Computer Science at Kent

Determinacy Inference by Suspension Inference

Andy King, Lunjin Lu and Samir Genaim

Technical Report No. 2-05
October 2005

Abstract The aim of determinacy inference is to infer a class of calls to a given logic program, for which execution will generate at most one answer and generate the answer only once. Two serious impediments to accurate determinacy inference are: (1) the way bindings imposed by a rightmost goal can make a leftmost goal deterministic; (2) the way determinacy is often enforced with cut. Rather surprisingly, this paper shows how: problem (1) can be tackled by recasting determinacy inference as a problem in concurrency; problem (2) can be addressed within this concurrency framework. Experimental evaluation shows that the new analysis can infer richer classes of deterministic calls for many programs.

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent, CT2 7NF, UK

1 Introduction

Understanding the determinacy behaviour of a logic program is important in program specialisation, implementation and program development. To this end, determinacy inference [16] has been proposed as an analysis for inferring conditions on goals that are sufficient to assure determinacy. The key difference between determinacy checking [15, 17, 18] and determinacy inference is that the former verifies that a given goal will generate at most one answer (if it yields any at all) whereas the latter infers, in a single application of the analysis, a class of goals that are deterministic. When applied within program development, determinacy inference is most insightful when it infers a class of calls that differs from what the programmer expects. If the class is smaller than expected, then either the predicate is unintentionally non-deterministic (i.e. buggy), or the analysis is insufficiently precise. If the class is larger than anticipated, then either the predicate possesses properties that the programmer overlooked (i.e. subtle sub-goal interactions that induce determinacy), or it has been coded incorrectly.

The fundamental problem with determinacy inference as previously proposed [16] is that it only considers left-to-right flow of bindings between sub-goals. Consider, for instance, the compound goal `q(X), r(X)` and the database:

```
q(a).    q(b).    r(a).
```

The compound goal is always deterministic, no matter how it is called, due to the way the bindings generated by the rightmost sub-goal constrain the leftmost sub-goal. However, the analysis of [16] would only infer that the goal is deterministic if called with `X` ground. Observe that `r(X)` is always deterministic and that it grounds `X`. Since `q(X)` is deterministic if called with `X` ground, it follows that the vacuous groundness condition of *true* is sufficient for determinacy of the compound goal. One contribution of this paper is to show that this deficiency in determinacy inference can be remedied by reformulating the determinacy inference problem as a suspension inference problem [6]. The intuition is to add delay declarations to the program so that a goal can only be selected if no more than one clause in the matching predicate can generate an answer. The sub-goals `r(X)` and `q(X)` are thus selected when, respectively, the groundness conditions of `X` and *true* are satisfied. Suspension inference then deduces that the condition *true* is sufficient for the compound goal not to suspend. Correctness results provided in the paper show that non-suspension conditions can then be safely reinterpreted as determinacy conditions. Quite from improving precision, this transformational approach enables existing analysis machinery [6] to be reapplied to determinacy inference. This is an advantage because developing any analysis is a major undertaking. Another contribution of the paper is that it explains how the cut operator can be supported within this transformational approach. Yet another, is experimental work that empirically demonstrates the value of the techniques proposed in the paper. In fact, during the experiments a number of anomalies were discovered in classical benchmark programs [20].

The paper is organised as follows. Section 2 presents a worked example which illustrates the new approach. The section that follows is more theoretical in nature and formally proves the correctness of the new analysis. Section 4 explains how cut can be accommodated within the framework and Section 5 evaluates the analysis. Finally, Sections 6 and 7 present the related work and conclude.

2 Worked Example

Since the correctness argument is necessarily theoretical, this section illustrates the key ideas in the new approach to determinacy inference by way of an example:

```
(1) rev([],[]).
(2) rev([X|Xs],Ys) :- rev(Xs,Zs), app(Zs,[X],Ys).
(3) app([],X,X).
(4) app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).
```

2.1 The common ground

The chief novelty in the previous approach to determinacy inference [16] was in the way success patterns for the individual clauses of a predicate were used to infer mutual exclusion conditions for a call to that predicate. The new analysis builds on these mutual exclusion conditions. Such a condition, when satisfied by a call, ensures that no more than one clause of the matching predicate can lead to a successful derivation. To illustrate, consider characterising the success patterns with an argument-size analysis in which size is measured as list-length:

(1) $\texttt{rev}(x_1, x_2)$:- $x_1 = 0$, $x_2 = 0$.
(2) $\texttt{rev}(x_1, x_2)$:- $x_1 \geq 1$, $x_1 = x_2$.
(3) $\texttt{app}(x_1, x_2, x_3)$:- $x_1 = 0$, $x_2 \geq 0$, $x_2 = x_3$.
(4) $\texttt{app}(x_1, x_2, x_3)$:- $x_1 \geq 1$, $x_2 \geq 0$, $x_1 + x_2 = x_3$.

An algorithm that takes, as input, success patterns and produces, as output, mutual exclusion conditions is detailed in our previous paper [16]. Rather than repeat the algorithm, we give the intuition of how rigidity relates to mutual exclusion. Observe that if $\texttt{rev}(x_1, x_2)$ is called with x_1 bound to a rigid list, then if clause (1) succeeds the size constraint $x_1 = 0$ implies that the x_1 argument was initially bound to an empty list. Clause (2) cannot succeed with this call since the size constraint $x_1 \geq 1$ requires x_1 to be a list with at least one element. The argument follows in the other direction, hence the rigidity condition x_1 is sufficient for mutual exclusion. By similar reasoning, the rigidity of x_2 is also sufficient hence the combined condition $x_1 \vee x_2$ is also a mutual exclusion condition. Repeating this argument for $\texttt{app}(x_1, x_2, x_3)$ yields $x_1 \vee (x_2 \wedge x_3)$ [16].

2.2 The problem

The value of mutual exclusion conditions is that if all sub-goals encountered whilst solving a goal satisfy their conditions, then the goal is deterministic [16]. This motivates the application of backward analysis [10] which infers conditions on goals which ensure that a given set of assertions are not violated. By adding assertions that check calls meet their mutual exclusion conditions, the backward analysis will infer conditions on goals that assure determinacy. To illustrate backward analysis, consider clause (2) on the previous page. Since `[X]` is rigid, it is enough for $d_2 = \texttt{Zs} \vee \texttt{Ys}$ to hold for `app(Zs,[X],Ys)` to satisfy its condition. By inspecting the success patterns of `rev` it follows that after the sub-goal `rev(Xs,Zs)` the rigidity property $f_1 = \texttt{Xs} \wedge \texttt{Zs}$ holds. Thus, if the condition $f_1 \rightarrow d_2$ holds before the sub-goal, then $(f_1 \rightarrow d_2) \wedge f_1$, hence d_2, holds after the sub-goal, and thus the mutual exclusion condition for `app(Zs,[X],Ys)` is satisfied. Since $d_1 = \texttt{Xs} \vee \texttt{Zs}$ is the mutual exclusion condition for the `rev(Xs,Zs)`, $d_1 \wedge (f_1 \rightarrow d_2) = (\texttt{Xs} \vee \texttt{Zs})$ guarantees that both sub-goals of the body satisfy their conditions when encountered. To satisfy this condition, it is enough for `[X|Xs]` to be rigid, that is, for `rev` to be called with a rigid first argument.

The astute reader will notice that `rev` is deterministic when called with a rigid second argument. To see this, observe that each answer to `rev(Xs, Zs)` will instantiate `Zs` to a list of different size. Now consider executing the compound goal `app(Zs,[X],Ys), rev(Xs, Zs)` with `Ys` rigid. The rigidity of `Ys` ensures that `app(Zs,[X],Ys)` is deterministic and thus `Zs` has at most one solution of fixed size. This, in turn, guarantees that `rev(Xs, Zs)` can yield at most one answer, hence the compound goal is deterministic. (Actually, `rev` gives one answer and then loops in this mode, though this does not compromise determinacy and it is therefore disappointing that [16] only discovers one deterministic mode).

2.3 The solution

The two deterministic modes stem from different flows of bindings between the sub-goals. This motivates an analysis that considers different schedulings of sub-goals and selects a sub-goal only when its mutual exclusion condition is satisfied. In effect, mutual exclusion conditions are interpreted as delay conditions like so:

```
delay rev(X, Y) until rigid_list(X) ; rigid_list(Y).
rev([],[]).
rev([X|Xs],Ys) :- rev(Xs,Zs), app(Zs,[X],Ys).
delay app(X, Y, Z) until rigid_list(X) ; (rigid_list(Y) , rigid_list(Z)).
app([],X,X).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).
```

The delay declarations, which are reminiscent of Gödel syntax [9], block `rev` and `app` goals until combinations of their arguments are bound to rigid lists. Suspension inference [6] (which can be considered to be a black-box) is then applicable to this transformation of the original program and, for this derived program, can infer classes of initial goals which cannot lead to suspending states. For these initial goals, the program cannot reduce to a state that only contains suspending sub-goals, that is, sub-goals which violate their mutual exclusion conditions. Since each sub-goal satisfies its condition when executed, the computation is deterministic. Furthermore, executing any such initial goal under left-to-right selection will also generate at most one answer (it may not terminate but again this will not undermine determinacy). Applying suspension inference [6] to the above program yields the rigidity conditions of $x_1 \vee x_2$ and $x_1 \vee (x_2 \wedge x_3)$ for $\texttt{rev}(x_1, x_2)$ and $\texttt{app}(x_1, x_2, x_3)$ respectively, as desired.

3 The semantics, monotonicity and correctness

To define the transform and argue correctness in full generality, i.e. for constraints, it is necessary to: (1) clarify what properties are required to hold on the underlying domain of constraints; (2) specify a multiset semantics to observe non-determinacy, i.e., one that collects the answers to a query in a multiset; (3) formalise a concurrency semantics that can interpret the synchronisation operators introduced by the transform; (4) specify the transformation itself; and (5) prove correctness. Sections 3.1–3.5 respectively address each of these issues.

3.1 Computational domain of constraints

The transformation is formulated in terms of a generic domain of constraints Con that is pre-ordered by an entailment (implication) relation which is denoted by $\models \subseteq \mathsf{Con} \times \mathsf{Con}$. Con is pre-ordered (rather than partially ordered) because syntactically different constraints may entail one another, for example, the systems $x = 1, y = 2x$ and $x + y = 3, y = x + 1$. To express semantic equivalence on Con, the relation $\equiv \subseteq \mathsf{Con} \times \mathsf{Con}$ is introduced which is defined by $\theta_1 \equiv \theta_2$ iff $\theta_1 \models \theta_2$ and $\theta_2 \models \theta_1$. The relation $\equiv$ factorises Con to obtain a domain of equivalence classes $\mathsf{Con}/\!\equiv$ in which a class is denoted $[\theta]_\equiv$ where θ is a representative member of the class. The pre-ordering $\models$ extends from Con to $\mathsf{Con}/\!\equiv$ by $[\theta_1]_\equiv \models [\theta_2]_\equiv$ iff $\theta_1 \models \theta_2$ to obtain the poset $\langle \mathsf{Con}/\!\equiv, \models \rangle$. To clarify when the transformation is applicable, a series of axioms are introduced to formalise the assumptions made about the underlying constraint domain. The following axiom, for example, states that assumptions made about conjunctions of constraints:

Axioms 1 (conjoining constraints)

- $[\theta_1]_\equiv \wedge [\theta_2]_\equiv \models [\theta_i]_\equiv$ for $i = 1, 2$
- If $[\sigma]_\equiv \models [\theta_1]_\equiv$ and $[\sigma]_\equiv \models [\theta_2]_\equiv$ then $[\sigma]_\equiv \models [\theta_1]_\equiv \wedge [\theta_2]_\equiv$

We additionally assume that the meet semi-lattice contains a top element, $[true]_\equiv$, that represents the vacuous constraint.

The reoccurring name-clash problem arises in our multiset and concurrency semantics as the problem of avoiding name clashes between the environment of a call and that of the agent it invokes. An elegant solution to this problem is offered by the use of hiding operators $\overline{\exists}_{\vec{x}}([\theta]_\equiv)$ [19] where $\vec{x}$ is a vector of variables. Such an operator projects onto $\vec{x}$; it hides information in $[\theta]_\equiv$ that pertains to variables other than those in $\vec{x}$. The following axioms echo the axiomatisation of cylindric algebras [7] that are classically used [19] to specify hiding:

Axioms 2 (projecting constraints)

- $[\theta_1]_\equiv \models [\theta_2]_\equiv$ then $\overline{\exists}_{\vec{x}}([\theta_1]_\equiv) \models \overline{\exists}_{\vec{x}}([\theta_2]_\equiv)$

- $[\theta]_\equiv \models \overline{\exists}_{\vec{x}}([\theta]_\equiv)$
- $\overline{\exists}_{\vec{x}}([\theta_1]_\equiv) \wedge \overline{\exists}_{\vec{x}}([\theta_2]_\equiv) \models \overline{\exists}_{\vec{x}}([\theta_1]_\equiv \wedge \overline{\exists}_{\vec{x}}([\theta_2]_\equiv))$

By combining axioms, the last can be strengthened $\overline{\exists}_{\vec{x}}([\theta_1]_\equiv) \wedge \overline{\exists}_{\vec{x}}([\theta_2]_\equiv) = \overline{\exists}_{\vec{x}}([\theta_1]_\equiv \wedge \overline{\exists}_{\vec{x}}([\theta_2]_\equiv))$. Furthermore, by putting $[\theta_1]_\equiv = \overline{\exists}_{\vec{x}}([\theta]_\equiv)$ and $[\theta_2]_\equiv = [\theta]_\equiv$ it follows that $\overline{\exists}_{\vec{x}}(\overline{\exists}_{\vec{x}}([\theta]_\equiv)) \wedge \overline{\exists}_{\vec{x}}([\theta]_\equiv) = \overline{\exists}_{\vec{x}}(\overline{\exists}_{\vec{x}}([\theta]_\equiv))$, hence $\overline{\exists}_{\vec{x}}(\overline{\exists}_{\vec{x}}([\theta]_\equiv)) \models \overline{\exists}_{\vec{x}}([\theta]_\equiv)$, thus $\overline{\exists}_{\vec{x}}(\overline{\exists}_{\vec{x}}([\theta]_\equiv)) = \overline{\exists}_{\vec{x}}([\theta]_\equiv)$, which shows that hiding is idempotent.

To model parameter passing in our multiset and concurrency semantics, the underlying constraint system is assumed to contain so-called diagonal constraints of the form $d_{\vec{x},\vec{y}}$ where $\vec{x}$ and $\vec{y}$ are vectors of disjoint variables. The constraint $d_{\vec{x},\vec{y}}$ provides a mechanism for equating formal arguments with actual arguments and, when combined with projection, effects a form of variable renaming. Diagonal constraints are assumed to satisfy the following properties where $\mathsf{var}(o)$ denotes the set of variables occurring in a syntactic object o:

Axioms 3 (diagonal constraints) Let $\mathsf{var}(\vec{x}) \cap \mathsf{var}(\vec{y}) = \emptyset$. Then

- $[d_{\vec{x},\vec{y}}]_\equiv \models [d_{\vec{y},\vec{x}}]_\equiv$
- $[true]_\equiv \models \overline{\exists}_{\vec{x}}([d_{\vec{x},\vec{y}}]_\equiv)$
- $\overline{\exists}_{\vec{x}}([d_{\vec{x},\vec{y}}]_\equiv \wedge \overline{\exists}_{\vec{y}}([d_{\vec{x},\vec{y}}]_\equiv \wedge \overline{\exists}_{\vec{x}}([\theta]_\equiv))) = \overline{\exists}_{\vec{x}}([\theta]_\equiv)$

Definition 3.1 Let $\mathsf{var}(\vec{x}) \cap \mathsf{var}(\vec{z}) = \emptyset$ and $\mathsf{var}(\vec{y}) \cap \mathsf{var}(\vec{z}) = \emptyset$. Then

$$\rho_{\vec{x},\vec{y}}([\theta]_\equiv) = \begin{cases} \overline{\exists}_{\vec{y}}(\overline{\exists}_{\vec{x}}([\theta]_\equiv) \wedge [d_{\vec{x},\vec{y}}]_\equiv) & \text{if } \mathsf{var}(\vec{x}) \cap \mathsf{var}(\vec{y}) = \emptyset \\ \rho_{\vec{z},\vec{y}}(\rho_{\vec{x},\vec{z}}([\theta]_\equiv)) & \text{otherwise} \end{cases}$$

The intuition is that the operator $\rho_{\vec{x},\vec{y}}([\theta]_\equiv)$ simultaneously substitutes each variable of $\vec{x}$ with its corresponding variable in $\vec{y}$. Note that the vectors $\vec{x}$ and $\vec{y}$ do not have to be variable disjoint to apply the renaming operator. Since renaming is central to our semantics (whereas diagonal elements are merely a device for constructing renaming operators), we show that the following properties follow from the axiomatisation of diagonal constraints and then, henceforth, consider renaming to be an indivisible, atomic operator.

Proposition 3.1 (renaming constraints)

- If $[\theta_1]_\equiv \models [\theta_2]_\equiv$ then $\rho_{\vec{x},\vec{y}}([\theta_1]_\equiv) \models \rho_{\vec{x},\vec{y}}([\theta_2]_\equiv)$

- $\rho_{\vec{x},\vec{y}}([\theta_1]_\equiv) \wedge \rho_{\vec{x},\vec{y}}([\theta_2]_\equiv) \models \rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}([\theta_1]_\equiv) \wedge [\theta_2]_\equiv)$
- $\overline{\exists}_{\vec{y}}(\rho_{\vec{x},\vec{y}}([\theta]_\equiv)) \models \rho_{\vec{x},\vec{y}}([\theta]_\equiv)$
- $\rho_{\vec{x},\vec{y}}(\rho_{\vec{y},\vec{x}}([\theta]_\equiv)) = \overline{\exists}_{\vec{y}}([\theta]_\equiv)$

Note that the proofs for these results, and those for the other results stated in the body of the paper, are given in appendix A.

When constructing the multiset and concurrency semantics, and later establishing a correctness result that relates the two via the transformation, it turns out to be technically easier to formulate the semantics over the computational domain of sets of constraints that are closed under entailment. This domain of closed sets (ideals) is defined by $\wp^{\downarrow}(\mathsf{Con}) = \{\Theta \subseteq \mathsf{Con} \mid \downarrow\Theta = \Theta\}$ where the closure operator $\downarrow$ is given by $\downarrow\Theta = \{\theta_1 \in \mathsf{Con} \mid \exists\theta_2 \in \Theta.[\theta_1]_\equiv \models [\theta_2]_\equiv\}$. The crucial point is that operations over $\langle \mathsf{Con}/\equiv, \models, \wedge\rangle$ can be simulated within $\langle \wp^{\downarrow}(\mathsf{Con}), \subseteq, \cap\rangle$ which has a simpler but richer structure. The following
proposition asserts that a conjunction of constraints the form $[\theta_1]_\equiv \wedge [\theta_2]_\equiv$ can be faithfully modelled by set intersection $\Theta_1 \cap \Theta_2$ where $\Theta_i = \downarrow\{\theta_i\}$ for $i = 1, 2$.

Proposition 3.2 (conjoining sets of constraints)
$(\downarrow\Theta_1) \cap (\downarrow\Theta_2) = \{\theta \mid [\theta]_\equiv = ([\theta_1]_\equiv \wedge [\theta_2]_\equiv) \wedge \theta_1 \in \downarrow\Theta_1 \wedge \theta_2 \in \downarrow\Theta_2\}$

Note that $\downarrow\{true\} = \mathsf{Con}$ and that Con coincides with the top element of $\langle \wp^{\downarrow}(\mathsf{Con}), \subseteq, \cap\rangle$. The following definitions complete the lifting by specifying how the projection and renaming operators map over to sets of closed constraints.

Definition 3.2 $\overline{\exists}_{\vec{x}}(\Theta) = \downarrow\{\overline{\exists}_{\vec{x}}([\theta]_\equiv) \mid \theta \in \Theta\}$ and $\rho_{\vec{x},\vec{y}}(\Theta) = \downarrow\{\rho_{\vec{x},\vec{y}}([\theta]_\equiv) \mid \theta \in \Theta\}$

The following propositions state some of the rich relationships hold on and between the versions of projection and renaming that operate over closed sets of constraints. These results build towards monotonicity results (Corollary 3.1) that, in turn, underpin the main correctness result (Theorem 3.1).

Proposition 3.3 (projecting sets of constraints)

- $\downarrow\overline{\exists}_{\vec{x}}(\Theta) = \overline{\exists}_{\vec{x}}(\Theta) = \overline{\exists}_{\vec{x}}(\downarrow\Theta)$
- $\Theta \subseteq \overline{\exists}_{\vec{x}}(\Theta) = \overline{\exists}_{\vec{x}}(\overline{\exists}_{\vec{x}}(\Theta))$

- $\overline{\exists}_{\vec{x}}(\Theta_1) \cap \overline{\exists}_{\vec{x}}(\Theta_2) = \overline{\exists}_{\vec{x}}(\downarrow\Theta_1 \cap \overline{\exists}_{\vec{x}}(\Theta_2))$

Proposition 3.4 (renaming sets of constraints)

- $\rho_{\vec{x},\vec{y}}(\Theta) = \downarrow\rho_{\vec{x},\vec{y}}(\Theta) = \rho_{\vec{x},\vec{y}}(\downarrow\Theta) = \overline{\exists}_{\vec{y}}(\rho_{\vec{x},\vec{y}}(\Theta))$
- $\rho_{\vec{x},\vec{y}}(\cup_{i=1}^{n}\Theta_i) = \cup_{i=1}^{n}\rho_{\vec{x},\vec{y}}(\Theta_i)$
- $\rho_{\vec{x},\vec{y}}(\rho_{\vec{y},\vec{x}}(\Theta)) = \overline{\exists}_{\vec{y}}(\Theta) = \rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}(\Theta))$
- $\rho_{\vec{x},\vec{y}}(\Theta_1) \cap \rho_{\vec{x},\vec{y}}(\Theta_2) = \rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}(\Theta_1) \cap \Theta_2)$
- $\rho_{\vec{x},\vec{y}}(\rho_{\vec{y},\vec{x}}(\Theta_1) \cap \downarrow\Theta_2) = \overline{\exists}_{\vec{y}}(\Theta_1) \cap \rho_{\vec{x},\vec{y}}(\Theta_2)$
- If $\Theta_i = \downarrow\Theta_i$ for $i = 1, 2$ then $\rho_{\vec{x},\vec{y}}(\Theta_1) \cap \Theta_2 \neq \emptyset$ iff $\Theta_1 \cap \rho_{\vec{y},\vec{x}}(\Theta_2) \neq \emptyset$.

3.2 Multiset and success set semantics

To express the transformation that maps a sequential program into a concurrent program, it is helpful to express both classes of program within the same language. Thus we adopt concurrent constraint programming (ccp) [19] style in which a program P takes the form $P ::= \epsilon | p(\vec{x}) :- A | P_1.P_2$ where A is an agent that is defined by $A ::= \mathsf{ask}(\Theta) \rightarrow A | \mathsf{tell}(\Theta) | A_1, A_2 | \sum_{i=1}^{n} A_i | p(\vec{x})$ and $\Theta \in \wp^{\downarrow}(\mathsf{Con})$ and throughout $\vec{x}$ denotes a vector of distinct variable arguments. A sequential program is merely a program devoid of any $\mathsf{ask}(\Theta) \rightarrow A$ agents. In the concurrent setting, $\mathsf{ask}(\Theta) \rightarrow A$ blocks A until the constraint store Σ entails the ask constraint Θ, that is, $\Sigma \subseteq \Theta$. In both settings, the agent $\mathsf{tell}(\Theta)$ updates the store from Σ to $\Sigma \cap \Theta$ by imposing the constraints Θ. In a sequential program, the composition operator , is interpreted as a sequencing operator which induces left-to-right control, whereas in a concurrent program, the same operator is interpreted as parallel composition. For both classes of program $\sum_{i=1}^{n} A_i$ is an explicit choice operator which systematically searches all A_i agents in parallel for answers. Due to this operator, predicates are assumed to be defined with exactly one definition of the form $p(\vec{x}) :- A$ without any loss of generality.

The multiset semantics is formulated in terms of multisets of closed sets of constraints which capture whether an answer does not occur, occurs exactly once or occurs multiply to a given query. To distinguish between determinate and non-determinate computations, it is sufficient to consider a restricted class of multisets in which no element occurs more than twice. This set of all multisets coincides with the function space

$\wp^{\downarrow}(\mathsf{Con}) \to \{0,1,2\}$ which will henceforth be abbreviated to $\widehat{\mathsf{Con}}$. The domain $\widehat{\mathsf{Con}}$ is ordered by the multiset inclusion relation $\widehat{\subseteq}$ which is defined $M_1 \widehat{\subseteq} M_2$ iff $M_1(\Theta) \leq M_2(\Theta)$ for all $\Theta \in \wp^{\downarrow}(\mathsf{Con})$. In fact, $\langle \widehat{\mathsf{Con}}, \widehat{\subseteq}, \widehat{\sqcup}, \widehat{\sqcap} \rangle$ is a complete lattice where $M_1 \widehat{\sqcup} M_2 = \lambda\Theta.\max(M_1(\Theta), M_2(\Theta))$ and $M_1 \widehat{\sqcap} M_2 = \lambda\Theta.\min(M_1(\Theta), M_2(\Theta))$. Multiset union over $\widehat{\mathsf{Con}}$ is defined by $M_1 \widehat{\cup} M_2 = \lambda\Theta.\min(M_1(\Theta) + M_2(\Theta), 2)$. Thus if an element occurs singly in both M_1 and M_2, then it occurs twice in $M_1 \widehat{\cup} M_2$. However, if it occurs singly in M_1 and twice in M_2, then it occurs twice in $M_1 \widehat{\cup} M_2$. Henceforth, to simplify the presentation, we write $\wr\int$ for the multiset $\lambda\Theta.0$ and $\wr\Psi, \Sigma, \Sigma\int$, for instance, for $\lambda\Theta.\mathsf{if}\ \Theta = \Psi\ \mathsf{then}\ 1\ \mathsf{else}\ (\mathsf{if}\ \Theta = \Sigma\ \mathsf{then}\ 2\ \mathsf{else}\ 0)$. Furthermore, we adopt multiset comprehensions with the interpretation that the predicate $\Theta \widehat{\in} M$ succeeds once/twice iff Θ occurs once/twice in M. For example, if $M' = \wr\Theta, \Theta, \Sigma\int$ then $\wr\Psi \cup \Sigma \mid \Sigma \widehat{\in} M'\int = \wr\Psi \cup \Theta, \Psi \cup \Theta, \Psi \cup \Sigma\int$. Finally, the cardinality of a multiset M is defined $|M| = \|\{\Theta \in \widehat{\mathsf{Con}} \mid M(\Theta) = 1\}\| + 2\|\{\Theta \in \widehat{\mathsf{Con}} \mid M(\Theta) = 2\}\|$ where $\|S\|$ denotes the cardinality of a set S. For instance, $|M'| = 3$.

The multiset semantics is a mapping of type $\mathsf{Age}_P \to \wp^{\downarrow}(\mathsf{Con}) \to \widehat{\mathsf{Con}}$ where Age_P denote the set of agents that can be constructed from calls to predicates defined within P. The intuition is that, for a given $A \in \mathsf{Age}_P$, the multiset semantics maps a closed set Σ to a multiset of closed sets. The function space over which the multiset semantics is defined, namely, $\mathsf{Age}_P \to \wp^{\downarrow}(\mathsf{Con}) \to \widehat{\mathsf{Con}}$ is a complete lattice due to the completeness of $\langle \widehat{\mathsf{Con}}, \widehat{\subseteq}, \widehat{\sqcup}, \widehat{\sqcap} \rangle$. Specifically, the ordering $\widehat{\subseteq}$ lifts to $\mathsf{Age}_P \to \wp^{\downarrow}(\mathsf{Con}) \to \widehat{\mathsf{Con}})$ by $f_1 \widehat{\subseteq} f_2$ iff $f_1[\![A]\!](\Theta) \widehat{\subseteq} f_2[\![A]\!](\Theta)$ for all $A \in \mathsf{Age}_P$ and $\Theta \in \wp^{\downarrow}(\mathsf{Con})$. The operations $\widehat{\sqcup}$ and $\widehat{\sqcap}$ can be lifted analogously. With these structures in place, the multiset semantics can be defined as follows:

Definition 3.3 The mapping $\mathcal{M}_P : \mathsf{Age}_P \to \wp^{\downarrow}(\mathsf{Con}) \to \widehat{\mathsf{Con}}$ is the least solution to the following system of recursive equations:

$$
\begin{aligned}
\mathcal{M}_P[\![\mathsf{tell}(\Sigma)]\!](\Theta) &= \mathsf{if}\ \Sigma \cap \Theta = \emptyset\ \mathsf{then}\ \wr\int\ \mathsf{else}\ \wr\Sigma \cap \Theta\int \\
\mathcal{M}_P[\![A_1, A_2]\!](\Theta) &= \widehat{\cup}\wr\mathcal{M}_P[\![A_2]\!](\Sigma) \mid \Sigma \widehat{\in} \mathcal{M}_P[\![A_1]\!](\Theta)\int \\
\mathcal{M}_P[\![\textstyle\sum_{i=1}^{n} A_i]\!](\Theta) &= \widehat{\cup}\wr\mathcal{M}_P[\![A_i]\!](\Theta)\int_{i=1}^{n} \\
\mathcal{M}_P[\![p(\vec{x})]\!](\Theta) &= \wr\Theta \cap \rho_{\vec{y},\vec{x}}(\Sigma) \mid \Sigma \in \mathcal{M}_P[\![A]\!](\rho_{\vec{x},\vec{y}}(\Theta))\int \text{ where } p(\vec{y}) :- A \in P
\end{aligned}
$$

Because $\mathsf{Age}_P \to \wp^{\downarrow}(\mathsf{Con}) \to \widehat{\mathsf{Con}}$ is a complete lattice and each primitive operation is monotonic, it follows by the Knaster-Tarski theorem that the above system of equations admits a least fixpoint and thus $\mathcal{M}_P$ is well-defined. The following lemma, which is a generalisation of the classic lifting lemma [14, p 47], explains how the number of answers obtained for a query to a program relates to the number of answers computed for a more general query.

Lemma 3.1 Let $A \in \mathsf{Age}_P$. Then
$\mathcal{M}_P[\![A]\!](\Omega \cap \Theta) = \wr \Sigma \cap \Omega \mid \Sigma \widehat{\in} \mathcal{M}_P[\![A]\!](\Theta) \wedge \Sigma \cap \Omega \neq \emptyset \wr$

It should be noted that the lemma only holds because the multiset semantics applies parallel search and does not distinguish between failing and diverging computations. To see this, observe that if the agents $A_1 = \mathsf{tell}(x = [\_|y]), p(y)$ and $A_2 = \mathsf{tell}(x = [])$ that occur in $p(x) :- \sum_{i=1}^{2} A_i$ are tried in textual order, a la Prolog, then $p(x)$ will loop and yield no solutions (resp. succeed with one solution) when called under the constraints Con (resp. $\downarrow\{x = []\}$). This does not mean that the multiset semantics cannot be used to reason about determinancy. The crucial point is that any call that is deterministic under the multiset semantics is also deterministic under more stringent control, i.e., the control model of Prolog. The following results, which relate to monotonicity, flow from lemma 3.1:

Corollary 3.1 Let $A \in \mathsf{Age}_P$. Then

- $\cup \mathcal{M}_P[\![A]\!](\Theta) \subseteq \Theta$
- If $\Omega = \cup \mathcal{M}_P[\![A]\!](\Theta)$ then $\mathcal{M}_P[\![A]\!](\Omega \cap \Theta) = \mathcal{M}_P[\![A]\!](\Theta)$
- If $\Theta_1 \subseteq \Theta_2$ then $|\mathcal{M}_P[\![A]\!](\Theta_1)| \leq |\mathcal{M}_P[\![A]\!](\Theta_2)|$
- If $\Theta_1 \subseteq \Theta_2$ then $\cup \mathcal{M}_P[\![A]\!](\Theta_1) \widehat{\subseteq} \cup \mathcal{M}_P[\![A]\!](\Theta_2)$

The following success set semantics provides a bridge between the multiset semantics and some concurrency semantics that will be developed in what follows.

Definition 3.4 The mapping $\mathcal{S}_P : \mathsf{Age}_P \to \wp^{\downarrow}(\mathsf{Con})$ is the least solution to the following system of recursive equations:

$$\begin{aligned} \mathcal{S}_P[\![\mathsf{tell}(\Sigma)]\!] &= \Sigma \\ \mathcal{S}_P[\![A_1, A_2]\!] &= \mathcal{S}_P[\![A_1]\!] \cap \mathcal{S}_P[\![A_2]\!] \\ \mathcal{S}_P[\![\textstyle\sum_{i=1}^{n} A_i]\!] &= \cup\{\mathcal{S}_P[\![A_i]\!]\}_{i=1}^{n} \\ \mathcal{S}_P[\![p(\vec{x})]\!] &= \rho_{\vec{y},\vec{x}}(\mathcal{S}_P[\![A]\!]) \text{ where } p(\vec{y}) :- A \in P \end{aligned}$$

Proposition 3.5 $\cup \mathcal{M}_P[\![A]\!](\Theta) = \Theta \cap \mathcal{S}_P[\![A]\!]$.

3.3 Concurrency semantics

To verify the correctness of the transformation, it is necessary to introduce a concurrency semantics for both observing the presence and verifying the absence of suspending agents. An agent suspends if it can branch to a state in which the only sub-agents that remain to be activated take the form $\mathsf{ask}(\Sigma) \rightarrow A$ where $\Sigma \not\subseteq \Theta$ and Θ is the state of the store. Agents that result in an inconsistent store silently fail and therefore do not suspend. Further, agents that diverge always include a sub-agent that can be activated and hence do not suspend. Therefore, for the purposes of reasoning about suspension, the concurrency semantics does not need to distinguish between failing agents and diverging agents. Whether suspensions manifest themselves or not depends on the quiescent (resting) state of the store, which motivates the construction of the following semantics that was inspired by the quiescent state semantics proposed for deterministic ccp [19].

Definition 3.5 The mapping $\mathcal{Q}_P : \mathsf{Age}_P \rightarrow \wp(\wp^{\downarrow}(\mathsf{Con}) \times \{0,1\})$ is the least solution to the following system of recursive equations:

$$
\begin{aligned}
\mathcal{Q}_P[\![\mathsf{ask}(\Sigma) \rightarrow A]\!] &= \{\langle \Theta, 1\rangle \mid \downarrow\!\Theta = \Theta \wedge \Theta \not\subseteq \Sigma\} \;\cup\; \{\langle \Theta, b\rangle \in \mathcal{Q}_P[\![A]\!] \mid \Theta \subseteq \Sigma\} \\
\mathcal{Q}_P[\![\mathsf{tell}(\Sigma)]\!] &= \{\langle \Theta, 0\rangle \mid \downarrow\!\Theta = \Theta \wedge \Theta \subseteq \Sigma\} \\
\mathcal{Q}_P[\![A_1, A_2]\!] &= \{\langle \Theta, b_1 \vee b_2\rangle \mid \langle \Theta, b_i\rangle \in \mathcal{Q}_P[\![A_i]\!]\} \\
\mathcal{Q}_P[\![\textstyle\sum_{i=1}^{n} A_i]\!] &= \cup\{\mathcal{Q}_P[\![A_i]\!]\}_{i=1}^{n} \\
\mathcal{Q}_P[\![p(\vec{x})]\!] &= \{\langle \Theta, b\rangle \mid \downarrow\!\Theta = \Theta \wedge \langle \Sigma, b\rangle \in \mathcal{Q}_P[\![A]\!] \wedge \rho_{\vec{y},\vec{x}}(\Sigma) = \overline{\exists}_{\vec{x}}(\Theta)\} \\
&\quad \text{where } p(\vec{y}) :- \; A \in P
\end{aligned}
$$

$\mathcal{Q}_P$ is well-defined since $\mathsf{Age}_P \rightarrow \wp(\wp^{\downarrow}(\mathsf{Con}) \times \{0,1\})$ is a complete lattice and each primitive operation in the system of equations is monotonic.

The semantics expresses the resting points of an agent [19] and tags each constraint set with either 1 or 0 to indicate whether or not an agent contains a suspending sub-agent (in the latter case the agent has successfully terminated). Consider, for instance, the agent $\mathsf{ask}(\Sigma) \rightarrow A$ and the set Θ. If $\Theta \not\subseteq \Sigma$ then the agent suspends in Θ, and hence quiesces, and thus the pair $\langle \Theta, 1\rangle$ is included in the set of quiescent states of the agent. Otherwise, if $\Theta \subseteq \Sigma$ and $\langle \Theta, b\rangle$ is a quiescent state of A then $\langle \Theta, b\rangle$ is also a quiescent state of $\mathsf{ask}(\Sigma) \rightarrow A$. Any set Θ such that $\Theta \subseteq \Sigma$ is a succeeding quiescent state of the agent $\mathsf{tell}(\Sigma)$. A compound agent A_1, A_2 quiesces under Θ iff Θ is a quiescent set of both A_1 and A_2. Θ is tagged as suspending iff either A_1 or A_2 suspend in Θ. The branching agent $\sum_{i=1}^{n} A_i$ inherits quiescent states from each of its sub-agents. Finally, an agent $p(\vec{x})$ that invokes an agent A via a definition $p(\vec{y}) :- \; A$ inherits quiescent states from A by the application of projection and renaming. The intuition is that the variables $\vec{x}$ and $\vec{y}$ act as windows on the sets of constraints associated with $p(\vec{x})$ and A in that they hide information not pertaining to $\vec{x}$ and $\vec{y}$ respectively. If these projected sets coincide under renaming and the set for A is quiescent, then set for $p(\vec{x})$ is also quiescent. The following lemma states the natural property that non-suspension is downward closed:

Lemma 3.2 If $\langle \Theta, 0 \rangle \in \mathcal{Q}_P[\![A]\!]$, $\Theta' \in \wp^{\downarrow}(\mathsf{Con})$ and $\Theta' \subseteq \Theta$ then $\langle \Theta', 0 \rangle \in \mathcal{Q}_P[\![A]\!]$.

3.4 Transformation

The transformation is parameterised in terms of (1) an upper approximation (post-fixpoint) of the success set semantics and (2) mutual exclusion function $\mathsf{mux} : \widehat{\mathsf{Con}} \to \wp^{\downarrow}(\mathsf{Con})$ which, given a multiset of closed sets of constraints, derives a set of constraints that intersects with at most one member of the multiset. This generic formulation allows for different mutual exclusion algorithms to be applied that are taylored to specific success set descriptions [16]. The key requirement of mux is specified by the following axiom:

Axioms 4 If $\mathsf{mux}(\wr \Theta_i \S_{i=1}^{n}) = \Sigma$ and $\Sigma \cap \Theta_i \neq \emptyset$ then $\Sigma \cap \Theta_j = \emptyset$ for all $i \neq j$.

Definition 3.6 Let $\mathcal{S}_P[\![A_i]\!] \subseteq \Theta_i$. Then

$$\begin{aligned}
T[\![P_1.P_2]\!] &= T[\![P_1]\!].T[\![P_2]\!] \\
T[\![p(\vec{x}) :- A]\!] &= p(\vec{x}) :- T[\![A]\!] \\
T[\![\mathsf{tell}(\Sigma)]\!] &= \mathsf{tell}(\Sigma) \\
T[\![A_1, A_2]\!] &= T[\![A_1]\!], T[\![A_2]\!] \\
T[\![\textstyle\sum_{i=1}^{n} A_i]\!] &= \mathsf{ask}(\mathsf{mux}(\wr \Theta_i \S_{i=1}^{n})) \to \textstyle\sum_{i=1}^{n} T[\![A_i]\!] \\
T[\![p(\vec{x})]\!] &= p(\vec{x})
\end{aligned}$$

3.5 Correctness of the transformation

The following correctness result is founded on: the algebraic properties of the lifted projection and renaming operators (propositions 3.3 and 3.4); the generalised lifting lemma (lemma 3.1); the monotonicity results for the multiset semantics (corollary 3.1) and the downward closure of non-suspension (lemma 3.2). The theorem states two results. The first result asserts that if an agent in the transformed program is called with a set of constraints imposed, and the agent cannot reduce to a suspending state, then the success set semantics characterises the quiescent state semantics. The second result additionally states the original program is deterministic when called with the same constraints imposed on the store. The corollary merely reiterates the determinacy result for the special case of calls.

Theorem 3.1 Suppose $A \in \mathsf{Age}_P$, $Q = \mathcal{Q}_{T[\![P]\!]}[\![\mathsf{tell}(\Omega), T[\![A]\!]]\!]$ and $\langle \Pi, 1 \rangle \notin Q$ for all $\Pi \in \wp(\mathsf{Con})$. Then

- $\Omega \cap \mathcal{S}_P[\![A]\!] = \cup\{\Theta \mid \langle \Theta, 0 \rangle \in Q\}$.
- $|\mathcal{M}_P[\![A]\!](\Omega)| \leq 1$

Corollary 3.2 Suppose $\langle \Pi, 1 \rangle \notin \mathcal{Q}_{T[\![P]\!]}[\![\mathsf{tell}(\Omega), p(\vec{x})]\!]$ for all $\Pi \in \wp(\mathsf{Con})$. Then $|\mathcal{M}_P[\![p(\vec{x})]\!](\Omega)| \leq 1$

4 The cut, non-monotonicity and incorrectness

The technique for inferring mutual exclusion conditions [16] is not sensitive to the clause ordering. This does not compromise correctness but, due to the presence of cut, the inferred conditions may be overly strong. Rather than refining the semantics (which is a topic within itself [21]) and deriving a new analysis, this section shows how the procedure which computes the mutual exclusion conditions can be refined to accommodate this pruning operator.

The correctness of the analysis is founded on corollary 3.1 which is a monotonicity result that follows from the definition of the multiset semantics. Alas, the cut is a source of non-monotonicity as is illustrated by the `r` predicate:

```
r(X, Y) :- X = a, !, Y = b.
r(X, Y) :- atomic(X).
```

```
p(X, Y) :- q(X), r(X, Y).
q(a).
q(b).
```

If `r(X, Y)` is called under the binding $\{\mathtt{X} \mapsto b\}$ then the downward closure of the set of computed answers is $\Theta_1 = \downarrow\{\mathtt{X} = b\}$. However, if `r(X, Y)` is called with a more general binding – the empty substitution – the downward closure of the set of answers is $\Theta_2 = \downarrow\{\mathtt{X} = a, \mathtt{Y} = b\}$. The predicate is non-monotonic because $\Theta_1 \not\subseteq \Theta_2$. The predicate `p` illustrates the significance of monotonicity in that it shows how non-monotonicity can undermine correctness. To see this, consider applying suspension inference in which the `r(X, Y)` goals are never delayed but `q(X)` goals are only selected when `X` is ground. Suspension inference [16] would infer the vacuous condition of *true* for `p(X, Y)` since the compound goal `q(X), r(X, Y)` can be scheduled in right-to-left order without incurring a suspension. However, this inference is unsafe, since the call `p(X, Y)` yields two answers.

One may think that the problem of non-monotonicity is insurmountable but correctness can be recovered by ensuring that the mutual exclusion conditions enforce monotonicity. The observation is that the occurrence of a cut in a clause cannot compromise monotonicity if any calls (typically tests) that arise before the cut are invoked with ground arguments. The idea is thus to strengthen the condition so as to ensure this. For example, `r(X, Y)` is monotonic if `X` is ground. The justification of this tactic is that in top-down clause selection, if a clause containing a cut succeeds, then any following clause is not considered. This behaviour can be modelled by adding negated calls to each clause that follows the cut (which is sound due to groundness [14]). Consider, for example, the following predicate:

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

For the sake of inferring mutual exclusion conditions, observe that the negation of `X =< M` can be inserted into the last clause since this clause is reached only if the test fails. Then, due to the negated test, the second and third clauses are mutually exclusive if the groundness condition $x_1 \wedge x_2$ holds where x_i describes the groundness of the i'th argument. It is not necessary to reason about cut to deduce that the first and second clauses are mutually exclusive if $x_1 \vee x_3$ holds. Likewise, a mutual exclusion condition for the first and third clause is $x_1 \vee x_4$. The cumulative mutual exclusion condition for the whole predicate is therefore $(x_1 \vee x_3) \wedge (x_1 \vee x_4) \wedge (x_1 \wedge x_2) = x_1 \wedge x_2$. Finally, note that when mechanising this approach, it is not actually necessary to insert the negated calls to deduce the grounding requirements; the negated tests were introduced merely to justify the tactic. Finally, applying this technique to `r` yields the condition x_1.

5 Experimental Results

The analyser is constructed as four components: (1) an argument-size analysis and (2) a depth-k analysis both of which infer success patterns for each clause; (3) an analysis to infer mutual exclusion conditions from the success patterns and generate the transformed concurrent program; and (4) the suspension inference itself which generates the conditions that assure determinacy. The analyser can be used through a web interface located at:

`http://www.cs.kent.ac.uk/~amk/detweb.html`

As well as employing standard norms such as term-size and list-length, the argument-size applies norms automatically derived from the types arising in the program [3, 11]. This, in turn, permits more general mutual exclusion conditions, hence determinacy conditions, to be inferred. Consider, for example, a predicate that succeeds when its single argument is a list of lists. Such a predicate (named `traverse`) can be found at the above URL. This predicate traverses the outer list, calling an auxiliary predicate to check that each element is also a list. By using an argument size analysis based on the term-size norm, it is possible to show that the predicate is deterministic if called with a ground argument. When the program is decorated with types, however, type-based norms enable this determinacy condition to be relaxed to a rigid list of rigid lists. The depth-k analysis also warrants comment since it tracks constraints between the leaves of a depth-k term. Consider, for instance, the `merge` predicate that arises within mergesort program that can again be found at the above URL. Depth-1 analysis will infer the following success patterns for three recursive clauses of `merge`:

```
merge([A|B],[C|D],[A|E]) :- A < C
merge([A|B],[C|D],[A|E]) :- A = C
merge([A|B],[C|D],[C|E]) :- A > C
```

From these success patterns, it can be deduced that the groundness of the first and second arguments is sufficient for mutual exclusion. This condition cannot be inferred unless the depth-k analysis is enriched to track constraints. The remaining component of the mutual exclusion code is the logic that handles cut.

The overhead of substituting a fixpoint engine of [16] with another that realised suspension inference [6] was negligible and thus the evaluation focusses on the issue of precision (scalability was demonstrated previously [16]). Figure 1 details the number of deterministic modes inferred for a series of programs, mainly drawn from [20]. To compare against previous work [16], the first column details the number of modes inferred using the fixpoint engine of [16] and mutual exclusion conditions derived without consideration of the cut, using a depth-k analysis that does not track constraints. The second column details the effect of replacing the fixpoint engine with

Program	A	B	C	D
ack	1	1	1	1
bool	3	3	3	3
circuit	7	9⋆	9	9
connected2	4	4	5⋆	5
courses-rules	5	5	5	5
delete	0	0	0	2⋆
exchange	3	4⋆	4	4
exp	5	5	5	5
fact	0	0	0	1⋆
factorial	6	6	6	6
factorial-iter	0	0	0	2⋆
family	7	7	7	7
fib	0	0	0	1⋆
flatten	1	1	2⋆	2
gcd	7	7	7	7
gcd-arith	0	0	0	1⋆
graph	2	2	2	2
heapify	1	1	1	1
inorder	2	3⋆	3	3
insert-sort	0	0	0	2⋆
lateb	3	3⋆	3	3
lee-route	2	2	3⋆	5⋆
length	3	3	3	3
lookup	0	0	0	1⋆
member	0	0	1⋆	1
member-check	0	0	0	1⋆
merge	0	0	0	1⋆
min-arith	0	0	1⋆	2⋆
minimum	4	4	4	4
nonmember	1	1	1	1
pension	4	4	4	4
perm-sort	1	1	1	1
polynomials	4	4	4	4
qsort-dl	0	0	0	5⋆
queens	3	3	7⋆	11⋆
quicksort	1	1	1	5⋆
range	1	1	1	1
reverse	3	4⋆	4	4
rotate-right	10	12⋆	12	12
satisfiable	0	0	0	0
select-first	0	0	0	1⋆
sieve	0	0	6⋆	6
split	0	0	1⋆	1⋆
sublists	4	4	4	4
substitute	0	0	0	2⋆
transpose	3	3	3	3
traversal	9	9	9	9
traverse	2	2	2	2
tree-member	0	0	0	0
unify-oc	0	0	0	10⋆
word-list	3	3	3	4⋆

Figure 1: Relative precision experiments: **(A)** using [16]; **(B)** using suspension inference; **(C)** adding cut logic to (B); and **(D)** adding decorated depth-k to (C)

another that realises suspension inference [6]. The third column refines this analysis by considering cut and the four column additionally applies a depth-k analysis that tracks constraints. The starred entries indicate that either more modes have been inferred or that modes have been refined. The `qsort` example illustrates the cascading effect of precision loss: if predicate p depends on q and no mode can be inferred for q, then no mode can be deduced for p. Large programs magnify this effect. A number of anomalies, i.e. subtle non-deterministic behaviours, were discovered during evaluation. For example, the `exp(N,X,Y)` predicate [20, Figure 3.5] which realises the function that binds `Y` to `X` raised to the power `N`, is non-deterministic for $\texttt{N} > 0$ and $\texttt{X} = 0$.

6 Related work

Dawson *et al.* [4] extract determinacy information from a logic program by applying a transformation that resembles magic. The chief novelty in their work is a transformation that simultaneously describes both success pattern constraints and constraints from the calling context. These constraints are then added to each clause without compromising the correctness of the program, so as to reduce backtracking. The authors state that "if the clause conditions of a predicate are pairwise non-unifiable, we infer that the predicate is determinate whenever the input arguments are sufficiently ground". However, to assure determinacy, it is also necessary to ensure that any calls invoked within the body of a clause are themselves deterministic. Ensuring this property leads onto the consideration of various computation rules, and the topic of this paper.

Different computation rules, such as leftmost and rightmost selection, have been considered before as a way of improving the precision of success pattern analysis when the abstract unification operation is not commutative [5]. The need for considering various computation rules that arises in determinacy inference, stems not from imprecision in the domain, but from the way bindings made by a rightmost sub-goal can enforce the determinacy of a leftmost sub-goal.

Goal-dependent analysis can be used to ensure that each sub-goal of a given goal cannot succeed with more than one clause of a predicate [15]. The key step is to detect whether two different clauses for a predicate are mutually exclusive with respect to the calling patterns of the predicate. It is important to observe that this approach requires each sub-goal of a compound goal to be determinate irrespective of any bindings that are posted by later sub-goals (much like the original version of determinacy inference [16]).

A wider review of various forms of determinacy analysis can be found in [8] and the references therein. Work on determinacy checking (rather determinacy inference) that is of particular note is that given by Braem *et al.* [1] who present an analysis that given a calling mode for a predicate, infers bounds on the number of solutions that can be produced by a call in a given mode. In the context of partial evaluation, Sahlin [18] presents a determinacy analysis that can detect whether if a given goal fail, succeeds once, twice or more times, or whether it possibly loops. Mogensen [17] provides a semantically justified reconstruction of the work of Sahlin [18] based on a denotational semantics for Prolog programs with cut. Quite independently, Le Charlier *et al.* [12] developed a denotational sequence-based abstract interpretation framework for Prolog that can, among other things, be instantiated to obtain Sahlin's [18] determinacy analysis. Interestingly, in partial evaluation, delay declarations are sometimes used to postpone the unfolding of goals until they become sufficiently deterministic [13] which hints at the transformation at the heart of this paper. Finally, it should be noted that determinacy analysis also arises in context of functional logic programming [2], though there, non-determinacy is inextricably linked with laziness.

7 Conclusions

This paper has shown how determinacy inference can be improved by transforming the problem to an analysis problem in concurrency. The paper shows that this approach is flexible enough to handle the cut and accurate enough to locate non-determinacy problems in existing programs.

Acknowledgments We thank John Gallagher, Manual Hermenegildo, Michael Leuschel and Fred Mesnard for discussions on determinacy inference. This work was funded, in part, by NSF grants CCR-0131862 and INT-0327760, the EPSRC grant EP/C015517 and a Royal Society Incoming Short Visit grant.

References

[1] C. Braem, B. Le Charlier, S. Modar, and P. Van Hentenryck. Cardinality Analysis of Prolog. In M. Bruynooghe, editor, *International*

Symposium on Logic Programming, pages 457–471. MIT Press, 1994.

[2] B. Braßel and M. Hanus. Nondeterminacy Analysis of Functional Logic Programs. In M. Gabbrielli and G. Gupta, editors, *International Conference on Logic Programming*, volume 3668 of *LNCS*, pages 265–279. Springer-Verlag, 2005.

[3] M. Bruynooghe, M. Codish, J. Gallagher, S. Genaim, and W. Vanhoof. Termination Analysis through Combination of Type Based Norms. *ACM Transactions on Programming Languages and Systems*, To appear.

[4] S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar. Extracting Determinacy in Logic Programs. In *International Conference on Logic Programming*, pages 424–438. MIT Press, 1993.

[5] J. Gallagher and D. A. de Waal. Fast and Precision Regular Approximations of Logic Programs. In P. Van Hentenryck, editor, *International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.

[6] S. Genaim and A. King. Goal-independent Suspension Analysis for Logic Programs with Dynamic Scheduling. In P. Degano, editor, *European Symposium on Programming*, volume 2618 of *LNCS*, pages 84–98. Springer-Verlag, 2003.

[7] L. Henkin, J. D. Monk, and A. Tarski. *Cylindric Algebras*. North Holland, 1971.

[8] P. Hill and A. King. Determinacy and Determinacy Analysis. *Journal of Programming Languages*, 5(1):135–171, 1997.

[9] P. Hill and J. Lloyd. *The Gödel Programming Language*. MIT Press, 1996.

[10] A. King and L. Lu. A Backward Analysis for Constraint Logic Programs. *Theory and Practice of Logic Programming*, 2:517–547, 2002.

[11] V. Lagoon and P. J. Stuckey. A Framework for Analysis of Typed Logic Programs. In *Functional and Logic Programming*, volume 2024 of *LNCS*, pages 296–310. Springer-Verlag, 2001.

[12] B. Le Charlier, S. Rossi, and P. Van Hentenryck. Sequence-based abstract interpretation of Prolog. *Theory and Practice of Logic Programming*, 2:25–84, 2002.

[13] M. Leuschel. Personal Communication on Partial Evaluation, April 2005.

[14] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[15] P. López-García, F. Bueno, and M. Hermenegildo. Determinacy Analysis for Logic Programs Using Mode and Type information. In *Logic-Based Program Synthesis and Transformation*, volume 3573 of *LNCS*, pages 19–35. Springer-Verlag, 2005.

[16] L. Lu and A. King. Determinacy Inference for Logic Programs. In M. Sagiv, editor, *European Symposium on Programming*, volume 3444 of *LNCS*, pages 108–123. Springer-Verlag, 2005.

[17] T. Mogensen. A Semantics-Based Determinacy Analysis for Prolog with Cut. In *Ershov Memorial Conference*, volume 1181 of *LNCS*, pages 374–385. Springer-Verlag, 1996.

[18] D. Sahlin. Determinacy Analysis for Full Prolog. In *Partial Evaluation and Semantics Based Program Manipulation*, pages 23–30, 1991. SIGPLAN Notices 26(9).

[19] V. A. Saraswat, M. C. Rinard, and P. Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *Principles of Programming Languages*, pages 333–352. ACM Press, 1991.

[20] E. Shapiro and L. Sterling. *The Art of Prolog*. MIT Press, 1994.

[21] W. Vanhoof, R. Tronçon, and M. Bruynooghe. A Fixed Point Semantics for Logic Programs Extended with Cuts. In *Logic-Based Program Synthesis and Transformation*, volume 2664 of *LNCS*, pages 238–257, 2002.

A Proof appendix (included for reviewing)

Proof A.1 (for proposition 3.1)

- Suppose $\mathsf{var}(\vec{x}) \cap \mathsf{var}(\vec{y}) = \emptyset$. Then

  - Immediate.
  - Now $\rho_{\vec{x},\vec{y}}([\theta_1]_\equiv) \wedge \rho_{\vec{x},\vec{y}}([\theta_2]_\equiv) =$
    $\overline{\exists}_{\vec{y}}([d_{\vec{x},\vec{y}}]_\equiv \wedge \overline{\exists}_{\vec{x}}([\theta_1]_\equiv)) \wedge \overline{\exists}_{\vec{y}}([d_{\vec{x},\vec{y}}]_\equiv \wedge \overline{\exists}_{\vec{x}}([\theta_2]_\equiv)) =$
    $\overline{\exists}_{\vec{y}}([d_{\vec{x},\vec{y}}]_\equiv \wedge \overline{\exists}_{\vec{x}}([\theta_1]_\equiv) \wedge \overline{\exists}_{\vec{y}}([d_{\vec{x},\vec{y}}]_\equiv \wedge \overline{\exists}_{\vec{x}}([\theta_2]_\equiv))) \models$
    $\overline{\exists}_{\vec{y}}([d_{\vec{x},\vec{y}}]_\equiv \wedge \overline{\exists}_{\vec{x}}([\theta_1]_\equiv) \wedge \overline{\exists}_{\vec{x}}([\theta_2]_\equiv)) =$
    $\overline{\exists}_{\vec{y}}([d_{\vec{x},\vec{y}}]_\equiv \wedge \overline{\exists}_{\vec{x}}(\overline{\exists}_{\vec{x}}([\theta_1]_\equiv) \wedge [\theta_2]_\equiv)) =$
    $\rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}([\theta_1]_\equiv) \wedge [\theta_2]_\equiv)$ as required.
  - Immediate.
  - Now $\rho_{\vec{x},\vec{y}}(\rho_{\vec{y},\vec{x}}([\theta]_\equiv)) =$
    $\overline{\exists}_{\vec{y}}([d_{\vec{x},\vec{y}}]_\equiv \wedge \overline{\exists}_{\vec{x}}(\overline{\exists}_{\vec{x}}([d_{\vec{y},\vec{x}}]_\equiv \wedge \overline{\exists}_{\vec{y}}([\theta]_\equiv)))) =$
    $\overline{\exists}_{\vec{y}}([d_{\vec{x},\vec{y}}]_\equiv \wedge \overline{\exists}_{\vec{x}}([d_{\vec{y},\vec{x}}]_\equiv \wedge \overline{\exists}_{\vec{y}}([\theta]_\equiv))) =$
    $\overline{\exists}_{\vec{y}}([d_{\vec{y},\vec{x}}]_\equiv \wedge \overline{\exists}_{\vec{x}}([d_{\vec{y},\vec{x}}]_\equiv \wedge \overline{\exists}_{\vec{y}}([\theta]_\equiv))) = \overline{\exists}_{\vec{y}}([\theta]_\equiv)$

- Suppose $\mathsf{var}(\vec{x}) \cap \mathsf{var}(\vec{y}) \neq \emptyset$. Then

  - Immediate.
  - Now $\rho_{\vec{z},\vec{y}}(\rho_{\vec{x},\vec{z}}([\theta_1]_\equiv)) \wedge \rho_{\vec{z},\vec{y}}(\rho_{\vec{x},\vec{z}}([\theta_2]_\equiv)) \models$
    $\rho_{\vec{z},\vec{y}}(\overline{\exists}_{\vec{z}}(\rho_{\vec{x},\vec{z}}([\theta_1]_\equiv)) \wedge \rho_{\vec{x},\vec{z}}([\theta_2]_\equiv)) \models$
    $\rho_{\vec{z},\vec{y}}(\rho_{\vec{x},\vec{z}}([\theta_1]_\equiv) \wedge \rho_{\vec{x},\vec{z}}([\theta_2]_\equiv)) \models$
    $\rho_{\vec{z},\vec{y}}(\rho_{\vec{x},\vec{z}}(\overline{\exists}_{\vec{x}}([\theta_1]_\equiv) \wedge [\theta_2]_\equiv)) =$
    $\rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}([\theta_1]_\equiv) \wedge [\theta_2]_\equiv)$
  - Immediate.
  - Now $\rho_{\vec{z},\vec{y}}(\rho_{\vec{x},\vec{z}}(\rho_{\vec{z},\vec{x}}(\rho_{\vec{y},\vec{z}}([\theta]_\equiv)))) =$
    $\rho_{\vec{z},\vec{y}}(\overline{\exists}_{\vec{z}}(\rho_{\vec{y},\vec{z}}([\theta]_\equiv))) = \rho_{\vec{z},\vec{y}}(\rho_{\vec{y},\vec{z}}([\theta]_\equiv)) = \overline{\exists}_{\vec{y}}([\theta]_\equiv)$

Proof A.2 (for proposition 3.2) Suppose $\sigma \in (\downarrow\Theta_1) \cap (\downarrow\Theta_2)$. Since $\sigma \in \downarrow\Theta_i$ for both $i = 1, 2$ then, by axioms 1, $[\sigma]_\equiv = [\sigma]_\equiv \wedge [\sigma]_\equiv$, hence $\sigma \in \{\theta \mid [\theta]_\equiv = [\theta_1]_\equiv \wedge [\theta_2]_\equiv \wedge \theta_1 \in\downarrow\Theta_1 \wedge \theta_2 \in\downarrow\Theta_2\}$. Now suppose there exists $\theta_i \in\downarrow\Theta_i$ for both $i = 1, 2$ such that $[\sigma]_\equiv = [\theta_1]_\equiv \wedge [\theta_2]_\equiv$. By axioms 1 $[\sigma]_\equiv \models [\theta_i]_\equiv$ for both $i = 1, 2$, hence $\sigma \in\downarrow\Theta_i$ for both $i = 1, 2$ and the result follows.

Proof A.3 (for proposition 3.3)

- To see that $\overline{\exists}_{\vec{x}}(\downarrow\Theta) \subseteq \overline{\exists}_{\vec{x}}(\Theta)$, let $\theta_1 \in \overline{\exists}_{\vec{x}}(\downarrow\Theta)$. Then there exists $\theta_2 \in\downarrow\Theta$ such that $[\theta_1]_\equiv \models \overline{\exists}_{\vec{x}}([\theta_2]_\equiv)$. Hence there exists $\theta_3 \in \Theta$ such that $[\theta_2]_\equiv \models [\theta_3]_\equiv$. By axioms 2, $[\theta_1]_\equiv \models \overline{\exists}_{\vec{x}}([\theta_2]_\equiv) \models \overline{\exists}_{\vec{x}}([\theta_3]_\equiv)$ hence the result follows.

- To see that $\overline{\exists}_{\vec{x}}(\overline{\exists}_{\vec{x}}(\Theta)) \subseteq \overline{\exists}_{\vec{x}}(\Theta)$ let $\sigma \in \overline{\exists}_{\vec{x}}(\overline{\exists}_{\vec{x}}(\Theta))$. There exists $\theta_1 \in \overline{\exists}_{\vec{x}}(\Theta)$ such that $[\sigma]_\equiv \models \overline{\exists}_{\vec{x}}([\theta_1]_\equiv)$. Moreover there exists $\theta_2 \in \Theta$ such that $[\theta_1]_\equiv \models \overline{\exists}_{\vec{x}}([\theta_2]_\equiv)$. By axioms 2, $[\sigma]_\equiv \models \overline{\exists}_{\vec{x}}(\overline{\exists}_{\vec{x}}([\theta_1]_\equiv)) = \overline{\exists}_{\vec{x}}([\theta_1]_\equiv)$. Thus $\sigma \in \overline{\exists}_{\vec{x}}(\Theta)$ as required.

- Now $\overline{\exists}_{\vec{x}}(\Theta_1) \cap \overline{\exists}_{\vec{x}}(\Theta_2) = \overline{\exists}_{\vec{x}}(\downarrow\Theta_1) \cap \overline{\exists}_{\vec{x}}(\downarrow\Theta_2) =$
$\{\sigma \in \mathsf{Con} \mid \theta_1 \in\downarrow\Theta_1 \wedge \theta_2 \in\downarrow\Theta_2 \wedge [\sigma]_\equiv \models \overline{\exists}_{\vec{x}}([\theta_1]_\equiv) \wedge [\sigma]_\equiv \models \overline{\exists}_{\vec{x}}([\theta_2]_\equiv)\}$
$=$
$\{\sigma \in \mathsf{Con} \mid \theta_1 \in\downarrow\Theta_1 \wedge \theta_2 \in\downarrow\Theta_2 \wedge [\sigma]_\equiv \models \overline{\exists}_{\vec{x}}([\theta_1]_\equiv) \wedge \overline{\exists}_{\vec{x}}([\theta_2]_\equiv)\} =$
$\{\sigma \in \mathsf{Con} \mid \theta_1 \in\downarrow\Theta_1 \wedge \theta_2 \in\downarrow\Theta_2 \wedge [\sigma]_\equiv \models \overline{\exists}_{\vec{x}}([\theta_1]_\equiv \wedge \overline{\exists}_{\vec{x}}([\theta_2]_\equiv))\} =$
$\overline{\exists}_{\vec{x}}(\{\sigma \mid \theta_i \in\downarrow\Theta_i \wedge [\sigma]_\equiv \models [\theta_1]_\equiv \wedge \overline{\exists}_{\vec{x}}([\theta_2]_\equiv)\}) =$
$\overline{\exists}_{\vec{x}}(\downarrow\Theta_1 \cap \overline{\exists}_{\vec{x}}(\downarrow\Theta_2)) = \overline{\exists}_{\vec{x}}(\downarrow\Theta_1 \cap \overline{\exists}_{\vec{x}}(\Theta_2))$ as required.

Proof A.4 (for proposition 3.4)

- To see that $\rho_{\vec{x},\vec{y}}(\downarrow\Theta) \subseteq \rho_{\vec{x},\vec{y}}(\Theta)$, let $\theta_1 \in \rho_{\vec{x},\vec{y}}(\downarrow\Theta)$. Then there exists $\theta_2 \in\downarrow\Theta$ such that $[\theta_1]_\equiv \models \rho_{\vec{x},\vec{y}}([\theta_2]_\equiv)$. Hence there exists $\theta_3 \in \Theta$ such that $[\theta_2]_\equiv \models [\theta_3]_\equiv$. By axioms 3.1, $[\theta_1]_\equiv \models \rho_{\vec{x},\vec{y}}([\theta_2]_\equiv) \models \rho_{\vec{x},\vec{y}}([\theta_3]_\equiv)$ hence $\rho_{\vec{x},\vec{y}}(\downarrow\Theta) \subseteq \rho_{\vec{x},\vec{y}}(\Theta)$ as desired.

  Now let $\theta \in \mathsf{Con}$ and observe that by axiom 1 and proposition 3.1,
  $\rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}([\theta]_\equiv)) = \rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}([\theta]_\equiv) \wedge \overline{\exists}_{\vec{x}}([\theta]_\equiv))$
  $= \rho_{\vec{x},\vec{y}}([\theta]_\equiv) \wedge \rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}([\theta]_\equiv)) \models \rho_{\vec{x},\vec{y}}([\theta]_\equiv)$. Also by axiom 2 and proposition 3.1, $\rho_{\vec{x},\vec{y}}([\theta]_\equiv) \models \rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}([\theta]_\equiv))$, and therefore $\rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}([\theta]_\equiv)) = \rho_{\vec{x},\vec{y}}([\theta]_\equiv)$.

- To see that $\rho_{\vec{x},\vec{y}}(\Theta_j) \subseteq \cup_{i=1}^n \rho_{\vec{x},\vec{y}}(\Theta_i)$, let $\sigma \in \rho_{\vec{x},\vec{y}}(\cup_{i=1}^n \Theta_i)$. Then there exists $\kappa \in \cup_{i=1}^n \Theta_i$ such that $[\sigma]_\equiv \models \rho_{\vec{x},\vec{y}}([\kappa]_\equiv)$. Hence there exists $1 \leq j \leq n$ such that $\kappa \in \Theta_j$, whence $\sigma \in \rho_{\vec{x},\vec{y}}(\Theta_j) \subseteq \cup_{i=1}^n \rho_{\vec{x},\vec{y}}(\Theta_i)$ as required.

- To see that $\rho_{\vec{x},\vec{y}}(\rho_{\vec{y},\vec{x}}(\Theta)) \subseteq \overline{\exists}_{\vec{y}}(\Theta)$, let $\theta_1 \in \rho_{\vec{x},\vec{y}}(\rho_{\vec{y},\vec{x}}(\Theta))$. Then there exists $\theta_2 \in \rho_{\vec{y},\vec{x}}(\Theta)$ such that $[\theta_1]_\equiv \models \rho_{\vec{x},\vec{y}}([\theta_2]_\equiv)$. Hence there exists $\theta_3 \in \Theta$ such that $[\theta_2]_\equiv \models \rho_{\vec{y},\vec{x}}([\theta_3]_\equiv)$. By axioms 3.1, $\theta_1 \models \rho_{\vec{x},\vec{y}}([\theta_2]_\equiv) \models \rho_{\vec{x},\vec{y}}(\rho_{\vec{y},\vec{x}}([\theta_3]_\equiv)) = \overline{\exists}_{\vec{y}}([\theta_3]_\equiv)$ as required. The other direction follows similarly.

- Let $\sigma \in \rho_{\vec{x},\vec{y}}(\Theta_1) \cap \rho_{\vec{x},\vec{y}}(\Theta_2)$. Then there exists $\theta_i \in \Theta_i$ for both $i = 1, 2$ such that $[\sigma]_\equiv \models \rho_{\vec{x},\vec{y}}([\theta_i]_\equiv)$. By axiom 1 and proposition 3.1, $[\sigma]_\equiv \models \rho_{\vec{x},\vec{y}}([\theta_1]_\equiv) \wedge \rho_{\vec{x},\vec{y}}([\theta_2]_\equiv) = \rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}([\theta_1]_\equiv) \wedge [\theta_2]_\equiv)$. Thus $\sigma \in \rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}(\Theta_1) \cap \Theta_2)$. The other direction, note $\rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}(\Theta_1) \cap \Theta_2) \subseteq \rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}(\Theta_1)) = \rho_{\vec{x},\vec{y}}(\Theta_1)$ and $\rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}(\Theta_1) \cap \Theta_2) \subseteq \rho_{\vec{x},\vec{y}}(\Theta_2)$, hence $\rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}(\Theta_1) \cap \Theta_2) \subseteq \rho_{\vec{x},\vec{y}}(\Theta_1) \cap \rho_{\vec{x},\vec{y}}(\Theta_2)$ as required.

- Now $\rho_{\vec{x},\vec{y}}(\rho_{\vec{y},\vec{x}}(\Theta_1) \cap \downarrow\Theta_2) =$
$\rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}(\overline{\exists}_{\vec{x}}(\rho_{\vec{y},\vec{x}}(\Theta_1)) \cap \downarrow\Theta_2)) =$
$\rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}(\rho_{\vec{y},\vec{x}}(\Theta_1)) \cap \overline{\exists}_{\vec{x}}(\downarrow\Theta_2)) =$
$\rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}(\rho_{\vec{y},\vec{x}}(\Theta_1))) \cap \rho_{\vec{x},\vec{y}}(\overline{\exists}_{\vec{x}}(\downarrow\Theta_2)) =$
$\rho_{\vec{x},\vec{y}}(\rho_{\vec{y},\vec{x}}(\Theta_1)) \cap \rho_{\vec{x},\vec{y}}(\Theta_2) = \overline{\exists}_{\vec{y}}(\Theta_1) \cap \rho_{\vec{x},\vec{y}}(\Theta_2)$

- Suppose $\Theta_1 \cap \rho_{\vec{x},\vec{y}}(\Theta_2) \neq \emptyset$. Since $\Theta_1 \subseteq \overline{\exists}_{\vec{x}}(\Theta_1)$, it follows that $\overline{\exists}_{\vec{x}}(\Theta_1) \cap \rho_{\vec{x},\vec{y}}(\Theta_2) \neq \emptyset$. Because $\overline{\exists}_{\vec{x}}(\Theta_1) \cap \rho_{\vec{x},\vec{y}}(\Theta_2) = \rho_{\vec{x},\vec{y}}(\rho_{\vec{y},\vec{x}}(\Theta_1) \cap \downarrow\Theta_2$, then $\rho_{\vec{x},\vec{y}}(\rho_{\vec{y},\vec{x}}(\Theta_1) \cap \downarrow\Theta_2) \neq \emptyset$. Therefore $\emptyset \neq \rho_{\vec{y},\vec{x}}(\Theta_1) \cap \downarrow\Theta_2 = \rho_{\vec{y},\vec{x}}(\Theta_1) \cap \Theta_2$ as required. The other direction is analogous.

Proof A.5 (for lemma 3.1) Proof by structural induction on $A \in \mathsf{Age}_P$.

- Suppose $A = \mathsf{tell}(\Sigma)$. Now $\mathcal{M}_P\llbracket A \rrbracket(\Omega \cap \Theta) = \mathsf{if}\ \Sigma \cap \Omega \cap \Theta = \emptyset\ \mathsf{then}\ \wr\int\ \mathsf{else}\ \wr\Sigma \cap \Omega \cap \Theta\int = \wr\Pi \cap \Omega \mid \Pi \widehat{\in} \mathcal{M}_P\llbracket A \rrbracket(\Theta) \wedge \Pi \cap \Omega \neq \emptyset\int$ as required.

- Suppose $A = A_1, A_2$. By definition $\mathcal{M}_P\llbracket A_1, A_2 \rrbracket(\Omega \cap \Theta) = \widehat{\cup}\wr\mathcal{M}_P\llbracket A_2 \rrbracket(\Sigma) \mid \Sigma \widehat{\in} \mathcal{M}_P\llbracket A_1 \rrbracket(\Omega \cap \Theta)\int$ and by induction $\mathcal{M}_P\llbracket A_1, A_2 \rrbracket(\Omega \cap \Theta) = \widehat{\cup}\wr\mathcal{M}_P\llbracket A_2 \rrbracket(\Omega \cap \Sigma) \mid \Sigma \widehat{\in} \mathcal{M}_P\llbracket A_1 \rrbracket(\Theta) \wedge \Omega \cap \Sigma \neq \emptyset\int$. Also by induction, $\mathcal{M}_P\llbracket A_2 \rrbracket(\emptyset) = \mathcal{M}_P\llbracket A_2 \rrbracket(\emptyset \cap \Sigma) = \wr\int$, hence the $\Omega \cap \Sigma \neq \emptyset$ condition can be relaxed and $\mathcal{M}_P\llbracket A_1, A_2 \rrbracket(\Omega \cap \Theta) = \widehat{\cup}\wr\mathcal{M}_P\llbracket A_2 \rrbracket(\Omega \cap \Sigma) \mid \Sigma \widehat{\in} \mathcal{M}_P\llbracket A_1 \rrbracket(\Theta)\int$. By induction again, $\mathcal{M}_P\llbracket A_2 \rrbracket(\Omega \cap \Sigma) = \wr\Pi \cap \Omega \mid \Pi \widehat{\in} \mathcal{M}_P\llbracket A_2 \rrbracket(\Sigma) \wedge \Pi \cap \Omega \neq \emptyset\int$ therefore $\mathcal{M}_P\llbracket A_1, A_2 \rrbracket(\Omega \cap \Theta) = \widehat{\cup}\wr\Pi \cap \Omega \mid \Pi \widehat{\in} \mathcal{M}_P\llbracket A_2 \rrbracket(\Sigma) \wedge \Pi \cap \Omega \neq \emptyset \wedge \Sigma \widehat{\in} \mathcal{M}_P\llbracket A_1 \rrbracket(\Theta)\int = \widehat{\cup}\wr\Pi \cap \Omega \mid \Pi \widehat{\in} \mathcal{M}_P\llbracket A \rrbracket(\Theta) \wedge \Pi \cap \Omega \neq \emptyset\int$ as required.

- Suppose $A = \sum_{i=1}^{n} A_i$. Then $\mathcal{M}_P\llbracket \sum_{i=1}^{n} A_i \rrbracket(\Omega \cap \Theta) =$
$\widehat{\cup}\wr\mathcal{M}_P\llbracket A_i \rrbracket(\Omega \cap \Theta)\int_{i=1}^{n} =$
$\wr\Sigma \cap \Omega \mid \Sigma \widehat{\in} \mathcal{M}_P\llbracket A_i \rrbracket(\Theta) \wedge \Sigma \cap \Omega \neq \emptyset \wedge 1 \leq i \leq n\int = \wr\Sigma \cap \Omega \mid \Sigma \widehat{\in} \mathcal{M}_P\llbracket \sum_{i=1}^{n} A_i \rrbracket(\Theta) \wedge \Sigma \cap \Omega \neq \emptyset\int$ as required.

- Suppose $A = p(\vec{x})$ and $p(\vec{y}) :\!-\ B \in P$. By induction
$\wr\Theta \cap \rho_{\vec{y},\vec{x}}(\Sigma) \mid \Sigma \widehat{\in} \mathcal{M}_P\llbracket B \rrbracket(\rho_{\vec{x},\vec{y}}(\Theta))\int =$
$\wr\Theta \cap \rho_{\vec{y},\vec{x}}(\rho_{\vec{x},\vec{y}}(\Theta) \cap \Pi) \mid \rho_{\vec{x},\vec{y}}(\Theta) \cap \Pi \neq \emptyset \wedge \Pi \widehat{\in} \mathcal{M}_P\llbracket B \rrbracket(\mathsf{Con})\int$. Now by proposition 3.4, $\rho_{\vec{x},\vec{y}}(\Theta) \cap \Pi \neq \emptyset$ iff $\Theta \cap \rho_{\vec{y},\vec{x}}(\Pi) \neq \emptyset$. Moreover by

proposition 3.4, since $\downarrow\Pi = \Pi$, $\rho_{\vec{y},\vec{x}}(\rho_{\vec{x},\vec{y}}(\Theta) \cap \Pi) = \overline{\exists}_{\vec{x}}(\Theta) \cap \rho_{\vec{y},\vec{x}}(\Pi)$, hence $\Theta \cap \rho_{\vec{y},\vec{x}}(\rho_{\vec{x},\vec{y}}(\Theta) \cap \Pi) = \Theta \cap \rho_{\vec{y},\vec{x}}(\Pi)$ by proposition 3.3, whence $\mathcal{M}_P[\![A]\!](\Theta) =$ $\wr \Theta \cap \rho_{\vec{y},\vec{x}}(\Pi) \mid \Theta \cap \rho_{\vec{y},\vec{x}}(\Pi) \neq \emptyset \wedge \Pi \widehat{\in} \mathcal{M}_P[\![B]\!](\mathsf{Con}) \int$. The result then follows.

Proof A.6 (for corollary 3.1)

- Observe $\wr \Sigma \mid \Sigma \widehat{\in} \mathcal{M}_P[\![A]\!](\Theta) \int = \mathcal{M}_P[\![A]\!](\Theta \cap \Theta) = \wr \Sigma \cap \Theta \mid \Sigma \widehat{\in} \mathcal{M}_P[\![A]\!](\Theta) \wedge \Sigma \cap \Theta \neq \emptyset \int$. Hence $\Psi \subseteq \Theta$ for all $\Psi \widehat{\in} \mathcal{M}_P[\![A]\!](\Theta)$ and the result follows.
- Now $\mathcal{M}_P[\![A]\!](\Omega \cap \Theta) =$ $\wr \Sigma \cap \Omega \mid \Sigma \widehat{\in} \mathcal{M}_P[\![A]\!](\Theta) \wedge \Sigma \cap \Omega \neq \emptyset \int$. Since $\Sigma \subseteq \Omega$ and $\Sigma \cap \Omega \neq \emptyset$ for all $\Sigma \widehat{\in} \mathcal{M}_P[\![A]\!](\Theta)$, it follows that $\wr \Sigma \cap \Omega \mid \Sigma \widehat{\in} \mathcal{M}_P[\![A]\!](\Theta) \wedge \Sigma \cap \Omega \neq \emptyset \int$ $=$ $\wr \Sigma \mid \Sigma \widehat{\in} \mathcal{M}_P[\![A]\!](\Theta) \int = \mathcal{M}_P[\![A]\!](\Theta)$ as required.
- Observe that $\mathcal{M}_P[\![A]\!](\Theta_1) = \mathcal{M}_P[\![A]\!](\Theta_1 \cap \Theta_2) =$ $\wr \Sigma \cap \Theta_1 \mid \Sigma \widehat{\in} \mathcal{M}_P[\![A]\!](\Theta_2) \wedge \Sigma \cap \Theta_1 \neq \emptyset \int$, hence $|\mathcal{M}_P[\![A]\!](\Theta_1)| \leq |\mathcal{M}_P[\![A]\!](\Theta_2)|$ as required.
- Observe that $\cup \mathcal{M}_P[\![A]\!](\Theta_1) = \cup \mathcal{M}_P[\![A]\!](\Theta_1 \cap \Theta_2) =$ $\cup \wr \Sigma \cap \Theta_1 \mid \Sigma \widehat{\in} \mathcal{M}_P[\![A]\!](\Theta_2) \wedge \Sigma \cap \Theta_1 \neq \emptyset \int \subseteq$ $\cup \wr \Sigma \cap \Theta_1 \mid \Sigma \widehat{\in} \mathcal{M}_P[\![A]\!](\Theta_2) \int \subseteq$ $\cup \wr \Sigma \mid \Sigma \widehat{\in} \mathcal{M}_P[\![A]\!](\Theta_2) \int = \cup \mathcal{M}_P[\![A]\!](\Theta_2)$ as required.

Proof A.7 (for proposition 3.5) Proof by structural induction on $A \in \mathsf{Age}_P$.

- Suppose $A = \mathsf{tell}(\Sigma)$. Then $\cup \mathcal{M}_P[\![A]\!](\Theta) = \Theta \cap \Sigma = \Theta \cap \mathcal{S}_P[\![A]\!]$ as required.
- Suppose $A = A_1, A_2$. Then, by the inductive hypothesis,

  $$\begin{array}{rcl} \cup \mathcal{M}_P[\![A]\!](\Theta) & = & \cup \{ \cup \mathcal{M}_P[\![A_2]\!](\Sigma) \mid \Sigma \widehat{\in} \mathcal{M}_P[\![A_1]\!](\Theta) \} \\ & = & \cup \{ \mathcal{S}_P[\![A_2]\!] \cap \Sigma \mid \Sigma \widehat{\in} \mathcal{M}_P[\![A_1]\!](\Theta) \} \end{array}$$

  Moreover,

  $$\begin{array}{rcl} \cup \mathcal{M}_P[\![A]\!](\Theta) & = & \mathcal{S}_P[\![A_2]\!] \cap (\cup \{ \Sigma \mid \Sigma \widehat{\in} \mathcal{M}_P[\![A_1]\!](\Theta) \}) \\ & = & \mathcal{S}_P[\![A_2]\!] \cap (\cup \mathcal{M}_P[\![A_1]\!](\Theta)) \\ & = & \mathcal{S}_P[\![A_2]\!] \cap \mathcal{S}_P[\![A_1]\!] \cap \Theta \\ & = & \mathcal{S}_P[\![A]\!] \cap \Theta \end{array}$$

  again by the induction hypothesis.

- Suppose $A = \sum_{i=1}^{n} A_i$. Then, by the induction hypothesis,

$$\begin{aligned}\cup\mathcal{M}_P[\![A]\!](\Theta) &= \cup\widehat{\cup}\wr\mathcal{M}_P[\![A_i]\!](\Theta)\wr_{i=1}^{n} \\ &= \cup\{\cup\mathcal{M}_P[\![A_i]\!](\Theta)\}_{i=1}^{n} \\ &= \cup\{\mathcal{S}_P[\![A_i]\!] \cap \Theta\}_{i=1}^{n}\end{aligned}$$

  Hence $\cup\mathcal{M}_P[\![A]\!](\Theta) = (\cup\{\mathcal{S}_P[\![A_i]\!]\}_{i=1}^{n}) \cap \Theta = \mathcal{S}_P[\![A]\!] \cap \Theta$ as required.

- Suppose $A = p(\vec{x})$ and $p(\vec{y}) :- B \in P$. Then $\cup\mathcal{M}_P[\![A]\!](\Theta) = \cup\wr\Theta \cap \rho_{\vec{y},\vec{x}}(\Sigma) \mid \Sigma\widehat{\in}\mathcal{M}_P[\![A]\!](\rho_{\vec{x},\vec{y}}(\Theta))\wr = \Theta\cap(\cup\{\rho_{\vec{y},\vec{x}}(\Sigma) \mid \Sigma\widehat{\in}\mathcal{M}_P[\![A]\!](\rho_{\vec{x},\vec{y}}(\Theta))\})$. By proposition 3.1, $\cup\mathcal{M}_P[\![A]\!](\Theta) = \Theta \cap \rho_{\vec{y},\vec{x}}(\cup\mathcal{M}_P[\![A]\!](\rho_{\vec{x},\vec{y}}(\Theta))) = \Theta\cap\rho_{\vec{y},\vec{x}}(\mathcal{S}_P[\![A]\!]\cap\rho_{\vec{x},\vec{y}}(\Theta)) = \Theta\cap\rho_{\vec{y},\vec{x}}(\mathcal{S}_P[\![A]\!])\cap\overline{\exists}_{\vec{x}}(\Theta) = \Theta\cap\rho_{\vec{y},\vec{x}}(\mathcal{S}_P[\![A]\!])$ as required.

Proof A.8 (for lemma 3.2) Proof by structural induction on $A \in \mathsf{Age}_P$. The first four cases are straightforward and so consider $A = p(\vec{x})$. Let $\langle\Theta, 0\rangle \in \mathcal{Q}_P[\![p(\vec{x})]\!]$ and $\Theta' \subseteq \Theta$ where $\Theta' \in \wp^{\downarrow}(\mathsf{Con})$. There exists $\langle\Sigma, 0\rangle \in \mathcal{Q}_P[\![A]\!]$ such that $\rho_{\vec{y},\vec{x}}(\Sigma) = \overline{\exists}_{\vec{x}}(\Theta)$ and $p(\vec{y}) :- A \in P$. Put $\Sigma' = \rho_{\vec{x},\vec{y}}(\Theta') \cap \Sigma \subseteq \Sigma$, whence $\langle\Sigma', 0\rangle \in \mathcal{Q}_P[\![A]\!]$ by induction. By proposition 3.4, $\rho_{\vec{y},\vec{x}}(\Sigma') = \rho_{\vec{y},\vec{x}}(\rho_{\vec{x},\vec{y}}(\Theta')\cap\Sigma) = \overline{\exists}_{\vec{x}}(\Theta')\cap\rho_{\vec{y},\vec{x}}(\Sigma) = \overline{\exists}_{\vec{x}}(\Theta')\cap\overline{\exists}_{\vec{x}}(\Theta) = \overline{\exists}_{\vec{x}}(\Theta')$ as required.

Proof A.9 (for theorem 3.1) Proof by structural induction on $A \in \mathsf{Age}_P$.

- Suppose $A = \mathsf{tell}(\Sigma)$.
  - Then $\cup\{\Theta \mid \langle\Theta, 0\rangle \in Q\} = \cup\{\Theta \mid \langle\Theta, 0\rangle \in \mathcal{Q}_{T[\![P]\!]}[\![\mathsf{tell}(\Omega), \mathsf{tell}(\Sigma)]\!]\}$ $= \{\Theta \mid \Theta \subseteq \Omega \wedge \Theta \subseteq \Sigma\} = \Omega \cap \Sigma = \Omega \cap \mathcal{S}_P[\![A]\!]$ as required.
  - Then $|\mathcal{M}_P[\![A]\!](\Omega)| \leq 1$ as required.

- Suppose $A = A_1, A_2$. Put, for $i = 1, 2$,

$$\begin{aligned}Q_i &= \mathcal{Q}_{T[\![P]\!]}[\![\mathsf{tell}(\Omega), T[\![A_i]\!]]\!] \\ \Gamma_i &= \{\Theta \mid \langle\Theta, 0\rangle \in Q_i\} \\ Q'_i &= \mathcal{Q}_{T[\![P]\!]}[\![\mathsf{tell}(\Omega \cap (\cup\Gamma_{3-i})), T[\![A_i]\!]]\!]\end{aligned}$$

  We now show that $\langle\Pi, 1\rangle \notin Q'_i$ for all $\Pi \in \wp^{\downarrow}(\mathsf{Con})$. Assume, for the sake of a contradiction, that $\langle\Pi, 1\rangle \in Q'_1$. Then there exists $\Theta \in \Gamma_2$ such that $\Pi \subseteq \Theta$. Thus $\langle\Theta, 0\rangle \in Q_2$ and by lemma 3.2, $\langle\Pi, 0\rangle \in Q_2$. Moreover, $Q'_1 \subseteq Q_1$, hence $\langle\Pi, 1\rangle \in Q_1$. Hence $\langle\Pi, 1\rangle \in Q$ which is a contradiction. Thus $\langle\Pi, 1\rangle \notin Q'_1$ and $\langle\Pi, 1\rangle \notin Q'_2$ for all $\Pi \in \wp^{\downarrow}(\mathsf{Con})$.

Now suppose $\langle\Psi,0\rangle \in Q_1\cap Q_2$. Then $\Psi\in\Gamma_1$ and $\langle\Psi,0\rangle\in Q_2$, whence $\langle\Psi,0\rangle \in Q_2'$. Similarly it follows that $\langle\Psi,0\rangle\in Q_1'$. But $Q_1'\cap Q_2' \subseteq Q_1\cap Q_2$, therefore $\cup\{\Psi \mid \langle\Psi,0\rangle\in Q_1\cap Q_2\} = \cup\{\Psi \mid \langle\Psi,0\rangle\in Q_1'\cap Q_2'\}$.

Let $\zeta \in (\cup\{\Psi_1 \mid \langle\Psi_1,0\rangle \in Q_1'\})\cap(\cup\{\Psi_2 \mid \langle\Psi_2,0\rangle\in Q_2'\})$. Then there exists $\langle\Psi_1,0\rangle\in Q_1'$ and $\langle\Psi_2,0\rangle\in Q_2'$ such that $\zeta\in\Psi_1\cap\Psi_2$. But by lemma 3.2, $\langle\Psi_1\cap\Psi_2,0\rangle\in Q_1'$ and $\langle\Psi_1\cap\Psi_2,0\rangle\in Q_2'$, whence $\zeta\in\cup\{\Psi\mid\langle\Psi,0\rangle\in Q_1'\cap Q_2'\}$. Thus $\cup\{\Psi\mid\langle\Psi,0\rangle\in Q_1'\cap Q_2'\} = (\cup\{\Psi_1\mid \langle\Psi_1,0\rangle\in Q_1'\})\cap(\cup\{\Psi_2\mid\langle\Psi_2,0\rangle\in Q_2'\})$.

- By induction $\cup\{\Psi \mid \langle\Psi,0\rangle\in Q_i'\} = \Omega\cap\mathcal{S}_P[\![A_i]\!]\cap(\cup\Gamma_{3-i})$. Therefore $\cup\{\Psi\mid\langle\Psi,0\rangle\in Q_1\cap Q_2\} = \Omega\cap\mathcal{S}_P[\![A_1]\!]\cap(\cup\Gamma_2)\cap\mathcal{S}_P[\![A_2]\!]\cap(\cup\Gamma_1)$. Since $Q_i'\subseteq Q_i$ it follows that $\mathcal{S}_P[\![A_i]\!]\subseteq(\cup\Gamma_i)$, hence $\cup\{\Psi\mid\langle\Psi,0\rangle\in Q_1\cap Q_2\} = \Omega\cap\mathcal{S}_P[\![A_1]\!]\cap\mathcal{S}_P[\![A_2]\!] = \Omega\cap\mathcal{S}_P[\![A_1,A_2]\!]$ as required.
- By induction $|\mathcal{M}_P[\![A_i]\!](\Omega\cap(\cup\Gamma_{3-i}))|\leq 1$. Now

  $$\mathcal{M}_P[\![A_1,A_2]\!](\Omega) = \widehat{\cup}\wr\mathcal{M}_P[\![A_2]\!](\Sigma)\mid\Sigma\widehat{\in}\mathcal{M}_P[\![A_1]\!](\Omega)\wr$$

  By corollary 3.1, $\cup\mathcal{M}_P[\![A_1]\!](\Omega)\subseteq\Omega$, hence $\Sigma=\Sigma\cap\Omega$. Moreover, by proposition 3.5, $\cup\mathcal{M}_P[\![A_2]\!](\Sigma) = \mathcal{S}_P[\![A_2]\!]\subseteq(\cup\Gamma_2)$ and by corollary 3.1, $\mathcal{M}_P[\![A_2]\!](\Sigma\cap(\cup\Gamma_2)) = \mathcal{M}_P[\![A_2]\!](\Sigma)$. Therefore

  $$\mathcal{M}_P[\![A_1,A_2]\!](\Omega) = \widehat{\cup}\wr\mathcal{M}_P[\![A_2]\!](\Sigma\cap\Omega\cap(\cup\Gamma_2))\mid\Sigma\widehat{\in}\mathcal{M}_P[\![A_1]\!](\Omega)\wr$$

  By lemma 3.1 and then using $\cup\mathcal{M}_P[\![A_1]\!](\Omega\cap(\cup\Gamma_2))\subseteq\cup\Gamma_1$ it follows that

  $$\mathcal{M}_P[\![A_1,A_2]\!](\Omega) = \widehat{\cup}\wr\mathcal{M}_P[\![A_2]\!](\Sigma\cap\Omega\cap(\cup\Gamma_1))\mid\Sigma\widehat{\in}\mathcal{M}_P[\![A_1]\!](\Omega\cap(\cup\Gamma_2))\wr$$

  By corollary 3.1, $|\mathcal{M}_P[\![A_2]\!](\Sigma\cap\Omega\cap(\cup\Gamma_1))| \leq |\mathcal{M}_P[\![A_2]\!](\Omega\cap(\cup\Gamma_1))|\leq 1$, hence $|\mathcal{M}_P[\![A_1,A_2]\!](\Omega)|\leq 1$ as required.

- Let $A=\sum_{i=1}^n A_i$ and $\Gamma = \mathsf{mux}(\wr\Theta_i\wr_{i=1}^n)$ where $\mathcal{S}_P[\![A_i]\!]\subseteq\Theta_i$.
  - Now $T[\![A]\!] = \mathsf{ask}(\Gamma)\to\sum_{i=1}^n T[\![A_i]\!]$ and

    $$\begin{aligned} Q &= \{\langle\Theta,1\rangle \mid \Theta\not\subseteq\Gamma\wedge\Theta\subseteq\Omega\} \;\cup \\ &\quad \{\langle\Theta,b\rangle\in\mathcal{Q}_{T[\![P]\!]}[\![\sum_{i=1}^n T[\![A_i]\!]]\!] \mid \Theta\subseteq\Gamma\wedge\Theta\subseteq\Omega\}\end{aligned}$$

Observe that if $\Omega \not\subseteq \Gamma$ then $\langle \Omega, 1 \rangle \in Q$ which is a contradiction. Thus, $\Omega \subseteq \Gamma$. Therefore,

$$\begin{array}{rcl} Q & = & \{\langle \Theta, b \rangle \in \mathcal{Q}_{T[\![P]\!]}[\![\sum_{i=1}^{n} T[\![A_i]\!]]\!] \mid \Theta \subseteq \Omega\} \\ & = & \cup_{i=1}^{n} \{\langle \Theta, b \rangle \in \mathcal{Q}_{T[\![P]\!]}[\![T[\![A_i]\!]]\!] \mid \Theta \subseteq \Omega\} \\ & = & \cup_{i=1}^{n} \mathcal{Q}_{T[\![P]\!]}[\![\mathsf{tell}(\Omega), T[\![A_i]\!]]\!] \end{array}$$

Since $\langle \Pi, 1 \rangle \notin Q$ for all $\Pi \in \wp^{\wr}(\mathsf{Con})$, $\langle \Pi, 1 \rangle \notin Q_i$ for all $\Pi \in \wp^{\wr}(\mathsf{Con})$ where $Q_i = \mathcal{Q}_{T[\![P]\!]}[\![\mathsf{tell}(\Omega), T[\![A_i]\!]]\!]$. By the induction hypothesis, $\cup\{\Theta \mid \langle \Theta, 0 \rangle \in Q_i\} = \Omega \cap \mathcal{S}_P[\![A_i]\!]$. Thus, $\cup\{\Theta \mid \langle \Theta, 0 \rangle \in Q\} = \cup(\cup_{i=1}^{n}(\Omega \cap \mathcal{S}_P[\![A_i]\!])) = \Omega \cap (\cup_{i=1}^{n} \mathcal{S}_P[\![A_i]\!]) = \Omega \cap \mathcal{S}_P[\![A]\!]$ as required.

– Since $\langle \Pi, 1 \rangle \notin \mathcal{Q}_P[\![\mathsf{tell}(\Sigma), T[\![A]\!]]\!]$ then $\langle \Pi, 1 \rangle \notin \mathcal{Q}_P[\![\mathsf{tell}(\Sigma), T[\![A_i]\!]]\!]$, hence by the inductive hypothesis, $|\mathcal{M}_P[\![A_i]\!](\Sigma)| \leq 1$. Suppose $\mathcal{M}_P[\![A_i]\!](\Sigma) = \wr \Pi_i \int$ and consider $j \neq i$. By proposition 3.5, $\emptyset \subset \Pi_i = \cup \mathcal{M}_P[\![A_i]\!](\Sigma) \subseteq \Sigma \cap \mathcal{S}_P[\![A_i]\!] \subseteq \Sigma \cap \Theta_i$. Since $\langle \Pi, 1 \rangle \notin \mathcal{Q}_P[\![\mathsf{tell}(\Sigma), T[\![A]\!]]\!]$ it follows that $\Sigma \subseteq \mathsf{mux}(\wr \Theta_i \int_{i=1}^{n})$ hence, as $\Sigma \cap \Theta_i \neq \emptyset$, then $\Sigma \cap \Theta_j = \emptyset$, then $\Sigma \cap \mathcal{S}_P[\![A_j]\!] = \emptyset$, whence by proposition 3.5, $\mathcal{M}_P[\![A_j]\!](\Sigma) = \wr \int$ and the result follows.

- Suppose $A = p(\vec{x})$. Let $p(\vec{y}) :- B \in P$ and let $\langle \Sigma, b \rangle \in \mathcal{Q}_{T[\![P]\!]}[\![T[\![B]\!]]\!]$. Then if $\rho_{\vec{y},\vec{x}}(\Sigma) \subseteq \Omega$ then $b = 0$. But if $\Sigma \subseteq \rho_{\vec{x},\vec{y}}(\Omega)$, then $\overline{\exists}(\Sigma) \subseteq \rho_{\vec{x},\vec{y}}(\Omega)$, hence $\rho_{\vec{y},\vec{x}}(\Sigma) \subseteq \overline{\exists}_{\vec{x}}(\Omega) \subseteq \Omega$. Thus if $\Sigma \subseteq \rho_{\vec{x},\vec{y}}(\Omega)$ then $b = 0$.

  – By the inductive hypothesis, $\cup\{\rho_{\vec{y},\vec{x}}(\Pi) \mid \langle \Pi, 0 \rangle \in \mathcal{Q}_{T[\![P]\!]}[\![T[\![B]\!]]\!] \wedge \Pi \subseteq \rho_{\vec{x},\vec{y}}(\Omega)\} = \rho_{\vec{y},\vec{x}}(\mathcal{Q}_{T[\![P]\!]}[\![\mathsf{tell}(\rho_{\vec{y},\vec{x}}(\Pi)), T[\![B]\!]]\!]\}) = \rho_{\vec{y},\vec{x}}(\rho_{\vec{x},\vec{y}}(\Omega) \cap \mathcal{S}_P[\![B]\!]) = \overline{\exists}_{\vec{x}}(\Omega) \cap \rho_{\vec{y},\vec{x}}(\mathcal{S}_P[\![B]\!]) = \overline{\exists}_{\vec{x}}(\Omega) \cap \mathcal{S}_P[\![p(\vec{x})]\!]$. The result then follows.

  – By the inductive hypothesis, $|\mathcal{M}_P[\![B]\!](\rho_{\vec{x},\vec{y}}(\Omega))| \leq 1$, and therefore
  $|\mathcal{M}_P[\![p(\vec{x})]\!](\Omega)| \leq 1$ as required.