Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Analyses

Pedro LOPEZ-GARCIA[1,2], Francisco BUENO[3] and Manuel HERMENEGILDO[1,3]

[1] *IMDEA Software, Madrid, Spain*
{pedro.lopez,manuel.hermenegildo}@imdea.org

[2] *Spanish Research Council (CSIC), Spain*

[3] *Technical University of Madrid (UPM), Spain*
{bueno,herme}@fi.upm.es

Received 17 August 2009

Abstract We propose an analysis for detecting procedures and goals that are deterministic (i.e., that produce at most one solution at most once), 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. The analysis takes advantage of the pruning operator in order to improve the detection of mutual exclusion and determinacy. It also supports arithmetic equations and disequations, as well as equations and disequations on terms, for which we give a complete satisfiability testing algorithm, w.r.t. available type information. Information about determinacy can be used for program debugging and optimization, resource consumption and granularity control, abstraction carrying code, 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 and Checking, Types, Program Analysis, Debugging, Optimization.

§1 Introduction

Knowing that certain predicates are deterministic for a given class of calls has a number of interesting applications such as 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, abstraction carrying code, etc.

By a predicate being deterministic we mean that it produces at most one solution at most once. It is also interesting to detect predicates whose clauses are mutually exclusive (which implies that at most one of them will succeed) even if they are not deterministic because they call other predicates that can produce more than one solution (i.e., that are not deterministic). In this paper we propose a method whereby we can detect procedures and goals that are deterministic, or predicates whose clauses are mutually exclusive. Moreover, we show that, given (upper approximations of) mode and type information, it is feasible to fully automatize our approach, yielding an effective automatic determinacy analysis. The paper is an extended version of our previous proposal [17], which includes more detailed descriptions of the algorithms, more illustrative examples, and more theorems about the termination, correctness or completeness of the algorithms.

There has been much interest on determinacy detection in the literature (see [14, 12] and its references), using several different forms of determinism. Arguably, one of the first practical determinacy analyses was the one proposed by Sahlin [23], in the context of the Mixtus partial evaluator. This analysis was later reconstructed and semantically justified, using a denotational semantics of Prolog programs with cut, by Mogensen [20]. The motivation behind this determinacy analysis was, indeed, to be able to unfold predicates with cuts in their clauses. Therefore, the analysis concentrated on the cut and the control flow of the program: interestingly, the proposal in [23] does not take into account predicate arguments. Using a small database of number of possible solutions of built-ins and an analysis of the control structure of the program, estimations of the number of solutions of predicates were performed. The accuracy of this approach has limitations and this is one of the reasons why we explore instead an approach based on the handling of built-ins as tests.

The line of work closest to ours starts with [6], in which functional com-

putations are detected and exploited. However, the notion of mutual exclusion in this work (and in our proposal [17], of which this paper is an extended version, as already mentioned) is not based on constraint satisfaction. This concept is also used in the analysis presented in [5], where, nonetheless, no algorithms are provided for the detection of mutual exclusion and also the cut is not taken into account. In [9] 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.

A notion of constraint satisfaction is also present in the approach of [19, 14], which might be considered complementary to ours. Their analyses differ from ours in that they are not goal-oriented and in the mutual exclusion conditions. In particular, the first work [19] does not handle the cut, and cannot exploit certain program tests that select clauses on execution (e.g., arithmetic tests) which our proposal handles. The second work [14] remedies these deficiencies. Still, it concentrates on inferring determinacy conditions, not on checking them. The conditions of [14] are richer than ours, since they use success pattern analysis to infer them, based on size relationships between arguments and depth-k abstractions, together with backward analysis. Determinacy conditions are then synthesised in the form of rigidity formulas. For checking them a rigidity analysis is required, to test whether the (propositional) formula holds or not. Instead, we focus on the checking and not on building the conditions. For conditions, we use tests on the instantiation state of arguments which are simply collected from the program text. For the checking, classical mode and type analyses are instrumental. Indeed, our main contribution is a procedure to check satisfiability of the tests which is complete, disregarding how conditions are synthesised.

Several programming systems also make use of determinacy, e.g., Mercury [24, 10] and HAL [7]. The Mercury and HAL systems allow the programmer to declare that a predicate will produce at most one solution, and attempt to verify this with respect to the Herbrand terms with unification tests. As far as we know, both systems use the same analysis [10], which does not handle disunification tests on the Herbrand domain. This approach also does not 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 other words, in addition to proposing concrete algorithms, we also show in this paper that our determinacy analysis

can be performed automatically, and is feasible, accurate, and efficient. We do this by integrating it into the Ciao programming system, in particular, into its preprocessor, CiaoPP [11], which performs analysis, debugging, verification and optimization tasks, and thus connecting the determinacy analysis with state-of-the-art type and mode analyses.

§2 Preliminaries

A goal, a class of goals, or a predicate (i.e., all goals for it) are *deterministic* when they produce at most one solution at most once. When reasoning about determinacy, it is a necessary condition (but not sufficient) that clauses of the predicate be pairwise mutually exclusive, i.e., that only one clause will produce solutions. Additionally, it has to produce only one solution. For reasoning about mutual exclusion one needs to gather success patterns for each predicate clause, i.e., constraints that the solutions produced by the clause satisfy. Then the basic condition for mutual exclusion is that such success patterns cannot be satisfied simultaneously. This is checked for against available information on the goals being analyzed for determinacy.

We will be using as success patterns *tests*, which will be unification equations and disequations on terms, and linear equations and disequations on integers or reals. For the checking, we will assume that type information is available, generally as the result of a previous analysis. For concreteness, the determinacy analysis we describe is based on *regular types* [4], which are specified by *regular term grammars*, as explained below, although the concepts should be easily adaptable to other type systems.

2.1 Regular Types

A type is a set of (Herbrand) terms, and can be defined by using a number of different representations, such as *type terms* and *regular term grammars* as in [4], or *type graphs* as in [13]), or simply predicates as in the Ciao system [3]. We will use the formalism of [4], and summarize below the relevant concepts.

A *type symbol* is an abstraction of a set of Herbrand terms (i.e., of a type). We use the Greek letter α for referring to type symbols in general (with subscripts if necessary). The γ function maps each type symbol to the set of Herbrand terms that it represents. Given a type symbol α, the set of terms (i.e., the type) represented by it is denoted as $\gamma(\alpha)$. To enhance readability, we abuse notation and use α instead of $\gamma(\alpha)$ when no ambiguity is possible.

We assume the existence of an infinite set of *type symbols*, which is disjoint with the sets of constant symbols, function symbols, and variables. There are two special type symbols: μ, that represents the type of the entire Herbrand universe; and ϕ, that represents the empty type (i.e., $\gamma(\phi) = \emptyset$). There is a distinguished non-empty finite subset of the set of type symbols called the set of *base type symbols*, which represent *base types*. For any base type α, we assume that $\gamma(\alpha)$ is infinite and there are effective tests for membership of a given Herbrand term in $\gamma(\alpha)$.

Example 2.1
Examples of base type symbols that we use in our determinacy analysis are: int, such that the base type $\gamma(int)$ is the set of all constant symbols that represent integer numbers; and atm, such that the base type $\gamma(atm)$ is the set of all constant symbols that do not represent numbers.

A *type term* is either a constant symbol, a variable, a type symbol, or a term $f(\omega_1, \ldots, \omega_n)$, where f is an n-ary function symbol, and each ω_i is a type term. Note that all type symbols are type terms, however, the converse is not true. A *pure type term* is one which does not contain variables. A *Herbrand term* is a type term which does not contain type symbols (it can contain variables).

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. We denote sets of type rules, that is, *regular term grammars*, by the letter T (as in [4]). A (non-base) type symbol α is *defined* in, or by, a set of type rules T if there exists a type rule $(\alpha \rightarrow \Upsilon) \in T$. 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 $(\alpha \rightarrow \Upsilon) \in T$, 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. Moreover, we will also assume that every type rule is *deterministic*, i.e., every element of Υ is a base type symbol or a pure type term of the form $f(\alpha_1, \ldots, \alpha_n)$, $n \geq 0$, 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 $\omega_2 = f(\omega_1^2, \ldots, \omega_n^2)$ (i.e., there is no pair of pure type terms in Υ with the same principal functor). The class of types that can be described by deterministic type rules is the same as the class of *tuple-distributive regular types* [4]. Additional background on type-related issues may be found in [4, 13].

Example 2.2
The type rule $list \rightarrow \{[\,],\ [\mu|list]\}$ defines the type symbol $list$, that denotes the set of all lists. The type rule $intlist \rightarrow \{[\,],\ [int|intlist]\}$ defines the type symbol $intlist$, that denotes the set of all lists of integer numbers.

It is also possible to provide polymorphism in our setting. Since we use types for describing instantiation patterns, a polymorphic type such as, e.g., $list(\alpha) \rightarrow \{[\,],\ [\alpha|list(\alpha)]\}$ is useful only in the description of the list structure, but not of the elements. Thus, the instance type $list(\mu)$ (i.e., $list$) serves the same purposes. Instances of polymorphic types are thus "computed away" (so that, e.g., $list(int)$ yields $intlist$) and our approach handles them in this way.

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

Definition 2.1 (type assignment)
Given a (finite) tuple of variables $\bar{x} = (x_1, \ldots, x_n)$, a *type assignment* ρ over $\bar{x}$ maps each variable x_i for $1 \leq i \leq n$ to a (nonempty) pure type term ω_i, i.e., $\rho(x_i) = \omega_i$. We write the *type assignment* ρ as $\bar{x} : \bar{\omega}$, where $\bar{\omega}$ is the tuple of pure type terms $(\omega_1, \ldots, \omega_n)$. However, we sometimes abuse of notation and write ρ as $(x_1 : \omega_1, \ldots, x_n : \omega_n)$.

2.2 Tests (and Modes)

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. A *primitive test* is a positive literal whose predicate symbol is a built-in such as the unification or some arithmetic built-in 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). Primitive tests which are true of the successes of a given clause are gathered together to form the test of that clause. For concreteness, in our experiments (Section 5) we will gather for each clause

the primitive tests occurring in the program text of that clause. One could use more sophisticated approaches, such as backwards analysis with a depth-k abstraction [12]. Our approach remains valid regardless of the means used to build the tests. For example, if term structure information is available it will be used in the algorithms below as if it appeared in the program text.

Because of limitations of state-of-the-art technology in type analysis, primitive tests have to be carefully selected. Actual, working type analyses infer types which denote sets of terms that are closed under substitution. On the contrary, our algorithms will be based on types which denote sets of ground terms. The gap between these two classes of types is covered with the use of modes.

In practice, the difference amounts to the interpretation of the universal type symbol μ. In the ground interpretation, μ denotes the set of all ground terms. Otherwise, μ (i.e., the classical *top* in type analyses) also denotes terms which may contain variables. This issue is important in deciding whether certain (unification) literals can act as tests or not and, therefore, whether they can be used in mutual exclusion conditions or not. For example, consider two tests `X=[a]` and `X=[b]` for different clauses. Assume we are analyzing goals which satisfy the type assignment $(\texttt{X}) : (\alpha)$ with type rule $\alpha \rightarrow \{[\mu]\}$. In the ground interpretation the two tests are mutually exclusive, but they are not in the other interpretation (since the head of the list constructor in `X` might be a free variable). Mode information is then essential in distinguishing such cases.

In our experiments, we will use groundness and freeness information obtained from a sharing analysis to establish the modes. This information is used to classify primitive tests, and only those regarded as *input tests* will be considered when building tests for clauses. Input tests perform a comparison of (numerical) values or a matching of terms, rather than a proper unification. Given mode and type information on the program, it is straightforward to identify them.

Example 2.3
Consider the literal `X is Y+1` appearing in the body of a clause. If the available mode and type information asserts that, just before calling this literal, variables `X` and `Y` are bound to integer numbers, then the literal is considered a primitive (arithmetic) input test. However, if the mode and type information asserts that `X` is an unbound variable and `Y` is bound to an integer, then the literal acts as an assignment and thus is not considered a test. If there is a body literal of the form `X = Y` and the information asserts that variables `X` and `Y` will be

bound to ground terms upon call, then the literal is considered to be a primitive (unification) input test. If the information asserts that any of the variables `X` or `Y` are free, then the literal is not considered a test.

Where necessary to emphasize the input test in a clause we will write the clause in "guarded" form. As an example, consider a predicate that is called as $\texttt{abs}(\texttt{X},\texttt{Y})$, where `X` is bound to an integer and `Y` is a free variable, to obtain the absolute value of `X`. Its definition will be written as:

$\texttt{abs}(\texttt{X},\texttt{Y}) :- \texttt{X} \geq \texttt{0} \parallel \texttt{Y} = \texttt{X}.$
$\texttt{abs}(\texttt{X},\texttt{Y}) :- \texttt{X} < \texttt{0} \parallel \texttt{Y is } -\texttt{X}.$

Obviously, for any particular call in the class above, only one of the tests $\texttt{X} \geq \texttt{0}$ or $\texttt{X} < \texttt{0}$ will succeed (i.e., the tests are mutually exclusive).

Note that the distinction between tests and input tests is due only to limitations in the technology used in our experiments. In fact, we will be using the word test throughout the rest of the paper when talking about mutual exclusion conditions. The following definition characterizes tests and will be instrumental in the formal results:

Definition 2.2 (solutions of a test)
Given a test $\tau(\bar{x})$, $Sols(\tau(\bar{x}))$ is the set of all tuples of ground terms $\bar{e}$ which are instances of $\bar{x}$ such that $\bar{x} = \bar{e} \wedge \tau(\bar{x})$ is satisfiable (i.e., test $\tau(\bar{e})$ succeeds).

2.3 Mutual Exclusion

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

Definition 2.3
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 $\bar{t} \in \gamma(\bar{\omega})$, $\bar{x} = \bar{t} \wedge \tau_1(\bar{x}) \wedge \tau_2(\bar{x})$ is unsatisfiable.

Definition 2.4 (mutual exclusion)
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. Let predicate p have type $\mathsf{type}[p]$: in the interest of simplicity, we sometimes say that predicate p is mutually exclusive w.r.t. the type $\mathsf{type}[p]$ (or simply that 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 $\bar{t} \in \gamma(\mathsf{type}[p])$.

§3 Determinacy Analysis

In this section we explain our algorithm for detecting predicates and calls that are deterministic. 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 3.1
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 of first determining which predicates are mutually exclusive (which is in fact the convoluted part, and is explained in detail in Section 4). Then, inferring determinacy is straightforward: from Theorem 3.1, 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 3.1
Consider the classical quicksort program with a main calling mode in which the first argument is ground and the second one is free. Figure 1 shows the guarded

```
qs(L,SL) :- L = []  || SL = [].
qs(L,SL) :- L = [H|T]  || part(H,T,Littles,Bigs),
       qs(Littles,SLs), qs(Bigs,SBs), app(SLs,[H|SBs],SL).

part(L, _C,Left,Right) :- L = []  || Left = [], Right = [].
part(L,C,Left,Right) :- L = [E|R], E < C  || Left = [E|Left1],
       part(R,C,Left1,Right).
part(L,C,Left,Right) :- L = [E|R], E >= C  || Right = [E|Right1],
       part(R,C,Left,Right1).

app(L1,L2,L3) :- L1 = []  || L2 = L3
app(L1,L2,L3) :- L1 = [X|Xs]  || L3 = [X|Zs], app(Xs,L2,Zs).
```

Fig. 1 A quicksort program.

version of the program for this mode. Assume calling type (`intlist`, `-`) for `qs`/2. The calling types for `part`/4 and `app`/3 are (`intlist`, `int`, `-`, `-`) and (`intlist`, `intlist`, `-`) respectively. Since determinacy analysis traverses the call graph in reverse topological order, it considers first predicates `part`/4 and `app`/3.

The input tests for the clauses of `part(L,C,Left,Right)` are $\tau_1^{part}(\mathtt{L},\mathtt{C}) \equiv \mathtt{L} = [\,]$, $\tau_2^{part}(\mathtt{L},\mathtt{C}) \equiv \mathtt{L} = [\mathtt{E}|\mathtt{R}] \wedge \mathtt{E} < \mathtt{C}$ and $\tau_3^{part}(\mathtt{L},\mathtt{C}) \equiv \mathtt{L} = [\mathtt{E}|\mathtt{R}] \wedge \mathtt{E} >= \mathtt{C}$. According to the calling type, the analysis uses the type assignment $\rho^{part} \equiv (\mathtt{L},\mathtt{C}) : (\mathtt{intlist},\mathtt{int})$, and infers that $\tau_i^{part}(\mathtt{L},\mathtt{C}), i = 1,2,3$ are mutually exclusive w.r.t. ρ^{part} (we will explain the details in Section 4). It means that at most one of these tests will succeed. Thus, clauses of `part`/4 are mutually exclusive. It follows that calls to `part`/4 which satisfy the calling types are deterministic.

Similarly, the input tests for the sequence of clauses of `app(L1,L2,L3)` are $\tau_1^{app}(\mathtt{L1},\mathtt{L2}) \equiv \mathtt{L1} = [\,]$ and $\tau_2^{app}(\mathtt{L1},\mathtt{L2}) \equiv \mathtt{L1} = [\mathtt{X}|\mathtt{Xs}]$. The type assignment ρ^{app} corresponding to the calling types for `app`/3 is $(\mathtt{L1},\mathtt{L2}) : (\mathtt{intlist},\mathtt{intlist})$. The analysis infers that $\tau_1^{app}(\mathtt{L1},\mathtt{L2})$ and $\tau_2^{app}(\mathtt{L1},\mathtt{L2})$ are exclusive w.r.t. the type assignment ρ^{app}. Thus, it follows that calls to `app`/3 which satisfy the calling types are also deterministic.

Finally, the input tests for the sequence of clauses of `qs(L,SL)` are $\tau_1^{qs}(\mathtt{L}) \equiv \mathtt{L} = [\,]$ and $\tau_2^{qs}(\mathtt{L}) \equiv \mathtt{L} = [\mathtt{H}|\mathtt{T}]$. The type assignment ρ^{qs} corresponding to the calling types is $(\mathtt{L}) : (\mathtt{intlist})$. We have that $\tau_1^{qs}(\mathtt{L})$ and $\tau_2^{qs}(\mathtt{L})$ are exclusive w.r.t. type assignment ρ^{qs}. Thus, clauses of `qs`/2 are mutually exclusive. Moreover, since the calls to the predicates `part`/4 and `app`/3 in the body of the

clauses defining qs/2 have been proved to be deterministic, it follows that calls to qs/2 with the first argument bound to a list of integers are deterministic.

3.1 Improving Determinacy Analysis using Cut

The presence of pruning operators in program clauses can be exploited to improve the overall process of detecting deterministic predicates. Besides helping the detection of mutual exclusion of clauses (as we will see below in Section 4.4), it can also improve the propagation algorithm given above. Assume that we want to infer whether 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. In other words, in Theorem 3.1, we can use a restricted definition ($\leadsto_r$) of the "call" relation ($\leadsto$) between predicates in a program, defined as follows: $p \leadsto_r q$, 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, $\leadsto_r^\star$ denotes the reflexive transitive closure of $\leadsto_r$.

§4 Checking Mutual Exclusion

Our approach to the problem of determining whether a set of tests $\tau_i(\bar{x})$ for $i = 1, \ldots, n$ are *mutually exclusive* w.r.t. a type assignment $\bar{x} : \bar{\omega}$, consists of reducing the problem to subproblems, each subproblem involving tests of the same type, i.e., defining a particular constraint system. Each subproblem is solved by applying an algorithm that is specific to the corresponding constraint system that checks mutual exclusion. In this paper we consider two commonly encountered constraint systems: Herbrand terms with unification and disunification tests, on variables with *tuple-distributive regular types* [4] (see Section 2.1) and linear arithmetic tests on integer variables.

Example 4.1

Consider the predicate part/4 taken from the quicksort program shown in Figure 1. For the sequence of clauses of part(L,C,Left,Right) we have three input tests $\tau_i(\bar{x})$, $i = 1, 2, 3$, where $\bar{x} \equiv (\mathtt{L}, \mathtt{C})$ in this case. As commented in Example 3.1, the input tests are (omitting $\bar{x}$ and the superscript *part* for simplicity): $\tau_1 \equiv \mathtt{L} = [\,]$, $\tau_2 \equiv \mathtt{L} = [\mathtt{E}|\mathtt{R}] \wedge \mathtt{E} < \mathtt{C}$ and $\tau_3 \equiv \mathtt{L} = [\mathtt{E}|\mathtt{R}] \wedge \mathtt{E} >= \mathtt{C}$. We will separate Herbrand tests from arithmetic tests and write τ_1 as $\tau_1^H \wedge \tau_1^A$, where $\tau_1^H \equiv \mathtt{L} = [\,]$ and $\tau_1^A \equiv \mathit{true}$. Similarly, $\tau_2^H \equiv \mathtt{L} = [\mathtt{E}|\mathtt{R}]$ and $\tau_2^A \equiv \mathtt{E} < \mathtt{C}$, and

$\tau_3^H \equiv \mathtt{L} = [\mathtt{E}|\mathtt{R}]$ and $\tau_3^A \equiv \mathtt{E} >= \mathtt{C}$.

We have to check that the tests $\tau_i(\bar{x})$, $i = 1, 2, 3$, are mutually exclusive w.r.t. the type assignment $\rho \equiv (\mathtt{L}, \mathtt{C}) : (\mathtt{intlist}, \mathtt{int})$. This problem is reduced to two subproblems: a) Checking that the tests $\mathtt{L} = [\,]$ and $\mathtt{L} = [\mathtt{E}|\mathtt{R}]$ are exclusive w.r.t. ρ, which prove that τ_1 and τ_2 (as well as τ_1 and τ_3) are exclusive (since the Herbrand parts of the tests are exclusive), and b) Checking that the tests $\mathtt{E} < \mathtt{C}$ and $\mathtt{E} >= \mathtt{C}$ are exclusive w.r.t. the type assignment $(\mathtt{C}, \mathtt{E}) : (\mathtt{int}, \mathtt{int})$, which prove that τ_2 and τ_3 are exclusive. In this second subproblem, the Herbrand parts of the tests are not exclusive (in fact, both of them are the same test, $\mathtt{L} = [\mathtt{E}|\mathtt{R}]$), and hence, it is necessary to check the exclusion of the arithmetic parts.

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 [15], that establishes 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 [4]. The reason the mutual exclusion checking algorithm for Herbrand is as convoluted as it is, is that we want a *complete* algorithm for unification and disunification tests. It is possible to make it more clear if we are interested in unification tests only.

Before describing the algorithm, we introduce some definitions and notation. 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 $\mathcal{H}^n$. We use the notions (to be defined in the following) 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), the denotation of S, $Den(S)$ refers to the subset of $\mathcal{H}^n$ denoted by S.

Definition 4.1 (type-annotated term)
A *type-annotated term* δ is a pair $(\bar{t}_\delta, \rho_\delta)$, where $\bar{t}_\delta$ is a tuple of terms, and ρ_δ is a type assignment.

We will represent type-annotated terms with the symbol δ possibly subscripted. Given a type-annotated term $\delta = (\bar{t}_\delta, \rho_\delta)$, the denotation of δ, $Den(\delta)$

is the set of all the ground terms $\bar{t}_\delta\theta$, where θ is some substitution, such that $x\theta \in \gamma(\rho_\delta(x))$ for each variable in $\bar{t}_\delta$. In other words, $Den(\delta)$ is the set of all the ground instances of $\bar{t}_\delta$ resulting from replacing the variables in $\bar{t}_\delta$ by a term belonging to the type assigned to those variables by ρ_δ.

Example 4.2
We define some examples of type-annotated terms δ_1, δ_2, and δ_3 as follows: $\delta_1 = ((x, y), (x, y) : (\alpha_1, \alpha_2))$, where $\alpha_1 \rightarrow \{f(\mu)\}$ and $\alpha_2 \rightarrow \{g(\mu),\ h(\mu)\}$; δ_2 is the type-annotated term $(\bar{t}_2, \rho_2)$ such that $\bar{t}_2 \equiv (f(z), w)$ and $\rho_2 \equiv (z, w) : (\mu, \alpha_2)$ (note that δ_1 and δ_2 denote the same subset of $\mathcal{H}^2$, i.e., $Den(\delta_1) = Den(\delta_2)$); δ_3 is the type-annotated term $(\bar{t}_3, \rho_3)$ with $\bar{t}_3 \equiv (f(v_1), g(v_2), v_3, v_4, f(a), f(v_5), v_6)$ and $\rho_3 \equiv (v_1, v_2, v_3, v_4, v_5, v_6) : (\mu, list, \alpha_2, \alpha_3, \alpha_3, list)$, where $\alpha_3 \rightarrow \{a,\ b\}$ and $list \rightarrow \{[\,],\ [\mu|list]\}$.

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*,[*1] 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 [4] for performing 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 $\bar{t}$. 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 4.2 (elementary set)
An *elementary set* is defined as follows:

- Λ is an elementary set.
- A type-annotated term $(\bar{t}, \rho)$ is an elementary set.
- If A and B are elementary sets, then $A \otimes B$, $A \oplus B$ and $comp(A)$ are elementary sets.

Since we have already defined the denotation of type-annotated terms, we define now the denotation of the rest of elementary sets.

- $Den(\Lambda) = \emptyset$ (the empty set).

[*1] A type substitution is similar to a substitution that maps variables to type terms. A detailed definition of type substitutions is given in [4].

- If A and B are elementary sets, then $Den(A \otimes B) = Den(A) \cap Den(B)$, $Den(A \oplus B) = Den(A) \cup Den(B)$ and $Den(comp(A)) = \mathcal{H}^n \setminus Den(A)$.

We also 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. A disunification test is represented by a cobasic set (which denotes the complementary set of a subset of $\mathcal{H}^n$).

Definition 4.3 (cobasic set)
A *cobasic set* is an elementary set of the form $comp(\bar{t})$, where $\bar{t}$ is a tuple of terms.

Definition 4.4 (minset)
A *minset* is either Λ or an elementary set of the form $\bar{t}_0 \otimes comp(\bar{t}_1) \otimes \cdots \otimes comp(\bar{t}_n)$, for some $n \geq 0$, where:

- $\bar{t}_0$ is a tuple of terms,
- $comp(\bar{t}_1), \ldots, comp(\bar{t}_n)$ are cobasic sets,
- for all i, $1 \leq i \leq n$, $\bar{t}_i \sqsubset \bar{t}_0$ (which implies that $\bar{t}_i = \bar{t}_0\theta_i$ for some substitution θ_i), and
- for all i, j such that $1 \leq i, j \leq n$ and $i \neq j$, it holds that $\bar{t}_i \not\sqsubseteq \bar{t}_j$.

For brevity, we write a minset of the form $\bar{t}_0 \otimes comp(\bar{t}_1) \otimes \cdots \otimes comp(\bar{t}_n)$ as $\bar{t}_0 \otimes \mathcal{C}$, where $\mathcal{C} = \{comp(\bar{t}_1), \ldots, comp(\bar{t}_n)\}$. The minset representation of a test is given by the *test2minset* function defined below.

Definition 4.5 (exact representation of a test)
A minset β is an exact representation of a test $\tau(\bar{x})$ if $Den(\beta) = Sols(\tau(\bar{x}))$. That is, for any tuple of ground terms $\bar{e}$ it holds that $\bar{e} \in Den(\beta)$ if and only if $\bar{x} = \bar{e} \wedge \tau(\bar{x})$ is satisfiable (i.e., the test $\tau(\bar{e})$ succeeds).

Definition 4.6 (*test2minset* function)
We define the *test2minset*$(\tau(\bar{x}))$ function which takes a test $\tau(\bar{x})$ and returns a

minset β which is an exact representation of $\tau(\bar{x})$. We assume without lost of generality that $\tau(\bar{x})$ is a conjunction of unification and disunification tests and is of the form $\mathcal{E} \wedge \mathcal{D}_1 \wedge \cdots \wedge \mathcal{D}_n$, where $\mathcal{E}$ is the conjunction of all unification tests of $\tau(\bar{x})$ (i.e., a system of equations) and each $\mathcal{D}_i$ a disunification test (i.e., a disequation). The returned value β is computed as follows:

1. Let θ_0 be the substitution associated with the solved form of $\mathcal{E}$ (this can be computed by using the techniques of Lassez et al. [16]).
2. If θ_0 does not exist, then make $\beta = \Lambda$.
3. Otherwise, let θ_i, for $1 \leq i \leq n$, be the substitution associated with the solved form of $\mathcal{E} \wedge \mathcal{N}_i$, where $\mathcal{N}_i$ is the negation of $\mathcal{D}_i$.
4. Let $\beta' = \bar{t}_0 \otimes comp(\bar{t}_1) \otimes \cdots \otimes comp(\bar{t}_n)$, where $\bar{t}_i = \bar{x}\theta_i$, if θ_i exists, for $0 \leq i \leq n$ (if θ_i does not exist, then $comp(\bar{t}_i)$ does not appear in the definition of β').
5. If $\bar{t}_0 \sqsubseteq \bar{t}_i$, for some cobasic set $comp(\bar{t}_i)$, then make $\beta = \Lambda$.
6. Otherwise, perform a simplification step on β' by removing all cobasic sets $comp(\bar{t}_i)$ for which there is a cobasic set $comp(\bar{t}_j)$, $1 \leq i, j \leq n$ and $i \neq j$, such that $\bar{t}_i \sqsubseteq \bar{t}_j$. Make β be the resulting minset.

Theorem 4.1
Let $\tau(\bar{x})$ be a conjunction of unification and disunification tests, and $\beta = test2minset(\tau(\bar{x}))$. Then β is an exact representation of $\tau(\bar{x})$.

Proof

- Since we use the techniques of Lassez et al. [16]) for computing solved forms of systems of equations over Herbrand terms, it follows that if θ_0 does not exist (step 2), then $\mathcal{E}$ is unsatisfiable and hence $\tau(\bar{x})$ also is, thus $\beta = \Lambda$ is an exact representation of $\tau(\bar{x})$.
- For the same reason, if θ_0 exists (step 3), then it is a most general unifier, and thus $\bar{t}_0$ is an exact representation of $\mathcal{E}$. We can prove it because for any tuple of ground terms $\bar{e}$ it holds that if $\bar{e} \in Den(\bar{t}_0)$ then $\bar{e} = \bar{t}_0\theta_e$ for some ground substitution θ_e. Since $\bar{t}_0 = \bar{x}\theta_0$, we have that $\bar{e} = \bar{x}\theta_0\theta_e$. Let $\theta'_e = \theta_e \circ \theta_0$, i.e., $\bar{e} = \bar{x}\theta'_e$. By definition, θ_0 is more general than θ'_e, and thus $\bar{x} = \bar{e} \wedge \mathcal{E}$ is satisfiable. Conversely, if $\bar{x} = \bar{e} \wedge \mathcal{E}$ is satisfiable then $\bar{e} = \bar{x}\theta'_e$ for some ground substitution θ'_e which is more specific than θ_0, i.e., $\theta'_e = \theta_e \circ \theta_0$, thus $\bar{e} \in Den(\bar{t}_0)$.
- In step 4 we have that $Den(\beta') = Den(\bar{t}_0 \otimes comp(\bar{t}_1) \otimes \cdots \otimes comp(\bar{t}_n)) = Den((\bar{t}_0 \otimes comp(\bar{t}_1)) \otimes \cdots \otimes (\bar{t}_0 \otimes comp(\bar{t}_n))) = Den(\bar{t}_0 \otimes comp(\bar{t}_1)) \cap \cdots \cap$

$Den(\bar{t}_0 \otimes comp(\bar{t}_n)) = Sols(\mathcal{E} \wedge \mathcal{D}_1) \cap \cdots \cap Sols(\mathcal{E} \wedge \mathcal{D}_n)) = Sols(\mathcal{E} \wedge \mathcal{D}_1 \wedge \cdots \wedge \mathcal{D}_n) = Sols(\tau(\bar{x}))$.

- In step 5 we have that if $\bar{t}_0 \sqsubseteq \bar{t}_i$, for some cobasic set $comp(\bar{t}_i)$, then $Den(\bar{t}_0) \subseteq Den(\bar{t}_i)$ and $Den(\bar{t}_0 \otimes comp(\bar{t}_i)) = Den(\bar{t}_0) \cap Den(comp(\bar{t}_i)) = \emptyset = Sols(\mathcal{E} \wedge \mathcal{D}_i)$. Thus $Den(\beta) = \emptyset = Sols(\tau(\bar{x}))$.
- In step 6, if $\bar{t}_i \sqsubseteq \bar{t}_j$, then $Den(comp(\bar{t}_i)) \subseteq Den(comp(\bar{t}_j))$ and $Den(\bar{t}_0 \otimes comp(\bar{t}_i)) \cap Den(\bar{t}_0 \otimes comp(\bar{t}_j)) = Den(\bar{t}_0 \otimes comp(\bar{t}_j))$, thus $Den(\beta) = Den(\beta')$.

■

Example 4.3

In order to illustrate the construction of minsets we have created the program below, instead of using the previous quicksort program or a real application. This program exhibits features that can appear in different real cases, and thus allows us to illustrate almost all cases of the algorithm using a single example.

```
p(X1,X2,X3) :- X1 = f(Y1, Y2), Y1 ≠ r(Z1), Y2 ≠ s(Z2) ‖ X3 = a.
p(X1,X2,X3) :- X1 = f(Y1, Y2), Y1 = s(Z1), Y2 ≠ r(Z2) ‖ X3 = b.
```

The guarded program assumes a mode in which the first two arguments of `p/3` are ground and the third one is free. Let the calling type be (α_1, α_1, `-`), where the type symbols α_1 and α_2 are defined as follows:

$$\alpha_1 \rightarrow \{f(\alpha_2, \alpha_2),\ g(\alpha_2, \alpha_2)\} \qquad \alpha_2 \rightarrow \{r(\mu), s(\mu)\}$$

Let us take $\tau(\bar{x})$ in *test2minset*$(\tau(\bar{x}))$ to be the test of the first clause of `p/3`. That is, $\bar{x}$ = `(X1, X2)` and $\tau(\bar{x})$ = τ`(X1, X2)` $\equiv$ `X1 = f(Y1, Y2)` $\wedge$ `Y1` $\neq$ `r(Z1)` $\wedge$ `Y2` $\neq$ `s(Z2)`. We have that τ`(X1, X2)` is written as $\mathcal{E} \wedge \mathcal{D}_1 \wedge \mathcal{D}_2$, where $\mathcal{E} \equiv$ `X1 = f(Y1, Y2)`, $\mathcal{D}_1 \equiv$ `Y1` $\neq$ `r(Z1)` and $\mathcal{D}_2 \equiv$ `Y2` $\neq$ `s(Z2)`. The minset β which represents τ(`X1`, `X2`) is computed as follows:

1. $\theta_0 = \{$`X1 = f(Y1, Y2)`$\}$
2. $\theta_1 = \{$`X1 = f(r(Z1), Y2)`, `Y1 = r(Z1)`$\}$ is the substitution associated with the solved form of `X1 = f(Y1, Y2)` $\wedge$ `Y1 = r(Z1)`, i.e., the system of equations $\mathcal{E} \wedge \mathcal{N}_1$, where $\mathcal{N}_1$ is the negation of `Y1` $\neq$ `r(Z1)`.
3. $\theta_2 = \{$`X1 = f(Y1, s(Z2))`, `Y2 = s(Z2)`$\}$ is the substitution associated with the solved form of `X1 = f(Y1, Y2)` $\wedge$ `Y2 = s(Z2)`.
4. Applying θ_0 to `(X1, X2)` we obtain $\bar{t}_0$, i.e., `(f(Y1, Y2), X2)`. Also, $\bar{x}\theta_1 = \bar{t}_1 =$ `(f(r(Z1), Y2), X2)` and $\bar{x}\theta_2 = \bar{t}_2 =$ `(f(Y1, s(Z2)),`

`X2)`. Thus $\beta' =$ `(f(Y1, Y2), X2)` $\otimes$ *comp*(`(f(r(Z1), Y2), X2)`) $\otimes$ *comp*(`(f(Y1, s(Z2)), X2)`).

5. Finally, the simplification steps does not remove any cobasic set from β', thus $\beta = \beta'$.

If we apply the algorithm to the second clause, we obtain the minset: `(f(s(Z1), Y2), X2)` $\otimes$ *comp*(`(f(s(Z1), r(Z2)), X2)`).

Definition 4.7 (type-annotated term instance)
Let $\delta_1 = (\bar{t}_1, \rho_1)$ and $\delta_2 = (\bar{t}_2, \rho_2)$ be two type-annotated terms. We say that δ_1 is an instance of δ_2 if $\delta_1 \sqsubseteq \delta_2$ and there is a substitution θ such that $\bar{t}_1 = \bar{t}_2\theta$.

Reduction of the Checking Exclusion Problem
Let $\tau_1(\bar{x})$ and $\tau_2(\bar{x})$ be tests which are conjunctions of unification and disunification tests, and ρ a type assignment. Let δ be a type-annotated term representing the type assignment ρ. Let β_i be a minset representing τ_i, for $i = 1, 2$, i.e., $\beta_i =$ *test2minset*(τ_i) (where the *test2minset* function is given in Definition 4.6). We have that $\tau_1(\bar{x})$ and $\tau_2(\bar{x})$ are exclusive w.r.t. ρ if and only if $\delta \otimes \beta_1 \otimes \beta_2 \simeq \Lambda$. Let β be the minset resulting of computing $\beta_1 \otimes \beta_2$ (this intersection can be trivially defined in terms of most general unifiers of the tuples of terms composing the minsets β_1 and β_2). Then, the fundamental problem is to devise an algorithm to test whether $\delta \otimes \beta \simeq \Lambda$, where δ is a type-annotated term and β a minset.

Example 4.4
Consider the mutual exclusion problem for the input tests and calling type given in Example 4.3 for predicate `p/3`. Such calling type is written as the type assignment $((X1, X2) : (\alpha_1, \alpha_1))$, which is represented as the type-annotated term δ, where $\delta = ((X1, X2), (X1 : \alpha_1, X2 : \alpha_1))$. The tests and minsets representing them respectively are:

$\tau_1(\bar{x}) =$ `τ1(X1, X2) ≡ X1 = f(Y1, Y2) ∧ Y1 ≠ r(Z1) ∧ Y2 ≠ s(Z2)`,
$\tau_2(\bar{x}) =$ `τ2(X1, X2) ≡ X1 = f(Y1, Y2) ∧ Y1 = s(Z1) ∧ Y2 ≠ r(Z2)`,
$\beta_1 =$ `(f(Y1, Y2), X2)` $\otimes$ *comp*(`(f(r(Z1), Y2), X2)`)$\otimes$ *comp*(`(f(Y1, s(Z2)), X2)`), and
$\beta_2 =$ `(f(s(Z1), Y2), X2)` $\otimes$ *comp*(`(f(s(Z1), r(Z2)), X2)`).
Thus, $\beta \simeq \beta_1 \otimes \beta_2 \equiv (f(s(X_3), X_4), X_5) \otimes comp(f(s(X_6), s(X_7)), X_8) \otimes comp(f(s(X_9), r(X_{10})), X_{11})$.

A High Level Description of the Algorithm

We first provide a high level description of the algorithm that we propose, whose detailed description is given by the boolean function $empty(\delta, \beta)$ in Definitions 4.13, 4.14 and 4.15:[*2]

- First, perform the "intersection" of the type-annotated term δ and the tuple of terms $\bar{t}_0$ of the minset β (we assume that $\beta = \bar{t}_0 \otimes \mathcal{C}$ and that $\beta \not\simeq \Lambda$). Let δ' denote the type-annotated term resulting from this intersection (i.e., $\delta' = \delta \otimes \bar{t}_0$). This operation is implemented by the $intersec(\delta, \bar{t}_0)$ function described in Definition 4.10 (recall that a tuple of terms is a type-annotated term). Consider for example δ and β given in Example 4.4. In this case, $\bar{t}_0$ denotes the tuple of terms $(f(s(X_3), X_4), X_5)$ and $\mathcal{C}$ denotes the set of cobasic sets $\{comp(f(s(X_6), s(X_7)), X_8), comp(f(s(X_9), r(X_{10})), X_{11})\}$. Thus, the "intersection" of δ and $\bar{t}_0$ is the type-annotated term $\delta' = ((f(s(X_{12}), X_{13}), X_{14}), (X_{12} : \mu, X_{13} : \alpha_2, X_{14} : \alpha_1)$.
- If δ' is empty (i.e., $\delta' \simeq \Lambda$), then it can be reported that $\delta \otimes \beta \simeq \Lambda$. Otherwise, if $\bar{t}_0$ is "included" in δ' (i.e., $\bar{t}_0 \sqsubseteq \delta'$), then it can be reported that $\delta \otimes \beta \not\simeq \Lambda$ (note that it always holds that $\beta \sqsubseteq \bar{t}_0$). The "inclusion" operation can be defined by using a straightforward adaptation of the $subset_T(\omega_1, \omega_2)$ function described in [4], 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. We denote our "inclusion" operation by the $included(\delta_1, \delta_2)$ function, which returns **true** if and only if $\delta_1 \sqsubseteq \delta_2$, where δ_1 and δ_2 are type-annotated terms. In our example, none of these conditions hold (recall that the tuple of terms $(f(s(X_3), X_4), X_5)$ represents the type-annotated term $((f(s(X_3), X_4), X_5), (X_3 : \mu, X_4 : \mu, X_5 : \mu))$, and that a type-annotated term can be trivially rewritten as a pure type term).
- Otherwise, the problem is reduced to checking whether $\delta' \otimes \mathcal{C} \simeq \Lambda$ (this is done by the auxiliary function *empty1*, described in detail in Definition 4.14). Note that $\delta' \otimes \mathcal{C}$ can be seen as a system of one equation (corresponding to δ') and zero or more disequations (each of them corresponding to a cobasic set in $\mathcal{C}$). Thus the problem can be seen as determining whether such system has no solutions.

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

- This way, if δ' is "included" in the tuple of terms of some cobasic set in $\mathcal{C}$, then it can be reported that $\delta' \otimes \mathcal{C} \simeq \Lambda$.
- Otherwise, it means that δ' is "too big", and thus, it is "expanded" to a set of "smaller" type-annotated terms (with the hope that each of them will be "included" in the tuple of terms of some cobasic set in $\mathcal{C}$). This way, the initial problem is reduced to a finite number of subproblems, one subproblem for each element in the set of "smaller" type-annotated terms to which δ' has been "expanded". This holds in the example, where the type-annotated term δ' is "expanded" to a set of two "smaller" type-annotated terms $\{\delta'_1, \delta'_2\}$ (expanding variable X_{13}) where δ'_1 denotes $((f(s(X_{15}), r(X_{16})), X_{17}), (X_{15} : \mu, X_{16} : \mu, X_{17} : \alpha_1))$ and δ'_2 denotes $((f(s(X_{18}), s(X_{19})), X_{20}), (X_{18} : \mu, X_{19} : \mu, X_{20} : \alpha_1))$. Then, two subproblems arise:
  - Checking whether $\delta'_1 \otimes comp(f(s(X_6), r(X_7)), X_8) \simeq \Lambda$, which holds because δ'_1 is "included" in $(f(s(X_6), r(X_7)), X_8)$, the tuple of terms of the cobasic set $comp(f(s(X_6), r(X_7)), X_8)$; and
  - Checking whether $\delta'_2 \otimes comp(f(s(X_9), r(X_{10})), X_{11}) \simeq \Lambda$ is empty, which also holds because δ'_2 is "included" in $(f(s(X_9), r(X_{10})), X_{11})$.
- Thus, it can be concluded that $\delta' \otimes \mathcal{C} \simeq \Lambda$ and hence $\delta \otimes \beta \simeq \Lambda$.

Termination of this algorithm is granted because a) the original problem is divided into a finite number of subproblems, since the type-annotated term of the problem is expanded into a finite number of type-annotated terms, each one giving rise to a subproblem, b) the number of cobasic sets in the initial problem is finite, and c) the number of cobasic sets decreases at least by one in each subproblem (recursive call).

A Detailed Description of the Algorithm

The detailed description of the $empty(\delta, \beta)$ function requires some (instrumental) definitions, namely the definition of "useless" cobasic set and the *aliased*, *intersec*, and *expansion* functions.

Definition 4.8 (useless cobasic set)

Given a type-annotated term δ, a set of cobasic sets $\mathcal{C}$, and a cobasic set $comp(\bar{t}) \in \mathcal{C}$, we say that $comp(\bar{t})$ is *useless* for determining whether $\delta \otimes \mathcal{C} \simeq \Lambda$, whenever if $\delta \otimes (\mathcal{C} - \{comp(\bar{t})\}) \not\simeq \Lambda$, then $\delta \otimes \mathcal{C} \not\simeq \Lambda$ (or, equivalently, if $\delta \otimes \mathcal{C} \simeq \Lambda$, then $\delta \otimes (\mathcal{C} - \{comp(\bar{t})\}) \simeq \Lambda$).

It is easy to prove that the reciprocal also holds. If $\delta \otimes (\mathcal{C} - \{comp(\bar{t})\}) \simeq \Lambda$, then obviously $\delta \otimes \mathcal{C} \simeq \Lambda$ (note that $(\delta \otimes \mathcal{C}) \sqsubseteq (\delta \otimes (\mathcal{C} - \{comp(\bar{t})\}))$). Thus, if $comp(\bar{t}) \in \mathcal{C}$ is an useless cobasic set, then $\delta \otimes \mathcal{C} \simeq \Lambda$ if and only if $\delta \otimes (\mathcal{C} - \{comp(\bar{t})\}) \simeq \Lambda$.

Definition 4.9 (*aliased*$(\delta, \bar{t})$ function)
Let δ be the type-annotated term $(\bar{t}_\delta, \rho_\delta)$, $\delta \not\simeq \Lambda$, $\bar{t}$ a tuple of terms, and $\theta = mgu(\bar{t}_\delta, \bar{t})$. We define the *aliased* function as follows:

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

Given a type-annotated term δ and a tuple of terms $\bar{t}$, the $intersec(\delta, \bar{t})$ function returns a type-annotated term whose meaning is the same as $\delta \otimes \bar{t}$ (recall that a tuple of terms is also a type-annotated term). This function can be defined as a straightforward adaptation of the $unify(\omega_1, \omega_2, T, \Theta)$ function described in [4], 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 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. The 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 in ω_f, 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 function $intersec(\delta, \bar{t})$ as follows:

Definition 4.10 ($intersec(\delta, \bar{t})$ function)
Given a type-annotated term δ and a tuple of terms $\bar{t}$, the process of function $intersec(\delta, \bar{t})$ is:

- Let δ be the pair $(\bar{t}_\delta, \rho_\delta)$, and $(\omega_f, T_f, \Theta_f) = unify(\bar{t}_\delta, \bar{t}, T, \Theta)$ (note that a tuple of terms is a particular case of type term, and that $\bar{t}_\delta$ and $\bar{t}$ are tuples of terms), where Θ is a type substitution constructed as follows:
$$x\Theta = \begin{cases} \omega & \text{if } x \in vars(\delta) \text{ and } \rho_\delta(x) = \omega \\ x & \text{otherwise.} \end{cases}$$
and T a set of type rules defining the type symbols in $\bar{t}_\delta\Theta$.
- Rewrite $\omega_f\Theta_f$ as a type-annotated term δ' and return it. For simplicity, we assume that the function returns only a type-annotated term δ', but in fact it returns a pair (δ', T_f), where T_f is a set of type rules defining the type symbols appearing in δ'.

Theorem 4.2
Given a type-annotated term δ and a tuple of terms $\bar{t}$, then: (i) $intersec(\delta, \bar{t})$ terminates, (ii) $intersec(\delta, \bar{t}) \simeq \delta \otimes \bar{t}$, and (iii) $intersec(\delta, \bar{t}) = \Lambda$ iff $\delta \otimes \bar{t} \simeq \Lambda$.

Proof It follows from Theorem 5.60 of [4], since the function *intersec* is an adaptation of the function $unify(\omega_1, \omega_2, T, \Theta)$. ■

The expansion of a type-annotated term into a set of smaller type-annotated terms is performed by the *expansion* function defined below.

Definition 4.11 (*expansion*$(\delta, comp(\bar{t}))$ function)
Let δ be a type-annotated term $(\bar{t}_\delta, \rho_\delta)$, $\delta \not\simeq \Lambda$, and $comp(\bar{t})$ a cobasic set such that $\delta \otimes \bar{t} \not\simeq \Lambda$ and $\delta \not\sqsubseteq \bar{t}$. We also assume that $vars(t_\delta) \cap vars(\bar{t}) = \emptyset$. The function $expansion(\delta, comp(\bar{t}))$ returns a pair (δ', Δ) which is a "partition" of δ, i.e.:

- δ' is a type-annotated term instance of δ, $(\bar{t}_{\delta'}, \rho_{\delta'})$, $\delta' \not\simeq \Lambda$. δ' is obtained by expanding δ to some "decision depth" that allows to detect if the cobasic set $comp(\bar{t})$ is useless (see Definition 4.8 of useless cobasic set);
- Δ is a set of type-annotated terms;
- for all $x \in vars(\delta')$, it holds that: $\rho_{\delta'}(x) = \mu$, $\rho_\delta(x)$ is a base type symbol, or $x\theta$ is a variable, where θ is the most general unifier of $\bar{t}_{\delta'}$ and $\bar{t}$ (note that the variables of δ whose type is μ or a base type are not "expanded");
- $(\cup_{\delta'' \in \Delta} Den(\delta'')) \cup Den(\delta') = Den(\delta)$ (i.e., $\delta \simeq (\bigoplus_{\delta'' \in \Delta} \delta'') \oplus \delta'$); and
- for all $\delta'' \in \Delta$, $\delta'' \otimes \bar{t} \simeq \Lambda$ (this is ensured because type rules are deterministic).

We define the *expansion* function as:

$$expansion(\delta, comp(\bar{t})) = expands(vars(t_\delta), mgu(\bar{t}_\delta, \bar{t}), \delta, \emptyset)$$

where the *expands* function is defined below:

Definition 4.12 (*expands*$(V, \theta, \delta, \Delta)$ function)
Let δ be a type-annotated term $(\bar{t}_\delta, \rho_\delta)$, V a set of variables $V \subseteq vars(t_\delta)$, θ a substitution for the variables in t_δ, and Δ a set of type-annotated terms which are pairwise disjoint and disjoint with δ. The *expands* function returns a pair (δ', Δ'), where δ' is a type-annotated term and Δ' a set of type-annotated terms, and is defined by the following algorithm:

1: $V' \leftarrow \{x \in V \mid x\theta$ is not a variable, $\rho_\delta(x) \neq \mu$, and $\rho_\delta(x)$ is not a base type symbol$\}$

2: **if** $V' = \emptyset$ **then return** the pair (δ, Δ)

3: **else**

4: Take a variable $x \in V'$

5: Let $x\theta = f(t_1, \ldots, t_n)$, where $n \geq 0$ and $t_1, \ldots, t_n$ are terms (if $x\theta$ is a constant, it is treated as the particular case where $n = 0$).

6: $\alpha \leftarrow \rho_\delta(x)$

7: **if** α is a type symbol **then**

8: Let the type rule defining α be $\alpha \rightarrow \Upsilon$

9: Let $\omega \in \Upsilon$, such that $\omega = f(\alpha_1, \ldots, \alpha_n)$, i.e., ω has the same principal function symbol (and arity) than $x\theta$ (note that such ω always exists, since $\delta \otimes \bar{t} \not\geq \Lambda$, and type rules are deterministic.)

10: $\Upsilon' \leftarrow \Upsilon - \{\omega\}$

11: **else** (necessarily α is a pure type term of the form $f(\alpha_1, \ldots, \alpha_n)$)

12: $\omega \leftarrow \alpha$

13: $\Upsilon' \leftarrow \emptyset$

14: **end if**

15: $\bar{t}'_\delta \leftarrow \bar{t}_\delta[x/f(y_1, \ldots, y_n)]$, where $y_1, \ldots, y_n$ are new and distinct variables, and $\bar{t}_\delta[y/t]$ denotes the instance of the tuple of terms $\bar{t}_\delta$ obtained by substituting all occurrences of variable y by term t in $\bar{t}_\delta$

16: $\rho'_\delta \leftarrow (\rho_\delta - \{x : \alpha\}) \cup \{y_1 : \alpha_1, \ldots, y_n : \alpha_n\}$, i.e., ρ'_δ is the type assignment obtained by removing $x : \alpha$ from ρ_δ, and adding $y_1 : \alpha_1, \ldots, y_n : \alpha_n$ to the result

17: $\delta' \leftarrow (\bar{t}'_\delta, \rho'_\delta)$

18: $V'' \leftarrow (V' - \{x\}) \cup \{y_1, \ldots, y_n\}$

19: $\theta' \leftarrow (\theta - \{x = x\theta\}) \cup \{y_1 = t_1, \ldots, y_n = t_n\}$

20: $\Delta' \leftarrow (\cup_{\omega' \in \Upsilon'} \{(\bar{t}_\delta, \rho_\delta^{\omega'})\}) \cup \Delta$

where $\rho_\delta^{\omega'}$ is the type assignment obtained by removing $x : \alpha$ from ρ_δ, and adding $x : \omega'$ to ρ_δ.

21: **return** *expands*$(V'', \theta', \delta', \Delta')$

22: **end if**

Example 4.5

Reconsider the type annotated term:

$$\delta' = ((f(s(X_{12}), X_{13}), X_{14}), (X_{12} : \mu, X_{13} : \alpha_2, X_{14} : \alpha_1)$$

and the set of cobasic sets:

$$\mathcal{C} = \{comp(f(s(X_6), s(X_7)), X_8), comp(f(s(X_9), r(X_{10})), X_{11})\}$$

of Example 4.4.
Let us choose the cobasic set $comp(\bar{t}) = comp(f(s(X_6), s(X_7)), X_8)$ from $\mathcal{C}$ (note that δ' is not included in the tuple of terms $\bar{t} = (f(s(X_6), s(X_7)), X_8))$. We expand δ' by calling the *expansion* function (in Definition 4.11) as follows:

$expansion(\delta', comp(\bar{t})) = expands(vars(t_{\delta'}), mgu(\bar{t}_{\delta'}, \bar{t}), \delta', \emptyset) =$
$expands(\{X_{12}, X_{13}, X_{14}\}, \{X_{12} = X_6, X_{13} = s(X_7), X_{14} = X_8\}, \delta', \emptyset) = (\delta_1, \Delta_1)$

where $\delta_1 = ((f(s(X_{15}), r(X_{16})), X_{17}), (X_{15} : \mu, X_{16} : \mu, X_{17} : \alpha_1))$ and
$\Delta_1 = \{((f(s(X_{18}), s(X_{19})), X_{20}), (X_{18} : \mu, X_{19} : \mu, X_{20} : \alpha_1))\}$. This is done by choosing variable X_{13} in step 4 of the *expands* function (see Definition 4.12) and using its type, α_2, in step 7.

Definition 4.13 (*empty*(δ, β) function)
Given a type-annotated term δ and a minset β such that $\beta \not\simeq \Lambda$ ($\beta = \bar{t}_0 \otimes \mathcal{C}$, where $\bar{t}_0$ is a tuple of terms, and $\mathcal{C}$ a set of cobasic sets), we define:

$$empty(\delta, \beta) = \begin{cases} \textbf{true} & \text{if } \delta' = \Lambda \\ \textbf{false} & \text{if } included(\bar{t}_0, \delta') \\ empty1(\mathcal{C}, \delta', \emptyset) & \text{otherwise} \end{cases}$$

where $\delta' = intersec(\delta, \bar{t}_0)$.

Definition 4.14 (*empty1*$(\mathcal{C}, \delta, \Gamma)$ function)
Given a type-annotated term δ (i.e., a pair $(\bar{t}_\delta, \rho_\delta)$) such that $\delta \not\simeq \Lambda$, a set of cobasic sets $\mathcal{C}$, and a set Γ of triples of the form $(\delta_1, \mathcal{V}, comp(\bar{t}))$ where:

- δ_1 is a type-annotated term $\delta_1 = (\bar{t}_1, \rho_1)$, such that $\delta_1 \not\simeq \Lambda$,
- $comp(\bar{t})$ is a cobasic set,
- $vars(\delta_1) \cap vars(comp(\bar{t})) = \emptyset$,
- $\theta = mgu(\bar{t}_1, \bar{t})$,
- for all $x \in vars(\delta_1)$, $x\theta$ is a variable, and
- $v \in \mathcal{V}$ iff $v \in vars(\delta_1)$, $\rho_{\delta_1}(v) \neq \mu$, $\rho_{\delta_1}(v)$ is not a base type symbol, and $\exists v' \in vars(\delta_1)$, $v \neq v'$, such that $v\theta = v'\theta$ (i.e., $\mathcal{V}$ is the set of variables in $vars(\delta_1)$ which are aliased with some other variable in $vars(\delta_1)$ by θ).

we define the *empty1* function in Algorithm 1.

The *empty1*$(\mathcal{C}, \delta, \Gamma)$ function performs a "first pass" over the cobasic sets in $\mathcal{C}$. This pass results in the removal of cobasic sets that are inferred to be useless. Some useless cobasic sets are removed in step 1: if $intersec(\delta, \bar{t}) = \Lambda$, for

Algorithm 1 *empty1*

Input: a type-annotated term δ, a set of cobasic sets $\mathcal{C}$ and a set Γ of triples of the form $(\delta_1, \mathcal{V}, comp(\bar{t}))$

Output: **true** if $\delta \otimes \mathcal{C}_1 \simeq \Lambda$, where $\mathcal{C}_1 = \mathcal{C} \cup \{comp(\bar{t}) \mid (\delta_1, \mathcal{V}, comp(\bar{t})) \in \Gamma\}$. **false** otherwise.

1: $\mathcal{C}' \leftarrow \{comp(\bar{t}) \in \mathcal{C} \mid \ intersec(\delta, \bar{t}) \not\simeq \Lambda\}$
2: **if** $\mathcal{C}' = \emptyset$ **then**
3: $\quad \Gamma' \leftarrow \ \{\xi \in \Gamma \mid \xi \equiv (\delta_1, \mathcal{V}, comp(\bar{t})),\ intersec(\delta, \bar{t}) \not\simeq \Lambda\}$
4: $\quad \Gamma'' \leftarrow \ \{\xi \in \Gamma' \mid \xi \equiv (\delta_1, \mathcal{V}, comp(\bar{t})),\ \theta = mgu(\bar{t}_\delta, \bar{t}_{\delta_1})$, and for all $x \in \mathcal{V}$, $y \in vars(x\theta)$: $\rho_\delta(y)$ is finite $\}$
5: $\quad$ **return** *empty2*(Γ'', δ)
6: **else if** *included*$(\delta, \bar{t})$ for some cobasic set $comp(\bar{t})$ in $\mathcal{C}'$ **then**
7: $\quad$ **return true**
8: **else**
9: $\quad$ select a cobasic set $comp(\bar{t}) \in \mathcal{C}'$
10: $\quad \mathcal{C}'' \leftarrow \mathcal{C}' - \{comp(\bar{t})\}$
11: $\quad (\delta', \Delta) \leftarrow expansion(\delta, comp(\bar{t}))$
12: $\quad$ **if** *included*$(\delta', \bar{t})$ **then**
13: $\quad\quad$ **return** $\bigwedge_{\delta'' \in \Delta}$ *empty1*$(\mathcal{C}'', \delta'', \Gamma)$
14: $\quad$ **else**
15: $\quad\quad \mathcal{V} \leftarrow aliased(\delta', \bar{t})$
16: $\quad\quad \theta' \leftarrow mgu(\bar{t}_{\delta'}, \bar{t})$
17: $\quad\quad$ **if** for some $x \in vars(\delta')$ s.t. $\rho_{\delta'}(x) = \mu$ or $\rho_{\delta'}(x)$ is a base type symbol: $x \in \mathcal{V}$ or $x\theta'$ is not a var. **then**
18: $\quad\quad\quad$ **return** *empty1*$(\mathcal{C}'', \delta, \Gamma)$
19: $\quad\quad$ **else**
20: $\quad\quad\quad \Gamma' \leftarrow \Gamma \cup \{(\delta', \mathcal{V}, comp(\bar{t}))\}$
21: $\quad\quad\quad$ **return** *empty1*$(\mathcal{C}'', \delta', \Gamma') \wedge \bigwedge_{\delta'' \in \Delta}$ *empty1*$(\mathcal{C}'', \delta'', \Gamma)$
22: $\quad\quad$ **end if**
23: $\quad$ **end if**
24: **end if**

some $comp(\bar{t}) \in \mathcal{C}$, then $comp(\bar{t})$ is useless for determining whether $\delta \otimes \mathcal{C} \simeq \Lambda$, because none of the instances of δ meet the equality constraint imposed by $\bar{t}$, and hence all the instances of δ meet the inequality constraint imposed by $comp(\bar{t})$. Thus, $\delta \otimes \mathcal{C} \simeq \Lambda$ if and only if the rest of cobasic sets, $\mathcal{C} - \{comp(\bar{t})\}$, impose (inequality) constraints that are not met by any instance of δ. If $included(\delta, \bar{t})$ for some cobasic set $comp(\bar{t})$ in $\mathcal{C}'$ (as it is checked in step 6), then all the instances of δ meet the equality constraint imposed by $\bar{t}$, and hence, none of the instances of δ meet the inequality constraint imposed by $comp(\bar{t})$. Thus, in this case, $\delta \otimes \mathcal{C} \simeq \Lambda$. The rationale behind steps 9 to 11 is that at this point (where not $included(\delta, \bar{t})$ nor $intersec(\delta, \bar{t}) = \Lambda$) δ is "too big," and thus it is "expanded" to a set of "smaller" type-annotated terms $\{\delta'\} \cup \Delta$ (using the *expansion* function given in definition 4.11), in the hope that each of them will be "included" in the tuple of terms of some cobasic set in $\mathcal{C}'$. In this expansion, δ' is obtained by expanding variables $v \in vars(\delta)$ to at most a depth given by $v\sigma$, where $\sigma = mgu(\bar{t}_\delta, \bar{t})$. When inclusion is checked at step 12, if $included(\delta', \bar{t})$, then necessarily for all $x \in vars(\delta')$ it holds that $x\theta'$ is a variable, where $\theta' = mgu(\bar{t}_{\delta'}, \bar{t})$ (step 16). In this case, $comp(\bar{t})$ is not considered in the recursive calls in step 13 since (according to definition 4.11) for all $\delta'' \in \Delta$, $\delta'' \otimes \bar{t} \simeq \Lambda$, and thus, $comp(\bar{t})$ is useless for all of these subproblems. If not $included(\delta', \bar{t})$, then: a) $\bar{t}$ imposes some equality constraints over some variables in δ (such variables are gathered together in step 15, where the set $\mathcal{V}$ is created using the *aliased* function given in Definition 4.9), or b) $\bar{t}$ restricts the values of some variable(s) in δ' whose type is μ, unifying them to some term (which is not a variable). If the condition checked at step 17 holds, then there is always an instance of δ' which does not meet the former constraints a) or b), and thus $comp(\bar{t})$ is useless. In step 20, cobasic sets which are not deemed useless at this point are stored in Γ, which is an accumulation parameter. δ' and $\mathcal{V}$ (besides $comp(\bar{t})$) are recorded in this parameter, because aliased variables whose type is infinite (or which after having been expanded are bound to a term containing variables whose type is infinite) allow us to detect useless cobasic sets, since it is always possible to find an instance of δ' which does not meet the equality constraints imposed by $\bar{t}$ (case a)). Useless cobasic sets are then subsequently removed in steps 3 and 4, before $empty2(\Gamma', \delta)$ is called in step 5. The first pass over the cobasic sets ends in step 2 when condition $\mathcal{C}' = \emptyset$ holds. Note that when this condition holds, step 4 checks that a type expression denotes a finite set of terms, and there are straightforward algorithms to test this. The *empty2* function performs a second pass over the

Algorithm 2 *empty2*

Input: a type-annotated term δ and a set Γ of triples of the form $(\delta', \mathcal{V}, comp(\bar{t}))$

Output: a boolean

1: **if** $\Gamma = \emptyset$ **then**
2: **return false**
3: **else**
4: select an item $\xi \in \Gamma$; assume $\xi \equiv (\delta', \mathcal{V}, comp(\bar{t}))$
5: $\Gamma' \leftarrow \Gamma - \{\xi\}$
6: $\sigma \leftarrow mgu(\bar{t}_{\delta'}, \bar{t}_{\delta})$
7: **if** $included(\delta, \bar{t})$ **then**
8: **return true**
9: **else**
10: initialize a set Δ
11: **for all** variables $x \in \mathcal{V}$ **do**
12: **for all** variables y such that $y \in vars(x\sigma)$ **do**
13: $\Delta \leftarrow \Delta \cup \{\ \delta[y/t] \mid t \in \gamma(\rho_\delta(y))\ \}$
14: **end for**
15: **end for**
16: $\Delta' \leftarrow \{\delta'' \in \Delta \mid intersec(\delta'', \bar{t}) \simeq \Lambda\}$
17: **if** $\Delta' = \emptyset$ **then**
18: **return true**
19: **else**
20: **return** $\bigwedge_{\delta'' \in \Delta'} empty2(\Gamma', \delta'')$
21: **end if**
22: **end if**
23: **end if**

remaining cobasic sets, checking whether the constraints described previously in case a) hold. Since the types of the variables involved in such constraints are finite (i.e., they represent finite sets of terms), the process performed by the *empty2* function is simple, sound, complete, and terminating.

Definition 4.15 (*empty2*(Γ, δ) function)
Given a type-annotated term δ, such that $\delta \not\simeq \Lambda$, and a set Γ of triples of the form $(\delta_1, \mathcal{V}, comp(\bar{t}))$ similar to those in the third parameter of the function *empty1*$(\mathcal{C}, \delta, \Gamma)$ in definition 4.14, but with the following additional constraint:

for all $x \in \mathcal{V}$, $\rho_{\delta_1}(x)$ is finite (note that for all $v \in vars(\delta_1)$ such that $v \notin \mathcal{V}$, $\rho_{\delta_1}(v)$ can be any type, including μ or a base type symbol),

we define the function *empty2* in Algorithm 2, where $\delta[y/t]$ denotes an instance of type annotated term δ obtained by substituting variable y by term t.

The function $empty2(\Gamma, \delta)$ selects a cobasic set $comp(\bar{t})$ in Γ, and, if δ is not included in $\bar{t}$, then δ is expanded (in step 13) to a set of type-annotated terms Δ by substituting only "decision variables." Such expansion ensures that every $\delta'' \in \Delta$ is either "included" in $\bar{t}$ or "disjoint" with it. It also ensures that δ is not infinitely expanded: the type of such variables is finite. Note that, at step 13, necessarily $y \in vars(\delta)$, and $\rho_\delta(y)$ is finite. Note also that, at step 16, necessarily, for all $\delta'' \in \Delta$ and $\delta'' \notin \Delta'$, it holds that $\delta'' \sqsubseteq \bar{t}$. For this reason, $comp(\bar{t})$ is removed from the recursive call at step 20.

Soundness and Completeness Results

The function $empty(\delta, S)$ is sound and complete for *tuple-distributive regular types* (a detailed proof is given in [18]). While sound, the function is not complete for regular types in general. However, our experience (as we will see in Section 5) 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 [11].

Theorem 4.3

Let δ be a type-annotated term such that all types appearing in it are *tuple-distributive regular types*, and β a minset with the conditions of definitions 4.13, 4.14, and 4.15. Let also functions *empty*, *empty1*, and *empty2* defined there. We have that:

1. *empty*, *empty1*, and *empty2* terminate.
2. $empty2(\Gamma, \delta) = \mathbf{true}$ if and only if $\delta \otimes \mathcal{C} \simeq \Lambda$, where $\mathcal{C} = \{comp(\bar{t}) \mid (\delta', \mathcal{V}, comp(\bar{t})) \in \Gamma$ for some δ' and $\mathcal{V}\}$ (i.e., $\mathcal{C}$ is the set of cobasic sets in Γ).
3. $empty1(\mathcal{C}, \delta, \Gamma) = \mathbf{true}$ if and only if $\delta \otimes \mathcal{C}_1 \simeq \Lambda$, where $\mathcal{C}_1 = \mathcal{C} \cup \{comp(\bar{t}) \mid (\delta_1, \mathcal{V}, comp(\bar{t})) \in \Gamma$ for some δ_1 and $\mathcal{V}\}$.
4. $empty(\delta, \beta) = \mathbf{true}$ if and only if $\delta \otimes \beta \simeq \Lambda$.

4.2 Checking Mutual Exclusion in Linear Arithmetic

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 `int` 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 in equation (1) below, where each of the tests $\phi_{kl}(\bar{x})$ is of the form $\phi_{kl}(\bar{x}) \equiv a_0 + a_1x_1 + \cdots + a_px_p \ ? \ 0$, with $? \in \{=, <, \leq, >, \geq\}$. For this transformation, note that a test $a_0 + a_1x_1 + \cdots + a_px_p \neq 0$ can be written in terms of two tests involving only '>' and '<', as in equation (2).

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

$$(\sum_{i=0}^{p} a_ix_i > 0) \ \vee \ (\sum_{i=0}^{p} a_ix_i < 0) \qquad (2)$$

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 [8], the following complexity result is immediate:

Theorem 4.4
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.

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. Assume, without loss of generality, that each $\tau_k(\bar{x})$, $1 \leq k \leq n$ is a conjunction of primitive tests (note that it is always possible to obtain an equivalent sequence of clauses where disjunctions have been removed). Assume also that each $\tau_k(\bar{x})$, $1 \leq k \leq n$, is written as $\tau_k^H \wedge \tau_k^A$, where τ_k^H and τ_k^A are a conjunction of primitive unification and arithmetic tests respectively (i.e., we write arithmetic tests after unification tests). Consider also each τ_k^H written as a minset β_k (the function *test2minset*, given in Definition 4.6, returns the minset representation of a test).

Assume that predicate p has type $\mathsf{type}[p]$. In order to check whether 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 any pair of tests $\tau_i(\bar{x})$ and $\tau_j(\bar{x})$, $1 \leq i, j \leq n$, $i \neq j$, is exclusive w.r.t. $\bar{x} : \mathsf{type}[p]$.

Before describing a sufficient condition for ensuring that these tests are exclusive, we define some instrumental elements. Let β_{ij} be the minset intersection of β_i and β_j. Let θ_i (resp. θ_j), be the most general unifier of the tuple of terms of β_{ij} and β_i (resp. β_j). That is, if $\beta_i \equiv \bar{t}_i \otimes \mathcal{C}_i$, $\beta_j \equiv \bar{t}_j \otimes \mathcal{C}_j$, and $\beta_{ij} \equiv \bar{t}_{ij} \otimes \mathcal{C}_{ij}$, then $\theta_i = mgu(\bar{t}_i, \bar{t}_{ij})$, $\bar{t}_{ij} \equiv \bar{t}_i\theta_i$, $\theta_j = mgu(\bar{t}_j, \bar{t}_{ij})$, $\bar{t}_{ij} \equiv \bar{t}_j\theta_j$ (note that there exists a substitution μ_{ij}, such that $\mu_{ij} = mgu(\bar{t}_i, \bar{t}_j)$). Let ρ be the type assignment $\bar{x} : \mathsf{type}[p]$ but written as a type-annotated term δ. We have that the tests $\tau_i(\bar{x})$ and $\tau_j(\bar{x})$, are exclusive w.r.t. ρ if:

1. $\delta \otimes \beta_i \otimes \beta_j \simeq \Lambda$ (which can be checked as explained in Section 4.1), or
2. $\delta \otimes \beta_i \otimes \beta_j \not\simeq \Lambda$ and $\tau_i^A\theta_i \wedge \tau_j^A\theta_j$ is unsatisfiable (which can be checked as explained in Section 4.2).

Example 4.6

Reconsider Example 4.1 with predicate `part`/4 from the quicksort program of Figure 1. We had reduced the mutual exclusion problem to two subproblems: a) checking that the tests $\mathtt{L} = [\,]$ and $\mathtt{L} = [\mathtt{E}|\mathtt{R}]$ are exclusive w.r.t. type assignment ρ, and b) checking that the tests $\mathtt{E} < \mathtt{C}$ and $\mathtt{E} >= \mathtt{C}$ are exclusive w.r.t. ρ. In this case, we have that δ is $((\mathtt{L}, \mathtt{C}), (\mathtt{L} : \mathtt{intlist}, \mathtt{C} : \mathtt{int}))$. Also, $\beta_1 \equiv ([\,], C)$, $\beta_2 \equiv ([\mathtt{E}|\mathtt{R}], \mathtt{C})$, and $\beta_3 \equiv ([\mathtt{E}|\mathtt{R}], \mathtt{C})$. We now have that `part`/4 is mutually exclusive because: $\delta \otimes \beta_i \otimes \beta_j \simeq \Lambda$, for $i = 1$ and $j \in \{2, 3\}$, and (although $\delta \otimes \beta_2 \otimes \beta_3 \not\simeq \Lambda$) also $\mathtt{E} < \mathtt{C} \wedge \mathtt{E} \geq \mathtt{C}$ is unsatisfiable (note that $\beta_{2,3} \equiv ([\mathtt{E}|\mathtt{R}], \mathtt{C})$, and θ_2 and θ_3 are the identity).

4.4 Checking Mutual Exclusion: Dealing with the Cut

The presence of a pruning operator (cut) in program clauses can help the detection of mutual exclusion. In order to take the cut into account, we simply redefine the concept of mutually exclusive clauses in Definition 2.4 as:

Definition 4.16 (mutual exclusion in the presence of cut)
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, $1 \leq i, j \leq n$, $i \neq j$: C_i has a cut and $i < j$, or C_j has a cut and $j < i$, or $\tau_i(\bar{x})$ 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 a sequence of n clauses $C_1, \ldots, C_n$, with input tests $\tau_1(\bar{x}), \ldots, \tau_n(\bar{x})$ respectively. 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 4.16, we take the test τ_i^c defined as follows:

$$\tau_i^c = (\bigwedge_{k \in I} \neg \tau_k^b) \wedge \tau_i$$

Example 4.7
Consider predicate `abs`/2 mentioned in page 8. Usually, this predicate is defined with a cut in the first clause and no check in the second. In this case, the test for the second clause will be $\neg$ `X` $\geq$ `0`.

Note that the introduction of 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 or more exclusion subproblems where the pair of tests involved in each subproblem are conjunctions of primitive tests (transforming tests to disjunctive normal form).

§5 A Prototype Implementation

In order to evaluate the effectiveness and efficiency of our approach to determinacy analysis we have constructed a prototype which performs such anal-

ysis in an automatic way. The system takes Prolog programs as input,[*3] which include a module definition in the standard way. In addition, the types and modes of the arguments of exported predicates are either given 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 [21], and the *eterms* domain to derive the types of predicates [25]. 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 [22]. 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, as mentioned in Section 1, 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 them 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/CiaoPP version 1.13, rev 10683, on an Intel Pentium M 1.86GHz, 1Gb of RAM memory, running Ubuntu Linux 8.04, 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 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

[*3] 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.

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

Program	**N**	**D (%)**	**M (%)**	**C**	$\mathbf{T}_D$	$\mathbf{T}_M$	$\mathbf{T}_T$
Hanoi	2	2 (100)	2 (100)	N/A	48	55	103
Fib	1	1 (100)	1 (100)	N/A	16	21	37
Mmatrix	3	3 (100)	3 (100)	N/A	24	39	63
Tak	1	1 (100)	1 (100)	N/A	24	23	47
Subs	1	1 (100)	1 (100)	N/A	12	16	28
Reverse	2	2 (100)	2 (100)	N/A	21	20	41
Qsort	3	3 (100)	3 (100)	3 (100)	40	34	74
Qsort2	5	5 (100)	5 (100)	5 (100)	64	43	107
Queens	6	3 (50)	5 (83)	2 (33)	65	36	101
Gabriel	20	6 (30)	11 (55)	4 (20)	308	241	549
Kalah	44	40 (91)	42 (95)	40 (91)	952	2432	3384
Plan	16	8 (50)	12 (75)	3 (19)	402	811	1213
Credit	25	18 (72)	21 (84)	16 (64)	1032	321	1353
Pg	10	6 (60)	9 (90)	6 (60)	372	177	549
Mean	–	71%	85%	61%	24 (/p)	31 (/p)	55 (/p)

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.

§6 Conclusion

We have proposed an analysis for detecting procedures and goals that are deterministic (i.e., that produce at most one solution at most once), or predicates whose clause tests are mutually exclusive, even if they are not deterministic (because they call other predicates which are nondeterministic). Our approach has advantages w.r.t. previous approaches in that it provides an algorithm for detecting mutual exclusion and it handles disunification tests 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.

Acknowledgment This work has been supported in part by the Information Society Technologies program of the European Commission, FP6 FET IST-15905 *MOBIUS*, IST-215483 *SCUBE*, and 06042-ESPASS, Ministry of Science projects TIN-2008-05624 *DOVES*, TIN2005-09207-C03 *MERIT-COMVERS*, Ministry of Industry project FIT-340005-2007-14, and CAM project S-0505/TIC/0407 *PROMESAS*.

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) F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.

4) P.W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In *Types in Logic Programming*, pages 157–187. MIT Press, 1992.

5) 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.

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. García de la Banda, W. Harvey, K. Marriott, and P. Stuckey. An Overview of HAL. In Joxan Jaffar, editor, *5th International Conference on Principles and Practice of Constraint Programming (CP'99)*, volume 1713 of *Lecture Notes in Computer Science*, pages 174–188. Springer-Verlag, October 1999.

8) M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, New York, 1979.

9) Roberto Giacobazzi and Laura Ricci. Detecting Determinate Computations by Bottom-up Abstract Interpretation. In Bernd Krieg-Brückner, editor, *4th European Symposium on Programming (ESOP'92)*, volume 582 of *Lecture Notes in Computer Science*, pages 167–181. Springer-Verlag, February 1992.

10) Fergus Henderson, Zoltan Somogyi, and Thomas Conway. Determinism Analysis in the Mercury Compiler. In Ramamohanarao Kotagiri, editor, *Proceedings of the 9th Australian Computer Science Conference*, volume 18 of *Australian Computer Science Communications*, pages 337–346. RMIT, The University of Melbourne, January 1996.

11) M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

12) P.M. Hill and A. King. Determinacy and determinacy analysis. *Journal of Programming Languages*, 5(1):135–171, December 1997.

13) 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.

14) Andy King, Lunjin Lu, and Samir Genaim. Detecting Determinacy in Prolog Programs. In Sandro Etalle and Miroslaw Truszczynski, editors, *Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4079 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2006.

15) 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.

16) 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.

17) P. López-García, F. Bueno, and M. Hermenegildo. Determinacy Analysis for Logic Programs Using Mode and Type Information. In *Proceedings of the 14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, number 3573 in LNCS, pages 19–35. Springer-Verlag, August 2005.

18) P. López-García, F. Bueno, and M. Hermenegildo. Inferring Determinacy and Mutual Exclusion in Logic Programs Using Mode and Type Analysis. Technical Report CLIP2/2009.0, Technical University of Madrid (UPM), School of Computer Science, UPM, February 2009.

19) Lunjin Lu and Andy King. Determinacy Inference for Logic Programs. In Mooly Sagiv, editor, *Programming Languages and Systems, 14th European Symposium on Programming (ESOP 2005)*, volume 3444 of *Lecture Notes in Computer Science*, pages 108–123. Springer, 2005.

20) Torben Æ. Mogensen. A Semantics-Based Determinacy Analysis for Prolog with Cut. In *Perspectives of System Informatics, Second International Andrei Ershov Memorial Conference*, volume 1181 of *Lecture Notes in Computer Science*, pages 374–385. Springer, 1996.

21) 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.

22) W. Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

23) Dan Sahlin. Determinacy Analysis for Full Prolog. In *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM'91)*, pages 23–30. ACM Press, 1991.

24) Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. *Journal of Logic Programming*, 29(1–3):17–64, October 1996.

25) C. Vaucheret and F. Bueno. More Precise yet Efficient Type Inference for Logic Programs. In *International Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, September 2002.