Determinacy Analysis for Logic Programs Using Mode and Type Information

P. López-García[1], F. Bueno[1], and M. Hermenegildo[1,2]

[1] School of Computer Science, Technical University of Madrid (UPM)
[2] Depts. of Comp. Science and El. and Comp. Eng., U. of New Mexico (UNM)
{pedro.lopez,bueno,herme}@fi.upm.es

Abstract. We propose an analysis for detecting procedures and goals that are deterministic (i.e. that produce at most one solution), or predicates whose clause tests are mutually exclusive (which implies that at most one of their clauses will succeed) even if they are not deterministic (because they call other predicates that can produce more than one solution). Applications of such determinacy information include detecting programming errors, performing certain high-level program transformations for improving search efficiency, optimizing low level code generation and parallel execution, and estimating tighter upper bounds on the computational costs of goals and data sizes, which can be used for program debugging, resource consumption and granularity control, etc. We have implemented the analysis and integrated it in the *CiaoPP* system, which also infers automatically the mode and type information that our analysis takes as input. Experiments performed on this implementation show that the analysis is fairly accurate and efficient.

Keywords: Determinacy Inference, Program Analysis, Modes, Types.

1 Introduction

Knowing that certain predicates are deterministic for a given class of calls has a number of interesting applications in program debugging, verification, transformation, and optimization. By a predicate being deterministic we mean that it produces at most one solution. It is also interesting to detect predicates whose clause tests are mutually exclusive (which implies that at most one of their clauses will succeed) even if they are not deterministic because they call other predicates that can produce more than one solution.

Perhaps the most important application of compile-time determinacy information is in the context of program development. If we assume that the programmer has indicated that certain predicates should be deterministic for certain calling patterns (using suitable assertions as those used in Ciao [24], Mercury [27], or HAL [7]) and a predicate is determined to be non-deterministic in one of those cases then, clearly, a compile-time error has been detected and can be reported [14, 12]. This is quite useful since certain classes of programming errors often result in turning predicates intended to be deterministic into non-deterministic ones. Also, in addition to detecting programming errors at compile time, determinacy inference can obviously be used to *verify* (i.e., prove correct) such determinacy assertions [14].

Determinacy information can also be used for performing low-level optimizations [28, 21, 27] as well higher-level program transformations for improving search efficiency. In particular, literals can be reordered so that deterministic goals are executed ahead of possibly non-deterministic goals where possible, improving the efficiency of parallel search [26]. Determinacy information is also very useful during program specialization. In addition, the implementation of (and-)parallelism is greatly simplified in presence of determinacy information: knowing that a goal is deterministic allows one to eliminate significant run-time overhead (due to *markers*) [15, 11, 23] and, in addition, performing data parallelism transformations [13].

Finally, determinacy (and mutual exclusion) information can be used to estimate much tighter upper bounds on the computational costs of goals [5]. Since it is generally not known in advance how many of the solutions generated by a predicate will be demanded, a conservative upper bound on the computational cost of a predicate can be obtained by assuming that all solutions are needed, and that all clauses are executed (thus the cost of the predicate is assumed to be the sum of the costs of all of its clauses). It is straightforward to take mutual exclusion into account to obtain a more precise estimate of the cost of a predicate, using the maximum of the costs of mutually exclusive groups of clauses. Moreover, knowing that all literals in a clause will produce at most one solution allows one to assume that an upper bound on the cost of the clauses is the sum of the cost of all literals in it, which simplifies the cost estimation (as explained in [5]). These upper bounds can be used for improved granularity control of parallel tasks [20] and for better performance/complexity debugging and verification of programs [14].

In this paper we propose a method whereby, given (upper approximations of) mode and type information, we can detect procedures and goals that are deterministic (i.e., that produce at most one solution), or predicates whose clause tests are mutually exclusive, even if they are not deterministic because they call other predicates that can produce more than one solution (i.e. that are not deterministic).

There has been much interest on determinacy detection in the literature (see [16] and its references), using several different forms of determinism. The line of work closest to ours starts with [6], in which functional computations are detected and exploited. However, the notion of mutual exclusion in this work is not based on constraint satisfaction. This concept is used in the analysis presented in [4], where, nonetheless, no algorithms are defined for the detection of mutual exclusion. The cut is not taken into account, either. In [10] a combined analysis of modes, types, and determinacy is presented, as well as in the more accurate [2]. As we will show, our analysis improves on these proposals.

Several programming systems also make use of determinacy, e.g., Mercury [16] and HAL [7]. The Mercury and HAL systems allow the programmer to declare that a predicate will produce at most one solution, and attempts to verify this with respect to the Herbrand terms with equality tests. As far as we know, both systems use the same analysis [12], which does not handle disequa-

lity constraints on the Herbrand domain. Nor does it handle arithmetic tests, except in the context of the if-then-else construct. As such, it is considerably weaker than the approach described here. Also, our approach does not require any annotations from programmers, since the types and modes on which it is based are *inferred* (in our case by *CiaoPP* [14]).

2 Modes, Types, Tests, and Mutual Exclusion

We assume an acquaintance with the basic notions of logic programming. In order to reason about determinacy, it is necessary to distinguish between unification operations that act as tests (and which may fail), and output unifications that act as assignments (and always succeed). To this end, we assume that mode information is available, as a result of a previous analysis, i.e., for each unification operation in each predicate, we know whether the operation acts as a test or creates an output binding. Note that this is weaker than most conventional notions of moding in that it does not require input arguments to be completely ground, and allows an output argument to occur as a subterm of an input argument.

We also assume that type information is available, generally also as the result of a previous analysis. A type refers to a set of terms, and can be denoted by using several type representations (e.g. *type terms* and *regular term grammars* as in [3], or *type graphs* as in [17] or simply predicates as in the Ciao system). We include below the definitions of *type term*, *type rule*, and *deterministic type rule* from [3], for a better understanding of the algorithms that we have developed, and in order to make this paper more self-contained.

We assume the existence of an infinite set of *type symbols* (each type symbol refers to a set of terms, i.e., to a type). There are two special type symbols: μ, that represents the type of the entire Herbrand universe and the type symbol ϕ, that represents the empty type.

Definition 1. [Type term] A *type term* is defined inductively as follows:

1. A constant symbol is a type term.
2. A variable is a type term.
3. A type symbol is a type term.
4. If f is a n-ary function symbol, and each ω_i is a type term, then $f(\omega_1, \ldots, \omega_n)$ is a type term.

A *pure type term* is a variable-free type term. A *logical term* is a type-symbol-free type term. ■

In this paper, we refer to *logical terms* as *Herbrand terms*. Note that according to this definition, all type symbols are type terms, however, the converse is not true.

There is a distinguished non-empty finite subset of the set of type symbols called the set of *base type symbols*. The set of Herbrand terms represented by a base type symbol is called a *base type*. For example, the set of all constant

symbols that represent integer numbers is a base type represented by the base type symbol *integer*. We assume that there are effective tests for membership of a given Herbrand term in each base type.

Definition 2. [Type rule] A *type rule* is an expression of the form $\alpha \rightarrow \Upsilon$, where α is a type symbol, and Υ is a set of pure type terms. ■

Example 1. The type rule:

$$intlist \rightarrow \{[\,],\ [integer|intlist]\}$$

defines the type symbol *intlist*, that denotes the set of all lists of integer numbers. □

We denote sets of type rules, that is, *regular term grammars*, by the letter T (as in [3]).

Definition 3. A (non-base) type symbol α, is *defined* in, or by, a set of type rules T if there exists a type rule $(\alpha \rightarrow \Upsilon) \in T$. ■

Definition 4. A pure type term ω is *defined* by a set of type rules T if each type symbol in ω is either μ, ϕ, a base type symbol, or a (non-base) type symbol defined in T. ■

We assume that for each type rule $(\alpha \rightarrow \Upsilon) \in T$ it holds that each element (i.e. pure type term) of Υ is defined in T, and that each type symbol defined in T has *exactly* one defining type rule in T.

Definition 5. [Deterministic type rule] A type rule $\alpha \rightarrow \Upsilon$ is *deterministic* if no element of Υ is a type symbol and there is no pair of pure type terms $\omega_1, \omega_2 \in \Upsilon$, such that $\omega_1 \neq \omega_2$, $\omega_1 = f(\omega_1^1, \ldots, \omega_n^1)$, and $\omega_2 = f(\omega_1^2, \ldots, \omega_n^2)$. ■

For instance, the type rule in Example 1 is deterministic. The class of types that can be described by deterministic type rules is the same as the class of *tuple-distributive regular types* [3]. Additional background on type-related issues may be found in [3, 17].

For concreteness, the determinacy analysis we describe is based on *regular types* [3], which are specified by *regular term grammars* in which each type symbol has exactly one defining *type rule*, although it can easily be generalized to other type systems.

Let $\mathsf{type}[q]$ denote the type of each predicate q in a given program. In this paper, we are concerned exclusively with "calling types" for predicates —in other words, when we say "*a predicate p in a program P has type* $\mathsf{type}[p]$", we mean that in any execution of the program P starting from some class of queries of interest, whenever there is a call $p(\bar{t})$ to the predicate p, the argument tuple $\bar{t}$ in the call will be an element of the set denoted by $\mathsf{type}[p]$.

A *primitive test* is an "atom" whose predicate is a built-in such as the unification or some arithmetic predicate ($<, >, \leq, \geq, \neq$, etc.) which acts as a "test"

(note that with our assumptions of having available both mode and type information for each variable in a program, it is straightforward to identify primitive tests in a program). We define a *test* to be either a primitive test, or a conjunction $\tau_1 \wedge \tau_2$, or a disjunction $\tau_1 \vee \tau_2$, or a negation $\neg\tau_1$, where τ_1 and τ_2 are tests.

We denote the Herbrand Universe (i.e., the set of all ground terms) as $\mathcal{H}$, and the set of n–tuples of elements of $\mathcal{H}$ as $\mathcal{H}^n$. Given a (finite) set of variables $V = \{x_1, \ldots, x_n\}$, a *type assignment* ρ over V is a mapping from V to a set of types, written as $(x_1 : \omega_1, \ldots, x_n : \omega_n)$, where $\rho(x_i) = \omega_i$, for $1 \leq i \leq n$, and ω_i is a (nonempty) type representation (a type term in the algorithms that we present). Given a term t and a type representation ω, in an abuse of terminology we say that $t \in \omega$, meaning that t belongs to the set of terms denoted by ω.

We now define some notions related to clause tests and determinacy. Where necessary to emphasize the input test in a clause (i.e. the conjunction of primitive tests in the body), we will write the clause in "guarded" form, as:

$p(x_1, \ldots, x_n) \ :- \ input\_tests(x_1, \ldots, x_n) \ [\!] \ Body.$

As an example, consider a predicate defined by the clauses:

$abs(X, Y) :- X \geq 0 \ [\!] \ Y = X.$
$abs(Y, Z) :- Y < 0 \ [\!] \ Z = -Y.$

Assume we know that this predicate will always be called with its first argument bound to an integer. Obviously, for any particular call, only one of the tests '$X \geq 0$' or '$X < 0$' will succeed (i.e. the tests are mutually exclusive).

Fundamental to our approach to detecting determinacy is the notion of tests being "exclusive" w.r.t. a type assignment:

Definition 6. Two tests $\tau_1(\bar{x})$ and $\tau_2(\bar{x})$ are *exclusive* w.r.t. a type assignment $\bar{x} : \bar{\omega}$, if for every $\bar{t} \in \bar{\omega}$, $\bar{x} = \bar{t} \wedge \tau_1(\bar{x}) \wedge \tau_2(\bar{x})$ is unsatisfiable. ∎

Definition 7. Let $C_1, \ldots, C_n$, $n > 0$, be a sequence of clauses, with input tests $\tau_1(\bar{x}), \ldots, \tau_n(\bar{x})$ respectively. Let ρ be a type assignment. We say that $C_1, \ldots, C_n$ is *mutually exclusive* w.r.t. ρ if either, $n = 1$, or, for every pair of clauses C_i and C_j, $1 \leq i, j \leq n$, $i \neq j$, $\tau_i(\bar{x})$ and $\tau_j(\bar{x})$ are exclusive w.r.t. ρ. ∎

Consider a predicate p defined by n clauses $C_1, \ldots, C_n$, with input tests $\tau_1(\bar{x}), \ldots, \tau_n(\bar{x})$ respectively:

$p(\bar{x}) :- \tau_1(\bar{x}) \ [\!] \ Body_1.$
$\ldots$
$p(\bar{x}) :- \tau_n(\bar{x}) \ [\!] \ Body_n.$

We assume, without loss of generality, that each $\tau_i(\bar{x})$ is a conjunction of primitive tests (note that it is always possible to obtain an equivalent sequence of clauses where disjunctions have been removed).

Suppose that the predicate p has type $\mathsf{type}[p]$: in the interest of simplicity, we sometimes say that the predicate p is mutually exclusive w.r.t. the type $\mathsf{type}[p]$ (or simply say that the predicate p is mutually exclusive) if the sequence of

clauses $C_1, \ldots, C_n$ defining p is mutually exclusive w.r.t. the type assignment $\bar{x} : \mathsf{type}[p]$. Given a call c to predicate p in the body of a clause, we also say that c is mutually exclusive if p is. Note that if the predicate p is mutually exclusive, then at most one of its clauses will succeed for any call $p(\bar{t})$, with $\bar{t} \in \bar{\omega}$.

3 Determinacy Analysis

In this section we explain our algorithm for detecting predicates and goals that are deterministic (i.e., that produce at most one solution). Before introducing our algorithm, we give some instrumental definitions. We define the "calls" relation between predicates in a program as follows: p calls q, written $p \rightsquigarrow q$, if and only if a literal with predicate symbol q appears in the body of a clause defining p. Let $\rightsquigarrow^\star$ denote the reflexive transitive closure of $\rightsquigarrow$. The following result shows the importance of mutual exclusion information for detecting determinacy:

Theorem 1. A predicate p in the program is deterministic if, for each predicate q such that $p \rightsquigarrow^\star q$, q is mutually exclusive.

Proof Assume that p is not deterministic, i.e., there is a goal $p(\bar{t})$, with $\bar{t} \in \mathsf{type}[p]$, which is not deterministic. It is a straightforward induction on the number of resolution steps to show that there is a q such that $p \rightsquigarrow^\star q$ and q is not mutually exclusive. ∎

Our algorithm for detecting determinacy consists on first determining which predicates are mutually exclusive (which is in fact the complex part, and is explained in detail in Section 4). Then, inferring determinacy is straightforward: from Theorem 1, analysis of determinacy reduces to the determination of reachability in the call graph of the program. In other words, a predicate p is deterministic if there is no path in the call graph of the program from p to any predicate q that is not mutually exclusive. It is straightforward to propagate this reachability information in a single traversal of the call graph in reverse topological order. The idea is illustrated by the following example.

Example 2. Consider the following predicate taken from a quicksort program:

```
qs(X1,X2)  :- X1 = []  || X2 = [].
qs(X1,X2)  :- X1 = [H|L]  || part(H,L,Sm,Lg),
   qs(Sm,Sm1), qs(Lg,Lg1), app(Sm1,[H|Lg1],X2).
```

Assume that it has been inferred that `qs`/2 will be used with mode (`in`, `out`) and type (`intlist`, `-`), and assume we have already shown that `part`/4 and `app`/3 are mutually exclusive w.r.t. the types `(integer, intlist, -, -)` and `(intlist, intlist, -)` inferred for their body literals in the recursive clause above. The input tests for the sequence of clauses of `qs`/2 are `X1 = []` , `X1 = [H|L]`, which are mutually exclusive w.r.t. the type `intlist`, which means that at most one head unification will succeed for `qs`/2. It follows that a call to `qs`/2 with the first argument bound to a list of integers is deterministic, in the sense that at most one of the clauses of `qs`/2 will succeed, and if it does, it succeeds only once (thus, at most, only one solution will be produced). □

4 Checking Mutual Exclusion

Our approach to the problem of determining whether two tests $\tau_1(\bar{x})$ and $\tau_2(\bar{x})$ are *exclusive* w.r.t. a type assignment $\bar{x} : \bar{\omega}$, consists of partitioning the test $\tau_1(\bar{x}) \wedge \tau_2(\bar{x})$ such that tests in different resulting partitions involve different constraint systems, and then applying to each partition an algorithm specific to the corresponding constraint system that checks mutual exclusion. In this paper we consider two commonly encountered constraint systems: Herbrand terms with equality and disequality tests, on variables with *tuple-distributive regular types* [3] (i.e., as mentioned in Section 2, types which are specified by regular term grammars in which each type symbol has exactly one defining type rule and each type rule is *deterministic*); and for linear arithmetic tests on integer variables.

4.1 Checking Mutual Exclusion in the Herbrand Domain

We present a decision procedure for checking mutual exclusion of tests that is inspired by a result, due to Kunen [18], that the emptiness problem is decidable for Boolean combinations of (notations for) certain "basic" subsets of the Herbrand universe of a program. It also uses straightforward adaptations of some operations described by Dart and Zobel [3].

The reason the mutual exclusion checking algorithm for Herbrand is as complex as it is, is that we want a *complete* algorithm for equality and disequality tests. It is possible to simplify this considerably if we are interested in equality tests only. Before describing the algorithm, we introduce some definitions and notation.

We use the notions (to be defined later in this section) of *type-annotated term*, and in general *elementary set*, as representations which denote some subsets of $\mathcal{H}^n$ (for some $n \geq 1$). These subsets can be, for example, the set of n-tuples for which a test succeeds, or a "calling type" for a predicate p (i.e. the set denoted by $\mathsf{type}[p]$). Given a representation S (elementary set or type-annotated term), $Den(S)$ refers to the subset of $\mathcal{H}^n$ denoted by S.

Definition 8. [type-annotated term] A *type-annotated term* is a pair $M = (\bar{t}_M, \rho_M)$, where $\bar{t}_M$ is a tuple of terms, and ρ_M is a type assignment. A type-annotated term $(\bar{t}_M, \rho_M)$ denotes the set of all the ground terms $\bar{t}_M\theta$, where θ is some substitution, such that $x\theta \in \rho_M(x)$ for each variable in $\bar{t}_M$. ∎

Given a type-annotated term $(\bar{t}, \rho)$, the tuple of terms $\bar{t}$ can be regarded as a Herbrand term (i.e. a type-symbol-free type term) and ρ can be considered to be a *type substitution* [3], so that, if we apply this type substitution to $\bar{t}$, we get a pure type term (a variable-free type term). This is useful for defining the "intersection" and "inclusion" operations over type-annotated terms (that we define later), using the algorithms described by Dart and Zobel [3] for performing

[3] A type substitution is similar to a substitution that maps variables to type terms. A detailed definition of type substitutions is given in [3].

these operations over pure type terms. When we have a type-annotated term $(\bar{t}, \rho)$ such that $\rho(x) = \mu$ for each variable x in $\bar{t}$, we omit the type assignment ρ for brevity and use the tuple of terms $\bar{t}$ (recall that μ denotes the type of the entire Herbrand universe). Thus, a tuple of terms $\bar{t}$ with no associated type assignment can be regarded as a type-annotated term which denotes the set of all ground instances of $\bar{t}$.

Definition 9. [elementary set] An elementary set is defined as follows:

- Λ is an elementary set, and denotes the empty set (i.e., $Den(\Lambda) = \emptyset$);
- a type-annotated term $(\bar{t}, \rho)$ is an elementary set; and
- if A and B are elementary sets, then $A \otimes B$, $A \oplus B$ and $comp(A)$ are elementary sets that denote, respectively, the sets of (tuples of) terms $Den(A) \cap Den(B)$, $Den(A) \cup Den(B)$, and $\mathcal{H}^n \setminus Den(A)$.

■

We define the following relations between elementary sets:

- $A \sqsubseteq B$ iff $Den(A) \subseteq Den(B)$.
- $A \sqsubset B$ iff $Den(A) \subset Den(B)$.
- $A \simeq B$ iff $Den(A) = Den(B)$.

We define below two particular classes of elementary sets, namely, *cobasic sets* and *minsets*, which are suitable representations of tests for the algorithms that we present in this paper. A test $\tau(\bar{x})$ that is a conjunction of unification and disunification tests, is represented as a minset that denotes the set of ground instances of $\bar{x}$ (i.e., subsets of $\mathcal{H}^n$, assuming that $\bar{x}$ is a n-tuple) for which the test succeeds. In Figure 1 we will provide the *test2minset* function, which gives the minset representation of a test. A disunification test is represented by a cobasic set (which denotes the complementary set of a subset of $\mathcal{H}^n$).

Definition 10. [cobasic set] A cobasic set is an elementary set of the form $comp(\bar{t})$, where $\bar{t}$ is a tuple of terms (recall that $\bar{t}$ is in fact a type-annotated term $(\bar{t}, \rho)$ such that $\rho(x) = \mu$ for each variable x in $\bar{t}$). ■

Definition 11. [minset] A *minset* is either Λ or an elementary set of the form $A \otimes comp(B_1) \otimes \cdots \otimes comp(B_n)$, for some $n \geq 0$, where A is a tuple of terms, $comp(B_1), \ldots, comp(B_n)$ are cobasic sets, and for all $1 \leq i \leq n$, $B_i = A\theta_i$ and $A \not\sqsubseteq B_i$ for some substitution θ_i (i.e. $B_i \sqsubset A$). ■

For brevity, we write a minset of the form $A \otimes comp(B_1) \otimes \cdots \otimes comp(B_n)$ as A/C, where $C = \{comp(B_1), \ldots, comp(B_n)\}$. We also denote the tuple of terms of a cobasic set $Cob \equiv comp(B)$ as $\bar{t}_{Cob}$, i.e. $\bar{t}_{Cob} \equiv B$.

Example 3. We define some examples of type-annotated terms A, B, and C as follows: $A = ((x, y), (x : \alpha_1, y : \alpha_2))$, where $\alpha_1 \rightarrow \{f(\mu)\}$, and $\alpha_2 \rightarrow \{g(\mu),\ h(\mu)\}$; B is the type-annotated term such that $\bar{t}_B \equiv (f(z), w)$ and $\rho_B \equiv (z : \mu, w : \alpha_2)$ (note that A and B denote the same subset of $\mathcal{H}^n$, i.e., $Den(A) = Den(B)$); C is the type-annotated term with $\bar{t}_C \equiv (f(v_1), g(v_2), v_3, v_4, f(a), f(v_5), v_6)$ and $\rho_C \equiv (v_1 : \mu, v_2 : list, v_3 : \alpha_2, v_4 : \alpha_3, v_5 : \alpha_3, v_6 : list)$, where $\alpha_3 \rightarrow \{a,\ b\}$ and $list \rightarrow \{[\,],\ [\mu|list]\}$. □

Definition 12. [type-annotated term instance] Let A and B be two type-annotated terms. We say that A is an instance of B if $A \sqsubset B$ and there is a substitution θ such that $\bar{t}_A = \bar{t}_B\theta$. ■

Let τ_1 and τ_2 be tests which are conjunctions of unification and disunification tests, and ρ a type assignment. Let M be a type-annotated term representing the type assignment ρ. Let S_i be a minset representing τ_i, for $i = 1, 2$, the function *test2minset*, defined in Figure 1, gives the minset representation of a test, i.e., $S_i = \mathit{test2minset}(\tau_i)$.

---

test2minset(τ):

Input: a conjunction of unification and disunification tests τ. We assume that τ is of the form $E \wedge D_1 \wedge \cdots \wedge D_n$, where E is the conjunction of all unification tests of τ (i.e., a system of equations) and each D_i a disunification test (i.e., a disequation).

Output: a minset S representing the test τ (i.e., the set of tuples of terms $Den(S)$ is equal to the set of solutions of τ).

1. Let θ be the substitution associated with the solved form of E (this can be computed by using the techniques of Lassez et al. [19]).
2. Let θ_i, for $1 \leq i \leq n$, be the substitution associated with the solved form of $E \wedge N_i$, where N_i is the negation of D_i.
3. $S = A \otimes \mathit{comp}(B_1) \otimes \cdots \otimes \mathit{comp}(B_n)$, where $A = (\bar{x})\theta$ and $B_i = (\bar{x})\theta_i$, for $1 \leq i \leq n$.

Fig. 1. Definition of the function *test2minset*.

We have that τ_1 and τ_2 are exclusive w.r.t. ρ if and only if $M \otimes S_1 \otimes S_2 \simeq \Lambda$. Let S be the minset resulting of computing $S_1 \otimes S_2$ (this intersection can be trivially defined in terms of most general unifiers of the tuples of terms composing the minsets S_1 and S_2). Then, the fundamental problem is to devise an algorithm to test whether $M \otimes S \simeq \Lambda$, where M is a type-annotated term and S a minset. The algorithm that we propose is given by the boolean function $\mathit{empty}(M, S)$, defined in Figure 2.[4] Before discussing the function *empty*, we give some (instrumental) definitions.

Definition 13. [$\mathit{intersection}(M, A)$] Let M and A be a type-annotated term and a tuple of terms respectively. The function $\mathit{intersection}(M, A)$ returns $M \otimes A$ (recall that a tuple of terms is also a type-annotated term). The function *intersection* can be defined as a straightforward adaptation of the function $\mathit{unify}(\omega_1, \omega_2, T, \Theta)$ described in [3], that performs a *type unification*, where ω_1 and ω_2 are the type terms to be unified, Θ a type substitution for the variables

[4] We use the type representation of [3], and assume that there is a common set of rules where type symbols are described. For brevity, we omit such set of rules in the description of the algorithms.

in ω_1 and ω_2, and T a set of type rules defining the type symbols appearing in $\omega_1\Theta$, $\omega_2\Theta$, and Θ. Output of the function *unify* is a triple $(\omega_f, T_f, \Theta_f)$, where ω_f is a type term, Θ_f a type substitution for the variables ω_1 and ω_2, and T_f a set of type rules defining the type symbols appearing in the pure type term $\omega_f\Theta_f$, such that $T \subseteq T_f$.

Since type terms can be trivially rewritten as type-annotated terms, we can define the function $intersection(M, A)$ as follows:

- Let $(\omega_f, T_f, \Theta_f) = unify(\bar{t}_M, A, T, \Theta)$ (note that a tuple of terms is a particular case of type term, and that $\bar{t}_M$ and A are tuples of terms), where Θ is a type substitution constructed as follows:

$$x\Theta = \begin{cases} \omega \text{ if } \ x \in vars(M) \ \text{ and } \ \rho_M(x) = \omega \\ x \text{ otherwise.} \end{cases}$$

- Rewrite $\omega_f\Theta_f$ as a type-annotated term B and return it. For simplicity, we assume that the function returns only a type-annotated term B, but in fact it returns a pair (B, T_f), where, as we said before, T_f is a set of type rules defining the type symbols appearing in B.

■

Theorem 2. *The following holds for the function intersection:*

- $intersection(R, B)$ *terminates,*
- $intersection(R, B) = I$ *iff* $R \otimes B \simeq I$*, and*
- $intersection(R, B) = \Lambda$ *iff* $R \otimes B \simeq \Lambda$*.*

Proof It follows from Theorem 5.60 of [3]. ■

Definition 14. [*aliased*] Let A and $\bar{t}$ be a type-annotated term and a tuple of terms respectively, such that for all $x \in vars(A)$, $x\theta$ is a variable, where $\theta = mgu(\bar{t}_A, \bar{t})$, or $\rho_A(x) = \mu$. We define the function *aliased* as follows:

$aliased(A, \bar{t}) = \{x \in vars(A) \mid x\theta$ is a variable, and exists $x' \in vars(A)$, $x \neq x'$, such that $x\theta = x'\theta\}$.

■

Definition 15. [*expansion*] Let A be a type-annotated term, and Cob a cobasic set. Let $mgu(A, B)$ be the most general unifier of the tuples of terms A and B. We define the function *expansion* as follows:

$expansion(A, Cob) = (A', Rest)$, where:

- A' is a type-annotated term;
- $Rest$ is a set of type-annotated terms;
- for all $x \in vars(A')$, it holds that $\rho'_A(x) = \mu$, or $x\theta$ is a variable, where $\theta = mgu(\bar{t}_{A'}, \bar{t}_{Cob})$;
- $(\cup_{X \in Rest} Den(X)) \cup Den(A') = Den(A)$; and
- for all $X \in Rest$, $X \otimes \bar{t}_{Cob} \simeq \Lambda$.

$(A', Rest)$ is a pair which is a "partition" of A. ■

We now discuss the function $empty(M, S)$. First, it performs the "intersection" of M and the tuple of terms of the minset S (we assume that $S = A/C$). This intersection is implemented by the function $intersection(M, A)$ (described in Definition 13), which returns $M \otimes A$ (recall that a tuple of terms is a type-annotated term). Then, if the mentioned intersection (which we call R) is not empty, nor A is "included" in R, it calls $empty1(C, R, \emptyset)$, defined in Figure 2, which checks whether $R/C \simeq \Lambda$. This is done by checking if R is "included" in some tuple of terms of some cobasic set in C (in which case $R/C \simeq \Lambda$). For this purpose, it uses the function $included(R, \bar{t}_{Cob})$, where $\bar{t}_{Cob}$ is the tuple of terms of some cobasic set Cob that belongs to C. This function is a straightforward adaptation of the function $subset_T(\omega_1, \omega_2)$ described in [3], that determines whether the type denoted by a pure type term (a variable-free type term) is a subset of the type denoted by another (i.e., $included(R, \bar{t})$ returns **true** if and only if $R \sqsubseteq \bar{t}$).

Note that R/C can be seen as a system of one equation (corresponding to R) and zero or more disequations (each of them corresponding to a cobasic set in C). Thus the problem can be seen as determining whether such system has no solutions. We say that a cobasic set Cob is "useless" (for determining the unsatisfiability of the system) whenever if $R/(C - \{Cob\}) \not\simeq \Lambda$, then $R/C \not\simeq \Lambda$. Any useless cobasic set Cob can be removed from C, since $R/(C - \{Cob\}) \simeq \Lambda$ if and only if $R/C \simeq \Lambda$ (note that if $R/(C - \{Cob\}) \simeq \Lambda$, then obviously $R/C \simeq \Lambda$). This is done in step 1 of function *empty1*. If the tuple of terms of a cobasic set Cob in C is "disjoint" with R, then it is useless (however, there can be useless cobasic sets in C whose tuples of terms are not disjoint with R). Once we have removed useless cobasic sets, if the remaining set of cobasic sets is not empty then we go to step 3. In this step, if R is not "included" in any of the tuples of terms of the cobasic sets in C, this means that R is "too big", and thus, it is "expanded" to a set of "smaller" type-annotated terms (with the hope that each of them be "included" in the tuple of terms of some cobasic set in C). This is done in step 4, where a cobasic set Cob of C'' is selected, and R is "expanded" by using the function *expansion*. We have that $expansion(R, Cob) = (R', Rest)$, where R' is an instance of R obtained by expanding R to some "decision depth." This depth allows us to detect if the cobasic set Cob is useless. For example, assume that $R = ((X, Y), (X : list, Y : list))$ and $C = \{C_1, C_2\}$, where $C_1 = comp(([H|L], Z))$ and $C_2 = comp(([\,], Z))$. R is not included in any of the tuples of terms of the cobasic sets in C, but if we expands R using C_1, i.e., $(R_1, \{R_2\}) = expansion(R, C_1)$, where $R_1 = (([H1|L1], Y1), (H1 : \mu, L1 : list, Y1 : list))$ and $R_2 = (([\,], Y2), (Y2 : list))$, we have that R_1 and R_2 are included in $\bar{t}_{C_1}$ and $\bar{t}_{C_2}$ respectively (and thus, $R/C \simeq \Lambda$). However, in other situations, the problem cannot be solved by expanding R: assume, for example, that now $C = \{comp((Z, Z))\}$, in this case, R is not included in (Z, Z) because this tuple of terms introduces an equality constraint in $Den(R)$ (note that here R is already expanded to the "decision depth," in which the equality constraints

are given by the "aliased variables"). These aliased variables are computed in step 6 by using the function $aliased(R, \bar{t}_{Cob})$ (described in Definition 14).

Also in step 6 (after computing aliased variables), if for some $x \in vars(R')$, it holds that $\rho'_R(x) = \mu$ and either $x \in AVars$, or $x\theta'$ is not a variable, then we can say that Cob is useless. This can be proved by using the variable x to construct an instance S of R such that: assuming that there exists an instance I of R, such that $I \otimes \bar{t}_{C_1} \simeq \Lambda$ for all $C_1 \in Cset$, where $Cset = C' \cup \{CS \mid (B, A, CS) \in AL\}$, then S can be constructed from I so that $S \otimes \bar{t}_{C_2} \simeq \Lambda$ for all $C_2 \in \{Cob\} \cup Cset$.

The function $empty1(C, R, AL)$, defined in Figure 2, performs a "first round" over the cobasic sets in C. After this round (whose end is detected in step 2 by the condition $C'' = \emptyset$), cobasic sets which have been detected to be useless are ignored (removed) and the rest are stored in AL, which is an accumulation parameter. In step 7, R' and $AVars$ (besides Cob) are recorded in this parameter, because aliased variables whose type is infinite (or which after having been expanded get bound to a term containing variables whose type is infinite) allow us to detect useless cobasic sets (which are removed before $empty2(AL', R)$ is called in step 2). The function $empty2(AL, R)$, defined in Figure 3, selects a cobasic set Cob in AL, and, if R is not included in $\bar{t}_{Cob}$, then R is expanded as a set of type-annotated terms $R_1, \ldots, R_n$ by expanding only "decision variables". This ensures that every R_i is either "included" in $\bar{t}_{Cob}$ or "disjoint" with it. It also ensures that R is not infinitely expanded (note that the type of such variables is finite).

Example 4. The function call $empty(M, S)$, with $M = (\bar{t}_M, \rho_M)$, $\bar{t}_M \equiv (X)$, $\rho_M \equiv (X : list)$ and $S \equiv (X3)/\{comp([]), comp([X1|X2])\}$, returns **true** , which means that $((X), (X : list)) \otimes (X3) \otimes comp(([\,])) \otimes comp(([X1|X2])) \simeq \Lambda)$. The computation proceeds as follows:

1. $intersection(M, (X3))$ returns the type-annotated term $((X4), (X4 : list))$.
2. Since this intersection is not "empty" and $(X3)$ —which represents the type-annotated term $((X3), (X3 : \mu))$— is not "included" in $((X4), (X4 : list))$, the call $empty1(\{comp([\,]), comp([X1|X2])\}, ((X4), (X4 : list)), \emptyset)$ is performed. This call returns **true** and the computation is as follows:
   (a) We have that $((X4), (X4 : list))$ is not "included" in any of tuples of terms of the cobasic sets in $\{comp([\,]), comp([X1|X2])\}$. Thus, a cobasic set is selected from this set. Assume that $comp([X1|X2])$ is the selected cobasic set;
   (b) $(R', Rest) = expansion(((X4), (X4 : list)), comp([X1|X2]))$, where $R' = (([X5|X6]), (X5 : \mu, X6 : list))$, and $Rest = \{(([\,]), \emptyset)\}$ ($\emptyset$ denotes an empty type assignment, since $([\,])$ has no variables).
   (c) The call $included(R', ([X1|X2]))$ returns **true**, and thus the call $empty1(\{comp([])\}, (([]), \emptyset), \emptyset)$ is performed. This call also returns **true**, because $(([\,]), \emptyset) \sqsubseteq ([\,])$. Thus, the initial call returns **true**. □

The function $empty(M, S)$ is sound and complete for *tuple-distributive regular types*. While sound, the function is not complete for regular types in general. However, our experience (as we will see in Section 6) is that it is fairly accurate

in practice. Note that our applications do not require analysis algorithms to be complete (impossible in general) but rather always safe and as accurate as possible [14].

4.2 Checking Mutual Exclusion in Linear Arithmetic over Integers

In this section, we give an algorithm for checking whether two linear arithmetic tests $\tau_i(\bar{x})$ and $\tau_j(\bar{x})$ are exclusive w.r.t. the type assignment of `integer` to each variable in $\bar{x}$. This amounts to determining whether $(\exists \bar{x})(\tau_i(\bar{x}) \wedge \tau_j(\bar{x}))$ is unsatisfiable.

The system $\tau_i(\bar{x}) \wedge \tau_j(\bar{x})$ can be transformed into disjunctive normal form as:

$$(\tau_i(\bar{x}) \wedge \tau_j(\bar{x})) = \bigvee_{k=1}^{n} \bigwedge_{l=1}^{m} \phi_{kl}(\bar{x})$$

where each of the tests $\phi_{kl}(\bar{x})$ is of the form $\phi_{kl}(\bar{x}) \equiv a_0 + a_1 x_1 + \cdots + a_p x_p \ ? \ 0$, with $? \in \{=, <, \leq, >, \geq\}$. For doing this transformation, note that a test of the form $\sum_{i=0}^{p} a_i x_i \neq 0$ can be written in terms of two tests involving the operators '$>$' and '$<$':

$$(\textstyle\sum_{i=0}^{p} a_i x_i > 0) \ \vee \ (\sum_{i=0}^{p} a_i x_i < 0)$$

The resulting system, transformed to disjunctive normal form, defines a set of integer programming problems: the answer to the original mutual exclusion problem is "yes" if and only if none of these integer programming problems has a solution. Since a test can give rise to at most finitely many integer programming problems in this way, it follows that the mutual exclusion problem for linear integer tests is decidable.

Since determining whether an integer programming problem is solvable is NP-complete [9], the following complexity result is immediate:

Theorem 3. The mutual exclusion problem for linear arithmetic tests over the integers is co-NP-hard. ∎

It should be noted, however, that the vast majority of arithmetic tests encountered in practice tend to be fairly simple: our experience has been that tests involving more than two variables are rare. The solvability of integer programs in the case where each inequality involves at most two variables, i.e., is of the form $ax + by \leq c$, can be decided efficiently in polynomial time by examining the loops in a graph constructed from the inequalities [1]. The integer programming problems that arise in practice, in the context of mutual exclusion analysis, are therefore efficiently decidable.

The ideas explained in this section for linear arithmetic over integers extend directly to linear tests over the reals, which turn out to be computationally somewhat simpler.

$empty(M, S)$:

Input: a type-annotated term M and a minset S ($S = A/C$, where A is a tuple of terms, and C a set of cobasic sets).
Output: a boolean value denoting whether $M \otimes S \simeq \Lambda$.
Process:

1. if $S = \Lambda$ then return(**true**), otherwise, let $R = intersection(M, A)$;
2. if $R = \Lambda$ then return(**true**);
3. otherwise, if $included(A, R)$ then return(**false**) else return($empty1(C, R, \emptyset)$).

$empty1(C, R, AL)$:

Input: a type-annotated term R, a set of cobasic sets C, and, a set AL of triples of the form (B, AV, CS) where:

- B is a type-annotated term,
- CS is a cobasic set,
- $vars(B) \cap vars(CS) = \emptyset$,
- for all $x \in vars(B)$, $x\theta$ is a variable, where $\theta = mgu(\bar{t}_B, \bar{t}_{CS})$, and $x \in AV$ iff $x \in vars(B)$ and $\exists x' \in vars(B)$, $x \neq x'$, such that $x\theta = x'\theta$ (i.e., AV is the set of variables in $vars(B)$ which are aliased with some other variable in $vars(B)$ by θ).

Output: the boolean value **true** if $R/C_1 \simeq \Lambda$, where $C_1 = C \cup \{Cob \mid (B, A, Cob) \in AL,\ for\ some\ B\ and\ A\}$. Otherwise, returns **false**.
Process:

1. Let $C'' = \{Cob \in C \mid\ intersection(R, \bar{t}_{Cob}) \not\simeq \Lambda\}$;
2. If $C'' = \emptyset$ then return($empty2(AL', R)$), where $AL' = \{(S, AVars, Cob) \mid (S, AVars, Cob) \in AL, intersection(R, \bar{t}_{Cob}) \not\simeq \Lambda, \theta = mgu(\bar{t}_S, \bar{t}_R)$, and for all x, y such that $x \in AVars$ and $y \in vars(x\theta)$, $\rho_R(y)$ is finite (there are straightforward algorithms to test whether a type expression denotes an infinite or finite set of terms) $\}$.
3. Otherwise, if $included(R, \bar{t}_{Co})$ for some cobasic set Co in C'' then return(**true**);
4. Otherwise, take a cobasic set Cob of C'', and let $C' = C'' - \{Cob\}$ and $(R', Rest) = expansion(R, Cob)$;
5. If $included(R', \bar{t}_{Cob})$ then return($\bigwedge_{X \in Rest} empty1(C', X, AL)$);
6. Otherwise, let $AVars = aliased(R', \bar{t}_{Cob})$. If for some $x \in vars(R')$, it holds that $\rho'_R(x) = \mu$ and $x \in AVars$ or $x\theta'$ is not a variable, where $\theta' = mgu(\bar{t}_{R'}, \bar{t}_{Cob})$, then return($empty1(C', R, AL)$);
7. Otherwise, let $AL' = AL \cup \{(R', AVars, Cob)\}$;
8. return($empty1(C', R', AL') \wedge (\bigwedge_{X \in Rest} empty1(C', X, AL))$);

Fig. 2. Definition of the functions *empty* and *empty1*.

empty2(AL, R):

Input:

- a type-annotated term R,
- a set AL of triples of the form (B, AV, CS) where:
  - B is a type-annotated term,
  - CS is a cobasic set,
  - $vars(B) \cap vars(CS) = \emptyset$,
  - for all $x \in vars(B)$, $x\theta$ is a variable, where $\theta = mgu(\bar{t}_B, \bar{t}_{CS})$ ($\rho_B(x)$ can be any type, including μ), and $x \in AV$ iff $x \in vars(B)$ and exists $x' \in vars(B)$, $x \neq x'$, such that $x\theta = x'\theta$ (i.e., AV is the set of variables in $vars(B)$ which are aliased with some other variable in $vars(B)$ by θ), and
  - for all $x \in AV$, $\rho_B(x)$ is finite.

Output: a boolean value, **true** if $R/C \simeq \Lambda$, where $C = \{CS \mid (B, AV, CS) \in AL \text{ for some } B \text{ and } AV\}$ (i.e., C is the set of cobasic sets in AL), **false** otherwise.

1. If $AL = \emptyset$ then return(**false**); otherwise, take an item $A \in AL$. Assume that $A \equiv (B, AV, Cob)$, and let $AL' = AL - \{A\}$ and $\sigma = mgu(\bar{t}_B, \bar{t}_R)$;
2. if *included*$(R, \bar{t}_{Cob})$ then return(**true**);
3. otherwise, for all variables $y \in AV$, expand all variables x such that $x \in vars(y\sigma)$ (necessarily $x \in vars(R)$ and $\rho_R(x)$ is finite). Let RS be the set of type-annotated terms resulting from these expansions.
4. Let $RS' = \{r \in RS \mid intersection(r, \bar{t}_{Cob}) \simeq \Lambda\}$ (necessarily for all $s \in RS$ and $s \notin RS'$, it holds that $s \sqsubseteq \bar{t}_{Cob}$);
5. if $RS' = \emptyset$ then return(**true**);
6. otherwise return($\bigwedge_{X \in RS'}$ *empty2*(AL', X)).

Fig. 3. Definition of the function *empty2*.

4.3 Checking Mutual Exclusion: Putting it All Together

Consider a predicate p defined by n clauses $C_1, \ldots, C_n$, with input tests $\tau_1(\bar{x}), \ldots, \tau_n(\bar{x})$ respectively:

$p(\bar{x}) :- \tau_1(\bar{x}) \parallel Body_1.$
$\ldots$
$p(\bar{x}) :- \tau_n(\bar{x}) \parallel Body_n.$

Assume that the predicate p has type $\mathsf{type}[p]$. We also assume, without loss of generality, that each $\tau_i(\bar{x})$ is a conjunction of primitive tests (see Section 2).

In order to check whether the predicate p is mutually exclusive (i.e. its clauses are mutually exclusive w.r.t. the type assignment $\bar{x} : \mathsf{type}[p]$) we need to solve the problem of determining whether a pair of tests $\tau_i(\bar{x})$ and $\tau_j(\bar{x})$, $1 \leq i, j \leq n$, $i \neq j$, are exclusive w.r.t. $\bar{x} : \mathsf{type}[p]$. Let ρ be the type assignment $\bar{x} : \mathsf{type}[p]$. Consider the type assignment ρ written as a type-annotated term M, and consider each $\tau_i(\bar{x})$ written as $\tau_i^H \wedge \tau_i^A$, where τ_i^H and τ_i^A are a conjunction of primitive unification and arithmetic tests respectively (i.e., we write arithmetic

tests after unification tests). Consider also each τ_i^H written as a minset D_i (the function *test2minset*, defined in Figure 1, gives the minset representation of a test). We have that the pair of tests $\tau_i(\bar{x})$ and $\tau_j(\bar{x})$, are exclusive w.r.t. ρ if:

1. $M \otimes D_i \otimes D_j \simeq \Lambda$ (this can be checked as explained in Section 4.1), or
2. $M \otimes D_i \otimes D_j \not\simeq \Lambda$ and $\tau_i^A\theta_i \wedge \tau_j^A\theta_j$ is unsatisfiable, where θ_i (resp. θ_j), is the most general unifier of the tuple of terms of D_{ij} and D_i (resp. D_j), and D_{ij} is the minset intersection of D_i and D_j. That is, if $D_i \equiv A_i/B_i$, $D_j \equiv A_j/B_j$, and $D_{ij} \equiv A_{ij}/B_{ij}$, then $\theta_i = mgu(A_i, A_{ij})$, $A_{ij} \equiv A_i\theta_i$, $\theta_j = mgu(A_j, A_{ij})$, $A_{ij} \equiv A_j\theta_j$ (note that there exists a substitution μ_{ij}, such that $\mu_{ij} = mgu(A_i, A_j)$). We use the algorithm described in Section 4.2 for checking whether $\tau_i^A\theta_i \wedge \tau_j^A\theta_j$ is unsatisfiable.

Example 5. Let p be the predicate `partition`/4 from the familiar quicksort program. Let $X = [\,], (X = [H|L] \wedge H > Y), (X = [H|L] \wedge H \leq Y)$ be the sequence of tests for the clauses in p and let ρ be $(X : intlist, Y : integer)$, where $intlist \rightarrow \{[\,],\ [integer|intlist]\}$. In this case, we have that M is $((X, Y), (X : intlist, Y : integer))$. $\tau_1(\bar{x}) \equiv X = [\,]$, $\tau_2(\bar{x}) \equiv X = [H|L] \wedge H > Y$, and $\tau_3(\bar{x}) \equiv X = [H|L] \wedge H \leq Y$. $\tau_1(\bar{x})$ can be written as $\tau_1^H \wedge \tau_1^A$, where $\tau_1^H \equiv X = [\,]$ and $\tau_1^A \equiv true$. Similarly, $\tau_2^H \equiv X = [H|L]$ and $\tau_2^A \equiv H > Y$, and $\tau_3^H \equiv X = [H|L]$ and $\tau_3^A \equiv H \leq Y$. $D_1 \equiv ([], Y)$, $D_2 \equiv ([H|L], Y)$, and $D_3 \equiv ([H|L], Y)$. We have that `partition`/4 is mutually exclusive because: $M \otimes D_i \otimes D_j \simeq \Lambda$, for $i = 1$ and $j \in \{2, 3\}$, and (although $M \otimes D_2 \otimes D_3 \not\simeq \Lambda$), we have that $H > Y \wedge H \leq Y$ is unsatisfiable (note that $D_{2,3} \equiv ([H|L], Y)$, and θ_2 and θ_3 are the identity). □

4.4 Checking Mutual Exclusion: Dealing with the Cut

The presence of the pruning operator (cut) in the clauses of a program can help the detection of mutual exclusion of clauses. In order to take the cut into account, we simply redefine the concept of mutually exclusive clauses given in Definition 7 as follows:

Definition 16. Let $C_1, \ldots, C_n$, $n > 0$, be a sequence of clauses, with input tests $\tau_1, \ldots, \tau_n$ respectively. Let ρ be a type assignment. We say that $C_1, \ldots, C_n$ is *mutually exclusive* w.r.t. ρ if either, $n = 1$, or, for every pair of clauses C_i and C_j, $1 \leq i, j \leq n$, $i \neq j$:

1. C_i has a cut and and $i < j$, or
2. C_j has a cut and and $j < i$, or,
3. $\tau_i(\bar{x})$ and $\tau_j(\bar{x})$ are exclusive w.r.t. ρ.

■

We also have to take into account that the pruning operator introduces implicit tests. Consider a predicate p defined by n clauses $C_1, \ldots, C_n$, with input tests $\tau_1(\bar{x}), \ldots, \tau_n(\bar{x})$ respectively:

$p(\bar{x}) :- \tau_1(\bar{x}) \parallel Body_1.$
...
$p(\bar{x}) :- \tau_n(\bar{x}) \parallel Body_n.$

Let I be the set of indexes k of clauses C_k which have a cut and are before the clause C_i (i.e. $k < i$). Let τ_k^b be the test (conjunction of tests) that is before the cut in clause C_k (i.e. $\tau_k \equiv \tau_k^b \wedge \tau_k^a$, where τ_k^a is the test that is after the cut in clause C_k).

Now, instead of considering the test τ_i, for $1 \leq i \leq n$, in Definition 16, we take the test τ_i^c defined as follows:

$$\tau_i^c = \begin{cases} \tau_i & \text{if } I = \emptyset \\ (\bigwedge_{k \in I} \neg \tau_k^b) \wedge \tau_i & \text{otherwise.} \end{cases}$$

Note that the introduction of the negation in the tests τ_i^c is not a problem, since it is always possible to reduce the problem of determining whether a pair of tests τ_i^c and τ_j^c are exclusive w.r.t. a given type assignment, to one o more exclusion subproblems where the pair of tests involved in each subproblem are conjunctions of primitive tests (transforming tests to disjunctive normal form).

5 Improving Determinacy Analysis using Cut

The presence of the pruning operator in the clauses of a program not only improves detection of mutual exclusion, but it can also help in the overall process of detecting deterministic predicates. Besides helping the detection of mutual exclusion of clauses (as we have seen before), it can also improve the propagation algorithm given in Section 3. Assume that we would like to infer that a predicate p is deterministic. Consider any clause defining p in which one or more cuts appear, and any body literals that appear to the left of the rightmost cut in that clause. Those literals are not required to be deterministic (we say that a literal with predicate symbol q is deterministic if q is). In other words, in Theorem 1, we can use a restricted definition ($\rightsquigarrow_r$) of the "call" relation ($\rightsquigarrow$) between predicates in a program, defined as follows: $p \rightsquigarrow_r q$, if and only if a literal with predicate symbol q appears in the body of a clause defining p, and there is no cut to the right of this literal in the clause. Similarly, $\rightsquigarrow_r^\star$ denotes the reflexive transitive closure of $\rightsquigarrow_r$.

6 A Prototype Implementation

In order to evaluate the effectiveness and efficiency of our approach to determinacy analysis we have constructed a relatively complete prototype which performs such analysis in an automatic way. The system takes Prolog programs as input,[5] which include a module definition in the standard way. In addition, the types and modes of the arguments of exported predicates are either given

[5] In fact, the input language currently supported includes also a number of extensions —such as functions or feature terms— which are translated by the first (expansion) passes of the Ciao compiler to clauses, possibly with cut.

or obtained from other modules during modular type and mode analysis (including the intervening type definitions). The system uses the *CiaoPP* PLAI analyzer to derive mode information, using, for the reported experiments, the Sharing+Freeness domain [22], and an adaptation of Gallagher's analysis to derive the types of predicates [8]. The resulting type- and mode-annotated programs are then analyzed using the algorithms presented for Herbrand and linear arithmetic tests.

Herbrand mutual exclusion is checked by a naive direct implementation of the analyses presented. Testing of mutual exclusion for linear arithmetic tests is implemented directly using the Omega test [25]. This test determines whether there is an integer solution to an arbitrary set of linear equalities and inequalities, referred to as a problem.

We have tested the prototype first on a number of simple standard benchmarks, and then on more complex ones. The latter are taken from those used in the cardinality analysis of Braem *et al.* [2], which is the closest related previous work that we are aware of. In the case of *Kalah*, we have inserted the missing cuts as is also done in [2], to make the comparison meaningful. Some relevant results of these tests are presented in Table 1. **Program** lists the program names, **N** the number of predicates in the program, **D** the number of predicates detected by the analysis as deterministic, **M** the number of predicates whose tests are mutually exclusive, **C** the number of deterministic predicates detected in [2], $\mathbf{T}_D$ the time required by the determinacy analysis (Ciao version 1.9p111 and CiaoPP-0.8, on a medium-loaded Pentium IV Xeon 2.0Ghz, 1Gb of RAM memory, running Red Hat Linux 8.0, and averaging several runs, eliminating the best and worst values), $\mathbf{T}_M$ the time required to derive the modes and types, and $\mathbf{T}_T$ the total analysis time (all times are given in milliseconds). Averages (per predicate in the case of analysis time) are also provided in the last row of the table.

The results are quite encouraging, showing that the developed analysis is fairly accurate. The analysis is more powerful in some cases than the cardinality analysis [2], and at least as accurate in the others. It is pointed out in [2] that determinacy information can be improved by using a more sophisticated type domain. This is also applicable to our analysis, and the types inferred by our system are similar to those used in [2]. The determinacy analysis times are also encouraging, despite the currently relatively naive implementation of the system (for example, the call to the omega test is done by calling an external process). The overall analysis times are also reasonable, even when including the type and mode analysis times, which are in any case very useful in other parts of the compilation process.

7 Conclusions

We have proposed an analysis for detecting procedures and goals that are deterministic (i.e. that produce at most one solution), or predicates whose clause tests are mutually exclusive, even if they are not deterministic (because they

Program	N	D (%)	M (%)	C	T_D	T_M	T_T
Hanoi	2	2 (100)	2 (100)	N/A	69	79	148
Fib	1	1 (100)	1 (100)	N/A	39	19	58
Mmatrix	3	3 (100)	3 (100)	N/A	89	79	168
Tak	1	1 (100)	1 (100)	N/A	49	29	78
Subs	1	1 (100)	1 (100)	N/A	70	19	89
Reverse	2	2 (100)	2 (100)	N/A	39	19	58
Qsort	3	3 (100)	3 (100)	3 (100)	50	69	119
Qsort2	5	5 (100)	5 (100)	5 (100)	99	70	169
Queens	6	3 (50)	5 (83)	2 (33)	99	59	158
Gabriel	20	6 (30)	11 (55)	4 (20)	360	279	639
Kalah	44	40 (91)	42 (95)	40 (91)	1110	3589	4699
Plan	16	8 (50)	12 (75)	3 (19)	459	949	1408
Credit	25	18 (72)	21 (84)	16 (64)	1209	359	1568
Pg	10	6 (60)	9 (90)	6 (60)	440	209	649
Mean	–	71%	85%	61%	30 (/p)	42 (/p)	72 (/p)

Table 1. Accuracy and efficiency of the determinacy analysis (times in mS).

call other predicates which are nondeterministic). This approach has advantages w.r.t. previous approaches in that it provides an algorithm for detecting mutual exclusion and it handles disequality constraints on the Herbrand domain and arithmetic tests.

We have implemented the proposed analysis and integrated it into the *CiaoPP* system, which also infers automatically the mode and type information that the proposed analysis takes as input. The results of the experiments performed on this implementation show that the analysis is fairly accurate and efficient, providing more accurate or similar results, regarding accuracy, than previous proposals, while offering substantially higher automation, since typically no information is needed from the user.

Acknowledgments

This work has been supported in part by the European Union IST program under contracts IST-2001-38059 "ASAP", by MCYT project TIC 2002-0055 "CUBICO", by FEDER infrastructure project UNPM-E012, and by the Prince of Asturias Chair in Information Science and Technology at the University of New Mexico. We would also like to thank the anonymous reviewers for their useful comments on earlier versions of the paper.

References

1. B. Aspvall and Y. Shiloach. A polynomial time algorithm for solving systems of linear inequalities with two variables per inequality. In *Proc. 20th ACM Symposium on Foundations of Computer Science*, pages 205–217, October 1979.
2. C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality analysis of prolog. In *Proc. International Symposium on Logic Programming*, pages 457–471, Ithaca, NY, November 1994. MIT Press.
3. P.W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In *Types in Logic Programming*, pages 157–187. MIT Press, 1992.
4. S. Dawson, C.R. Ramakrishnan, I.V. Ramakrishnan, and R.C. Sekar. Extracting Determinacy in Logic Programs. In *1993 International Conference on Logic Programming*, pages 424–438. MIT Press, June 1993.
5. S.K. Debray and N.W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.
6. S.K. Debray and D.S. Warren. Functional computations in logic programs. *ACM Transactions on Programming Languages and Systems*, 11(3):451–481, 1989.
7. B. Demoen, M. Garcia de la Banda, W. Harvey, K. Marriott, and P. Stuckey. An overview of HAL. In *PPCP'99: Principles and Practice of Constraint Programming*, pages 174–178, 1999.
8. J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.
9. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, New York, 1979.
10. Roberto Giacobazzi and Laura Ricci. Detecting determinate computations by bottom-up abstract interpretation. In *Symposium proceedings on 4th European symposium on programming*, pages 167–181. Springer-Verlag, 1992.
11. G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, July 2001.
12. F. Henderson, Z. Somogyi, and T. Conway. Determinism analysis in the Mercury compiler. In *Proc. Australian Computer Science Conference*, pages 337–346, Melbourne, Australia, January 1996.
13. M. Hermenegildo and M. Carro. Relating Data–Parallelism and (And–) Parallelism in Logic Programs. *The Computer Languages Journal*, 22(2/3):143–163, July 1996.
14. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International Static Analysis Symposium (SAS'03)*, number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.
15. M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.
16. P.M. Hill and A. King. Determinacy and determinacy analysis. *Journal of Programming Languages*, 5(1):135–171, December 1997.
17. G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):205–258, July 1992.
18. K. Kunen. Answer Sets and Negation as Failure. In *Proc. of the Fourth International Conference on Logic Programming*, pages 219–228, Melbourne, May 1987. MIT Press.

19. J.-L. Lassez, M. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–626. Morgan Kaufman, 1988.
20. P. López-García, M. Hermenegildo, and S.K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 22:715–734, 1996.
21. J. Morales, M. Carro, and M. Hermenegildo. Improving the Compilation of Prolog to C Using Moded Types and Determinism Information. In *Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages*, pages 86–103, Dallas (Texas), June 2004. Springer-Verlag.
22. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.
23. E. Pontelli, G. Gupta, D. Tang, M. Carro, and M. Hermenegildo. Improving the Efficiency of Nondeterministic And–parallel Systems. *The Computer Languages Journal*, 22(2/3):115–142, July 1996.
24. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.
25. W. Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *Communications of the ACM*, 35(8):102–114, August 1992.
26. V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–93. ACM, April 1991. SIGPLAN Notices vol 26(7), July 1991.
27. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *JLP*, 29(1–3), October 1996.
28. P. Van Roy. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming*, 19/20:385–441, 1994.