updates the assignments in the success environment and reflects the possible indeterminacy in a predicate by splitting the

resulting sequence up into the possibility resulting from executing G_1 and that resulting from either executing G_3 or G_4, depending on the success of G_2. Given a call to a predicate, $\mathcal{F}_G$ imposes onto each open possibility (i.e. each member of $\vec{\Theta}$) the constraints associated with that predicate in the given μ. The constraints are determined by the application of μ to that predicate, after first applying projection and renaming operations required to match formal and actual parameters. Information about other variables, which is lost in that process, is recovered by intersecting the result of the predicate call with the previous state of computation. The effect of this is, that constraints on the variables that the predicate is called on are strengthened in accordance with its definition, while those on all other variables are preserved. Given a goal of the form '$\mathsf{post}(\phi)$' or 'G_1, G_2', $\mathcal{F}_G$ does what you would expect: In the former case, it imposes ϕ onto each open possibility in the current state of computation, filtering out those possibilities which fail as a result. In the latter case, it successively evaluates G_1 and G_2. Notice further that given an empty sequence (i.e. a failed state of computation), $\mathcal{F}_G$ simply returns an empty sequence, regardless of its other parameters.

Example 1
To illustrate, suppose `member(A,S)` and `memberchk(A,S)` are called at a point in a program where there is only one possible set of bindings $\Theta = \downarrow\{A = 3 \wedge S = [3,2,3]\}$.
$\mathcal{F}_G[\![member(A,S)]\!]\ \mu\ [\Theta] = [\Theta \cap \downarrow\{S = [A|\_]\}, \Theta]$
$\mathcal{F}_G[\![memberchk(A,S)]\!]\ \mu\ [\Theta] = [\Theta \cap \downarrow\{S = [A|\_]\}]$

3.2 Success set semantics

For the purposes of the determinacy inference, a coarser representation of the constraints under which a goal can succeed is given by the following pair of functions.

Definition 2
For a given program P, $S_G : Goal \to Con^{\downarrow}$ and $S_H : Head \to Con^{\downarrow}$ are defined as the least maps, such that:

$$
\begin{array}{lcl}
S_G[\![\mathsf{post}(\phi)]\!] & = & \downarrow\{\phi\} \\
S_G[\![p(\vec{x})]\!] & = & \downarrow \rho_{\vec{y},\vec{x}}(S_H[\![p(\vec{y})]\!]) \\
 & \multicolumn{2}{l}{where\ p(\vec{y}) \leftarrow B \in P} \\
 & \multicolumn{2}{l}{and\ \ vars(\vec{x}) \cap vars(\vec{y}) = \emptyset} \\
S_G[\![G_1, G_2]\!] & = & S_G[\![G_1]\!] \cap S_G[\![G_2]\!] \\
 & & \\
S_H[\![p(\vec{y})]\!] & = & \downarrow \overline{\exists}_{\vec{y}}(S_G[\![G_1]\!] \cup S_G[\![G_2, G_3]\!] \cup S_G[\![G_4]\!]) \\
 & \multicolumn{2}{l}{where\ p(\vec{y}) \leftarrow B \in P\ and\ B = G_1\ ;\ G_2\ ,\ !\ ,\ G_3\ ;\ G_4}
\end{array}
$$

Example 2
To illustrate consider again `member` and `memberchk`: $S_G[\![memberchk(A,S)]\!] = S_G[\![member(A,S)]\!] = \downarrow\{S = [A|\_]\} \cup \downarrow\{S = [\_, A|\_]\} \cup \downarrow\{S = [\_, \_, A|\_]\} \cup \ldots$.

Theorem 1 states that S is a sound over-approximation of $\mathcal{F}$:

Theorem 1
$\bigcup \mathcal{F}_G[\![G]\!]\mu_P\vec{\Theta} \subseteq (\bigcup \vec{\Theta}) \cap S_G[\![G]\!]$ Proof: See Appendix.

3.3 Determinacy semantics

With these in place, we can construct and prove correct a group of functions to derive a set of constraints which guarantee the determinacy of a goal in the context of a program P, its determinacy condition, henceforth abbreviated to 'dc'. As before, the context is provided as an environment: A determinacy environment (δ) is a mapping from predicate heads to $Con^{\downarrow}$: $DEnv ::= Head \rightarrow Con^{\downarrow}$. Again, $\delta[p(\vec{y}) \mapsto \Theta]$ is an update operation. As above, the set E_P^d of determinacy environments for a program P is partially ordered point-wise by: $\delta_1 \sqsubseteq \delta_2$ *iff* $\forall p(\vec{y}).(\delta_1(p(\vec{y})) \subseteq \delta_2(p(\vec{y})))$. The lattice $\langle E_P^d, \sqsubseteq, \delta_\bot, \delta_\top, \bigsqcup, \bigsqcap \rangle$ is complete, with:

$$\begin{array}{ll} \delta_\bot = \lambda p(\vec{y}).\{false\} & \delta_\top = \lambda p(\vec{y}).\downarrow\{true\} \\ \delta_1 \sqcup \delta_2 = \delta_3 \ such\ that\ \forall p(\vec{y}) \in heads(P) \cdot (\delta_3(p(\vec{y})) & = \delta_1(p(\vec{y})) \cup \delta_2(p(\vec{y}))) \\ \delta_1 \sqcap \delta_2 = \delta_3 \ such\ that\ \forall p(\vec{y}) \in heads(P) \cdot (\delta_3(p(\vec{y})) & = \delta_1(p(\vec{y})) \cap \delta_2(p(\vec{y}))) \end{array}$$

And again, $\bigsqcup$ and $\bigsqcap$ are lifted to sets in the normal way.

Definition 3
The determinacy semantics - δ_P - of a program P is the greatest fixpoint of $\mathcal{D}_P[\![P]\!]$:

$$\begin{array}{lcl} \mathcal{D}_P & :: & Program \rightarrow DEnv \rightarrow DEnv \\ \mathcal{D}_P[\![\epsilon]\!]\delta & = & \delta \\ \mathcal{D}_P[\![P \cdot Ps]\!]\delta & = & \mathcal{D}_P[\![Ps]\!](\delta[p(\vec{y}) \mapsto (\mathcal{D}_H[\![P]\!]\delta)(p(\vec{y}))]) \\ & where & P = p(\vec{y}) \leftarrow B \end{array}$$

$$\begin{array}{lcl} \mathcal{D}_H & :: & Predicate \rightarrow DEnv \rightarrow DEnv \\ \mathcal{D}_H[\![p(\vec{y}) \leftarrow B]\!]\delta & = & \delta[p(\vec{y}) \mapsto\downarrow \overline{\forall}_{\vec{y}}(\mathcal{D}_G[\![G_1]\!]\delta \\ & & \qquad \cap (S_G[\![G_2]\!] \rightarrow \mathcal{D}_G[\![G_3]\!]\delta) \\ & & \qquad \cap \mathcal{D}_G[\![G_4]\!]\delta \cap \Theta_1 \cap \Theta_2)] \\ & where & \Theta_1 = mux(S_G[\![G_1]\!], S_G[\![G_4]\!]) \\ & and & \Theta_2 = mux(S_G[\![G_1]\!], S_G[\![G_2, G_3]\!]) \\ & and & p(\vec{y}) \leftarrow G_1 \ ; \ G_2 \ , \ ! \ , \ G_3 \ ; \ G_4 \in P \end{array}$$

$$\begin{array}{lcl} \mathcal{D}_G & :: & Goal \rightarrow DEnv \rightarrow Con^{\downarrow} \\ \mathcal{D}_G[\![\mathsf{post}(\phi)]\!]\delta & = & \downarrow\{true\} \\ \mathcal{D}_G[\![p(\vec{x})]\!]\delta & = & \downarrow \rho_{\vec{y},\vec{x}}\overline{\forall}_{\vec{y}}(\delta(p(\vec{y}))) \\ & where & p(\vec{y}) \in dom(\delta) \\ \mathcal{D}_G[\![G_1, G_2]\!]\delta & = & (S_G[\![G_2]\!] \rightarrow \mathcal{D}_G[\![G_1]\!]\delta) \cap (S_G[\![G_1]\!] \rightarrow \mathcal{D}_G[\![G_2]\!]\delta) \end{array}$$

Given a goal of the form '$\mathsf{post}(\phi)$', $\mathcal{D}_G$ returns $\downarrow\{true\}$ since the goal cannot introduce indeterminacy in the computation. As before, given a predicate call, $\mathcal{D}_G$ applies the projection and renaming necessary to match parameters before calling $\mathcal{D}_H$. Notice that the projection used here is $\overline{\forall}$, since an under-approximation is required to derive a sufficient condition. $\mathcal{D}_H$ maps predicates defined in *cut* normal form to a condition that entails: (a) the dc for G_1, (b) the dc for G_3 weakened by the success set of G_2 - the intuition here being that the dc for G_3 will only be relevant if G_2 can succeed and therefore its dc can be weakened by the success set of G_2 - (c) the dc for G_4, and finally mutual exclusion conditions for the two possibilities arising from the structure of the predicate definition. (The case that needs to be excluded

is that of G_1 succeeding and subsequently G_2 and G_3 succeeding or subsequently G_2 failing and G_4 succeeding.) Finally, when given a compound goal 'G_1, G_2', $\mathcal{D}_G$ returns a condition that entails both the dc for G_2 weakened by the success set of G_1 and the dc for G_1 weakened by the success set of G_2. The intuition here is, that the temporal order of execution is irrelevant. Weakening the dc for G_2 by the success set of G_1 is intuitive, since one can safely assume that G_1 will have succeeded at the point when determinacy of G_2 needs to be enforced. But similarly, when enforcing determinacy on G_1, one can safely assume that G_2 *will* succeed, since both G_1 and G_2 need to succeed for the compound goal to succeed.

Example 3
Consider again `member` and `memberchk`. Observe that $\mathcal{D}_G[\![member(A,S)]\!]\,\delta = \{false\}$ since $mux(S_G[\![G_1]\!], S_G[\![G_4]\!]) = \{false\}$ is a component of $\mathcal{D}_H[\![member(X,L)]\!]\delta$, where $G_1 = (L = [X|\_])$ and $G_4 = (L = [\_|L_1], member(X, L_1))$. `member` is therefore inferred to be non-deterministic for exactly the right reason: There is no groundedness condition on its parameters such that only one of its clauses can succeed.
$\mathcal{D}_G[\![memberchk(A,S)]\!]\ \delta =\downarrow\ \rho_{\vec{y},\vec{x}}\overline{\forall}_{\vec{y}}(\downarrow\{true\} \cap (S_G[\![member(A,S)]\!] \to \downarrow\{true\}) \cap \downarrow\{true\} \cap mux(\{false\},\{false\}) \cap mux(\{false\}, S_G[\![member(A,S), true]\!]))$
$= \downarrow\{true\}$
The crucial observation here is, that $\mathcal{D}_G[\![member(A,S)]\!]\ \delta$ is not required in this construction at all; `memberchk` does not simply inherit its condition from `member`.

Theorem 2 states that, in the context of a stratified program P, the condition given by $\mathcal{D}_G[\![G]\!]\delta_P$ is indeed sufficient to guarantee the determinacy of a call to G:

Theorem 2
If $\Theta \subseteq \mathcal{D}_G[\![G]\!]\delta_P$ then $|\mathcal{F}_G[\![G]\!]\mu_P[\Theta]| \leq 1$ for stratified P (i.e. $P = P_0 \cup \ldots \cup P_n$).
Proof: See Appendix

4 Abstraction

In order to synthesize a determinacy inference from the above determinacy semantics, we systematically under-approximate sets of constraints with boolean formulae that express groundness conditions. *Pos*, however, is augmented with a constant for falsity, so as to express unsatisfiable requirements. The abstract domain $\langle Pos_\bot, \models, true, false, \wedge, \vee\rangle$ is a complete lattice (Armstrong et al., 1998) and to define the abstraction of a single atomic constraint we introduce:

$$\alpha_{\vec{x}}(\theta) = \big(\bigwedge(vars(\vec{x}) \cap fix(\theta)) \wedge \neg \bigvee(vars(\vec{x}) \setminus fix(\theta))\big) \vee \bigwedge vars(\vec{x})$$

For example, if $\theta = A{=}c$ then $\alpha_{\langle A\rangle}(\theta) = A$, while $\alpha_{\langle A,B,C\rangle}(\theta) = (A \wedge \neg B \wedge \neg C) \vee (A \wedge B \wedge C)$. Notice that finiteness is achieved by limiting the scope to a finite vector of variables $\vec{x}$. A Galois connection can then be established thus:

$$\begin{array}{ll} \alpha_{\vec{x}} \quad :: Con^\downarrow \to Pos_\bot & \gamma_{\vec{x}} \quad :: Pos_\bot \to Con^\downarrow \\ \alpha_{\vec{x}}(\Theta) = \bigvee\{\alpha_{\vec{x}}(\theta) \mid \theta \in \Theta \wedge \theta \neq false\} & \gamma_{\vec{x}}(f) = \bigcup\{\Theta \in Con^\downarrow \mid \alpha_{\vec{x}}(\Theta) \models f\} \end{array}$$

For instance, if $\Theta = \downarrow\{A{=}c, B{=}d\}$ then $\alpha_{\langle A,B\rangle}(\Theta) = A \wedge B$.

The following two propositions and two axioms establish relations between the concrete notions of implication, mutual exclusion and the projections and their abstract counterparts. (Notice that abstract implication is simply boolean implication.)

Abstract Implication Proposition 1 establishes the link between concrete ($\rightarrow$) and abstract ($\Rightarrow$) implication as follows:

Proposition 1
If $\Theta_1 \subseteq \gamma_{\vec{x}}(f_1)$ and $\gamma_{\vec{x}}(f_2) \subseteq \Theta_2$ then $\gamma_{\vec{x}}(f_1 \Rightarrow f_2) \subseteq \Theta_1 \rightarrow \Theta_2$ Proof: See Appendix.

Abstract Mutual Exclusion In order to construct an abstract mutual exclusion operator we need to approximate elements of $Con^{\downarrow}$. We do so with depth-k abstractions which are finite sets $\Theta^{DK} \subseteq Con$ such that each atomic constraint θ of the form $x = t$ occurring in Θ^{DK} has a term t whose depth does not exceed k. From these we synthesize boolean requirements sufficient for mutual exclusion thus:

$$mux^{\alpha}_{\vec{x}}(\Theta_1^{DK}, \Theta_2^{DK}) = \vee \left\{ \wedge Y \;\middle|\; \begin{array}{l} Y \subseteq vars(\vec{x}) \quad \wedge \\ \forall \theta_1 \in \Theta_1^{DK}, \theta_2 \in \Theta_2^{DK}.(\overline{\exists}_Y(\theta_1) \wedge \overline{\exists}_Y(\theta_2) = \bot) \end{array} \right\}$$

Notice, again, that $mux^{\alpha}_{\vec{x}}(\Theta_1^{DK}, \Theta_2^{DK}) = \mathit{true}$ if either of Θ_1^{DK} or Θ_2^{DK} is $\{\mathit{false}\}$.

Example 4
Consider $mux^{\alpha}_{\langle X,L\rangle}(\{L = []\}, S_G[\![G_4]\!]^{DK})$ where $G_4 = (L = [\_|L_1], member(X, L_1))$. If depth $k = 3$, then $S_G[\![G_4]\!]^{DK} = \{\theta_1, \theta_2\}$ where $\theta_1 = (L_1 = [X|\_] \wedge L = [\_|L_1])$ and $\theta_2 = (L_1 = [\_, X|\_] \wedge L = [\_|L_1])$. In this situation $mux^{\alpha}_{\langle X,L\rangle}(\{L = []\}, S_G[\![G_4]\!]^{DK})$ is $L \vee (L \wedge X) = L$.

Proposition 2 states how this abstract construction and the concrete one are related:

Proposition 2
$\gamma_{\vec{x}}(mux^{\alpha}_{\vec{x}}(\Theta_1^{DK}, \Theta_2^{DK})) \subseteq mux(\Theta_1, \Theta_2)$ Proof: See Appendix.

Abstract Projections Had we defined a specific concrete projection on single constraints, we could synthesis abstract ones in the standard way (Cousot and Cousot, 1979). However, since both concrete projection operators on $Con^{\downarrow}$ are defined in terms of an arbitrary projection on single constraints, we follow Giacobazzi (1993, Sect.7.1.1) in simply requiring the following to hold for any such projection:

$$\overline{\exists}_{\vec{x}}(\gamma(f)) \subseteq \gamma(\overline{\exists}^{\alpha}_{\vec{x}}(f)) \qquad \gamma(\overline{\forall}^{\alpha}_{\vec{x}}(f)) \subseteq \overline{\forall}_{\vec{x}}(\gamma(f))$$

In addition to the above two axioms, a requirement on the relation between concrete and abstract renaming functions in the context of universal projection is stipulated:

$$\gamma_{vars(\vec{x})}(\rho^{\alpha}_{\vec{y},\vec{x}}\overline{\forall}^{\alpha}_{\vec{y}}(f)) \subseteq \rho_{\vec{y},\vec{x}}\overline{\forall}_{\vec{x}}(\gamma_{vars(\vec{y})}(f))$$

4.1 Abstract success semantics

The last construction that needs to be abstracted in order to mechanise the determinacy semantics presented above is the success set construction S.

Definition 4
The abstract success semantics is defined as the least maps S^α_G, S^α_H such that:

$$\begin{array}{lcl} S^\alpha_G[\![\mathsf{post}(\phi)]\!] & = & \alpha_{vars(\phi)}(\phi) \\ S^\alpha_G[\![p(\vec{x})]\!] & = & \downarrow \rho^\alpha_{\vec{y},\vec{x}}(\overline{\exists}^\alpha_{\vec{y}}(S^\alpha_H[\![p(\vec{y})]\!])) \\ & \textit{where } p(\vec{y}) \leftarrow B \in P & \\ S^\alpha_G[\![G_1, G_2]\!] & = & S^\alpha_G[\![G_1]\!] \wedge S^\alpha_G[\![G_2]\!] \end{array}$$

$$\begin{array}{lcl} S^\alpha_H[\![p(\vec{y})]\!] & = & \downarrow \overline{\exists}^\alpha_{\vec{y}}(S^\alpha_G[\![G_1]\!] \vee S^\alpha_G[\![G_2, G_3]\!] \vee S^\alpha_G[\![G_4]\!]) \\ & \textit{where } p(\vec{y}) \leftarrow B \in P \textit{ and } B = G_1 \ ; \ G_2 \ , \ ! \ , \ G_3 \ ; \ G_4 & \end{array}$$

Proposition 3 formalises the connection between S^α and its concrete counterpart:

Proposition 3
$S_G[\![G]\!] \subseteq \gamma_{vars(G)}(S^\alpha_G[\![G]\!])$ Proof: standard.

Depth-k abstractions can be derived analogously to groundness dependencies and therefore we omit these details.

4.2 Determinacy inference

Finally, an abstract determinacy environment (δ^α) is a mapping from predicate heads to Boolean formulae representing groundness conditions on the arguments of the predicate sufficient to guarantee determinacy of a call to that predicate: $ADEnv ::= Head \rightarrow Pos_\perp$. As in the case of determinacy environments, the set of abstract determinacy environments for a given program (E^α_P) is partially ordered point-wise by $\delta^\alpha_2 \sqsubseteq \delta^\alpha_1$ *iff* $\forall p(\vec{y}).(\delta^\alpha_1(p(\vec{y})) \models \delta^\alpha_2(p(\vec{y})))$. The lattice $\langle E^\alpha_P, \sqsubseteq, \delta^\alpha_\perp, \delta^\alpha_\top, \bigsqcup, \bigsqcap \rangle$ is complete, where $\delta^\alpha_\top = \lambda p(\vec{y}).true$, $\delta^\alpha_\perp = \lambda p(\vec{y}).false$ and $\bigsqcup$ and $\bigsqcap$ are constructed analogously to the case of concrete environments. For a given program P, its abstract determinacy semantics – δ^α_P – is defined as the greatest fixed point of $\mathcal{D}^\alpha_P[\![P]\!]\delta^\alpha_\top$, where $\mathcal{D}^\alpha_P$ is given by the following construction which, unsurprisingly, is very similar in structure to the definition of $\mathcal{D}_P$: (We write $(S_G[\![G]\!])^{DK}$ as $S^{DK}_G[\![G]\!]$.)

Definition 5

$$\begin{array}{lcl} \mathcal{D}^\alpha_P & :: & Program \rightarrow ADEnv \rightarrow ADEnv \\ \mathcal{D}^\alpha_P[\![\epsilon]\!]\delta^\alpha & = & \delta^\alpha \\ \mathcal{D}^\alpha_P[\![P \cdot Ps]\!]\delta^\alpha & = & \mathcal{D}_P[\![Ps]\!](\delta^\alpha[p(\vec{y}) \mapsto (\mathcal{D}^\alpha_H[\![P]\!]\delta^\alpha)(p(\vec{y}))]) \\ & \textit{where } P = p(\vec{y}) \leftarrow B & \end{array}$$

$$\begin{array}{lcl} \mathcal{D}^\alpha_H & :: & Predicate \rightarrow ADEnv \rightarrow ADEnv \\ \mathcal{D}^\alpha_H[\![p(\vec{y}) \leftarrow B]\!]\delta^\alpha & = & \delta^\alpha[p(\vec{y}) \mapsto \overline{\forall}^\alpha_{\vec{y}}(\mathcal{D}^\alpha_G[\![G_1]\!]\delta^\alpha \\ & & \qquad \wedge (S^\alpha_G[\![G_2]\!] \Rightarrow \mathcal{D}^\alpha_G[\![G_3]\!]\delta^\alpha) \\ & & \qquad \wedge \mathcal{D}^\alpha_G[\![G_4]\!]\delta^\alpha \wedge f_1 \wedge f_2] \\ & \textit{where } f_1 = mux^\alpha_{vars(\vec{y})}(S^{DK}_G[\![G_1]\!], S^{DK}_G[\![G_4]\!]) & \\ & \textit{and } \ f_2 = mux^\alpha_{vars(\vec{y})}(S^{DK}_G[\![G_1]\!], S^{DK}_G[\![G_2, G_3]\!]) & \\ & \textit{and } \ B = G_1; G_2, !, G_3; G_4 & \end{array}$$

$$
\begin{array}{lcl}
\mathcal{D}^{\alpha}_{G} & :: & \mathit{Goal} \rightarrow \mathit{ADEnv} \rightarrow \mathit{Pos}_{\perp} \\
\mathcal{D}^{\alpha}_{G}[\![\mathsf{post}(\phi)]\!]\delta^{\alpha} & = & \mathit{true} \\
\mathcal{D}^{\alpha}_{G}[\![p(\vec{x})]\!]\delta^{\alpha} & = & \rho^{\alpha}_{\vec{y},\vec{x}}\overline{\forall}^{\alpha}_{\vec{y}}(\delta^{\alpha}(p(\vec{y}))) \\
 & \mathit{where} & p(\vec{y}) \in \mathit{dom}(\delta^{\alpha}) \\
\mathcal{D}^{\alpha}_{G}[\![G_1, G_2]\!]\delta^{\alpha} & = & (S^{\alpha}_{G}[\![G_2]\!] \Rightarrow \mathcal{D}^{\alpha}_{G}[\![G_1]\!]\delta^{\alpha}) \wedge (S^{\alpha}_{G}[\![G_1]\!] \Rightarrow \mathcal{D}^{\alpha}_{G}[\![G_2]\!]\delta^{\alpha})
\end{array}
$$

Theorem 3 states that each parallel application of $\mathcal{D}_P$ and $\mathcal{D}^{\alpha}_P$ preserves the correspondence between the dc and its abstract counterpart and Corollary 1 states a direct consequence of this, namely that the same correspondence holds between the greatest fixpoints of these constructions.

Theorem 3
$\forall i \in \mathbb{N} : \gamma_{vars(G)}(\mathcal{D}^{\alpha}_{G}[\![G]\!]\delta^{\alpha}_{i}) \subseteq \mathcal{D}_{G}[\![G]\!]\delta_i$, where δ^{α}_{i} (resp. δ_i) are the results of i applications of $\mathcal{D}^{\alpha}_{P}[\![P]\!]$ (resp. $\mathcal{D}_{P}[\![P]\!]$) to $\delta^{\alpha}_{\top}$ (resp. $\delta_{\top}$). Proof: See Appendix.

Corollary 1
$\gamma_{vars(G)}(\mathcal{D}^{\alpha}_{G}[\![G]\!]\delta^{\alpha}_{P}) \subseteq \mathcal{D}_{G}[\![G]\!]\delta_P$ Proof: Straightforward.

These two statements establish, in effect, that δ^{α}_{P} is correct with respect to (i.e. is a sound under-approximation of) δ_P. The significance of this is, that the correctness of $\mathcal{D}_{G}[\![G]\!]\delta_P$ as a determinacy condition for G, which was proved in the last section, is carried over to $\mathcal{D}^{\alpha}_{G}[\![G]\!]\delta^{\alpha}_{P}$. Since the latter is finite and can be mechanised, an implementation is therefore proven to give a correct (if possibly overly strong) determinacy condition for a goal G in the context of a stratified program P.

5 Implementation

The determinacy inference specified in the previous section is realised as a tool called 'RedAlert', using a simple bottom-up fixpoint engine in the style of those discussed by Codish and Søndergaard (2002). Boolean formulae are represented in CNF as lists of lists of non-ground variables. In this way, renaming is straightforward and conjunction is reduced to list-concatenation (Howe and King, 2001). However, disjunction, implication and existential quantifier elimination are performed by enumerating prime implicants (Brauer et al., 2011), which reduces these operations to incremental SAT. The solver is called through a foreign language interface following Codish et al. (2008). It is interesting to note, that we have not found any of the benchmarks to be non-stratified, though even if this were the case, a problematic *cut* could be discarded albeit at the cost of precision.

In the case of the `memberchk` predicate mentioned in the introduction, the implementation does indeed infer *true* as its determinacy condition, as desired. To discuss a more interesting case, consider the partition predicate of quicksort.

```
pt([], _, [], []).
pt([X | Xs], M, [X | L], G) :- X =< M, !, pt(Xs, M, L, G).
pt([X | Xs], M, L, [X | G]) :- pt(Xs, M, L, G).
```

The method presented in King et al. (2006) handles this *cut* by enforcing monotonicity on the predicate. To this end, the negation of the constraint before the *cut* ($X > M$) is conceptually added to the last clause and the *cut* then disregarded. The

benchmark	*org*	*new*	*impr*	*mean*	*benchmark*	*org*	*new*	*impr*	*mean*
asm	44	157	5	0.6	peval	108	14	2	1
crypt_wamcc	11	12	2	2	nandc	12	5	2	0
semi	22	19	0	0	life	10	11	7	1.85
qsort	3	1	1	1	ronp	16	5	4	1
browse	15	7	1	2	tsp	23	2	10	1.4
ga	58	102	2	1.5	flatten	27	25	6	1.5
dialog	30	11	3	0	neural	34	23	3	0
unify	26	33	3	1.33	nbody	48	34	11	2
peep	20	189	0	0	boyer	26	95	4	0
read	42	89	0	0	qplan	65	41	7	2.57
reducer	31	57	9	2	simple_analyzer	60	50	9	2.22

Table 1. *Comparison*

groundness requirement inferred in this way for $pt(w, x, y, z)$ is $(w \wedge x) \vee (x \wedge y \wedge z)$. The determinacy condition inferred for the same predicate by the method presented in this paper is: $w \wedge (y \vee z)$, which is clearly an improvement, though still sufficient. Improvements similar to this can be observed when analysing a number of benchmark programs. Table 1 summarises the results of this comparison on 22 benchmarks (which are available at `http://www.cs.kent.ac.uk/people/staff/amk/cut-normal-form-benchmarks.zip`). Under '*org*' is the number of predicate definitions in the original program. To give a measure of the impact of the *cut* normal form transformation, under '*new*' is the number of new predicates introduced by it. Under '*impr*' is the number of predicates in the original benchmark (excluding any newly introduced ones) on which the determinacy inference is improved by our method over King et al. (2006). Under '*mean*' is the mean size of improvement (i.e. the mean number of variables which occur in the previous determinacy condition but not in the new one). The results show a uniform improvement. Note that randc, dialog, neural and boyer give precision improvements but no determinancy conditions are inferred which involve strictly fewer variables. The runtime for the groundness analysis, the depth-k analysis and the backwards analysis, that propagates determinacy requirements against the control flow, are all under a second for all benchmarks (and not even SCCs are considered in the bottom-up fixpoint calculations). However, the overall runtime is up to an order of magnitude greater, due to the time required to calculate the mutual exclusion conditions. This is because the definition of the abstract mutual exclusion in section 4 is inherently exponential in the arity of a predicate. This is currently the bottleneck.

6 Related Work

Determinacy inference and analysis As mentioned above, Lu and King (2005) and King et al. (2006) address the problem of inferring determinacy conditions on a predicate. Since their limitations have been discussed above, we will not repeat them here. Dawson et al. (1993) present a method for inferring determinacy information from a program by adding constraints to the clauses of a predicate which allow the inference of mutual exclusion conditions between these clauses rather than

determinacy conditions for a whole predicate. Sahlin (1991) presents a method for determinacy analysis, based on a partial evaluation technique for full Prolog which detects whether there are none, one or more than one ways a goal can succeed. This approach has been developed by Mogensen (1996) (see below). Le Charlier et al. (1994) present a top-down framework for abstract interpretation of Prolog which is based on sequences of substitutions and can be instantiated to derive an analysis equivalent to that of Sahlin (1991).

Denotational semantics for Prolog with cut Mogensen (1996) constructs a denotational semantics for Prolog with *cut* based on streams of substitutions as the basis for a formal correctness argument for the determinacy analysis. The problem of constructing a denotational semantics for Prolog with *cut* has been addressed before by Billaud (1990), Debray and Mishra (1988) and de Vink (1989) a good 20 years ago, around the same time that Apt et al. (1988) first published their theory of non-monotonic reasoning, introducing the idea of stratification. Billaud (1990) constructs an elegant denotational semantics based on streams of states of computation and proves it correct with respect to an operational semantics. Debray and Mishra (1988) construct a more complex semantics over a domain of sequences of substitutions, comparable to our $Con^{\downarrow}_{seq}$, which is partially ordered, in contrast to $Con^{\downarrow}_{seq}$, by a prefix-ordering, rather than a sub-sequence-ordering. Both proceed by first defining a semantics for *cut*-free Prolog and then extending it to *cut*. In both cases, they argue monotonicity for the former of these constructions and appear to assume that it carries over to the latter. Finally de Vink (1989), too, presents a denotational semantics of Prolog with *cut*. His approach is probably closest to ours, using environments to represent the context provided by a program in a similar fashion. However, as in the case of Debray and Mishra (1988), no argument is provided for the monotonicity of their semantic operators, which casts some doubt over the question whether the semantics is well-defined. Common to all these approaches is the view of *cut* as essentially an independent piece of syntax. This view requires *cut* to be treated on a par with success and failure, having an evaluation by itself, which creates the need for complex constructions involving the introduction and later elimination of *cut*-flags into the streams or sequences, to semantically simulate the effect that *cut* has on a computation. In contrast, we view *cut* as essentially relational. In our view, a *cut* has no semantics of its own, but only affects the evaluation of the goals in the context where it occurs. This reliefs us of the need for systematically introducing and eliminating *cut*-flags.

7 Conclusions

This paper has presented a determinacy inference for Prolog with *cut*, which treats *cut* in a uniform way, while being more elegant and powerful than previously existing methods. The inference has been proved correct with respect to a novel denotational semantics for Prolog with *cut*. We have demonstrated the viability of the method by reporting on the performance of an implementation thereof and evaluating it against a comparable existing method.

Acknowledgements This work was inspired by the cuts that are ravaging the UK, but funded by a ACM-W scholarship and a DTA bursary. We thank Lunjin Lu and Samir Genaim for discussions that provided the backdrop for this work. We thank Michel Billaud for sending us copies of his early work and for his comments on the wider literature. We also thank an anonymous reviewer for invaluable help with the proofs in the appendix.

References

Apt, K. R., Blair, H. A., and Walker, A. 1988. Towards a Theory of Declarative Knowledge. In *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 89–148.

Armstrong, T., Marriott, K., Schachte, P., and Søndergaard, H. 1998. Two Classes of Boolean Functions for Dependency Analysis. *Science of Computer Programming 31,* 1, 3–45.

Baker, N. and Søndergaard, H. 1993. Definiteness Analysis for CLP($\mathcal{R}$). *Australian Computer Science Communications 15,* 1, 321–332. Proceedings of the Sixteenth Australian Computer Science Conf.

Billaud, M. 1990. Simple Operational and Denotational Semantics for Prolog with Cut. *Theoretical Computer Science 71,* 2, 193–208.

Brauer, J., King, A., and Kriener, J. 2011. Existential Quantification as Incremental SAT. In *Twenty-third International Conference on Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Lecture Notes in Computer Science. Springer-Verlag. To appear.

Codish, M., Lagoon, V., and Stuckey, P. 2008. Logic Programming with Satisfiability. *Theory and Practice of Logic Programming 8,* 1, 121–128.

Codish, M. and Søndergaard, H. 2002. Meta-Circular Abstract Interpretation in Prolog. In *The Essence of Computation: Complexity, Analysis, Transformation*, T. Æ. Mogensen, D. Schmidt, and I. H. Sudborough, Eds. Lecture Notes in Computer Science, vol. 2566. Springer, 109–134.

Cousot, P. and Cousot, R. 1979. Systematic Design of Program Analysis Frameworks. In *Sixth Annual ACM Sypmosium on Principles of Programming Languages*. 269–282.

Dawson, S., Ramakrishnan, C. R., Ramakrishnan, I. V., and Sekar, R. C. 1993. Extracting Determinacy in Logic Programs. In *Proceedings of the Tenth International Conference on Logic Programming*. MIT Press, 424–438.

de Vink, E. P. 1989. Comparative Semantics for Prolog with Cut. *Science of Computer Programming 13,* 1, 237–264.

Debray, S. K. and Mishra, P. 1988. Denotational and Operational Semantics for Prolog. *Journal of Logic Programming 5,* 1, 81–91.

Genaim, S. and King, A. 2008. Inferring Non-Suspension Conditions for Logic Programs with Dynamic Scheduling. *ACM Transactions on Computational Logic 9,* 3 (November).

Giacobazzi, R. 1993. Semantic Aspects of Logic Program Analysis. Ph.D. thesis, Dipartimento di Informatica, Universitá di Pisa.

Howe, J. M. and King, A. 2001. Positive Boolean Functions as Multiheaded Clauses. In *Proceedings of the Seventeenth International Conference on Logic Programming*, P. Codognet, Ed. Lecture Notes in Computer Science, vol. 2237. Springer, 120–134.

King, A., Lu, L., and Genaim, S. 2006. Detecting Determinacy in Prolog Programs. In *Proceedings of the Twenty-second International Conference on Logic Programming*. Lecture Notes in Computer Science, vol. 4079. Springer, 132–147.

Kriener, J. and King, A. 2011. Appendix for RedAlert: Determinacy Inference for Prolog. Tech. Rep. 1-11, School of Computing, University of Kent, CT2 7NF, UK. Available from: `http://arxiv.org/corr/home`.

Le Charlier, B., Rossi, S., and Van Hentenryck, P. 1994. An Abstract Interpretation Framework which Accurately Handles Prolog Search-Rule and the Cut. In *Symposium on Logic Programming*. MIT Press, 157–171.

Lu, L. and King, A. 2005. Determinacy Inference for Logic Programs. In *Fourteenth European Symposium on Programming*, S. Sagiv, Ed. Lecture Notes in Computer Science, vol. 3444. Springer, 108–123.

Mogensen, T. Æ. 1996. A Semantics-Based Determinacy Analysis for Prolog with Cut. In *Ershov Memorial Conference*. Lecture Notes in Computer Science, vol. 1181. Springer, 374–385.

O'Keefe, R. A. 1990. *The Craft of Prolog*. MIT Press, Cambridge, MA, USA.

Sahlin, D. 1991. Determinacy Analysis for Full Prolog. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM, 23–30.

Schneider-Kamp, P., Giesl, J., Ströder, T., Serebrenik, A., and Thiemann, R. 2010. Automated Termination Analysis for Logic Programs with Cut. *Theory and Practice of Logic Programming 10,* 4-6, 365–381.