Non-Failure Analysis for Logic Programs

Saumya Debray
Department of Computer Science
University of Arizona
Tucson, AZ 85721, U.S.A.
debray@cs.arizona.edu

Pedro López-García, Manuel Hermenegildo
Facultad de Informática
Universidad Politécnica de Madrid
E-28660 Madrid, Spain
pedro@dia.fi.upm.es, herme@.fi.upm.es

Abstract

We provide a method whereby, given mode and (upper approximation) type information, we can detect procedures and goals that can be guaranteed to not fail (i.e., to produce at least one solution or not terminate). The technique is based on an intuitively very simple notion, that of a (set of) tests "covering" the type of a set of variables. We show that the problem of determining a covering is undecidable in general, and give decidability and complexity results for the Herbrand and linear arithmetic constraint systems. We give sound algorithms for determining covering that are precise and efficient in practice. Based on this information, we show how to identify goals and procedures that can be guaranteed to not fail at runtime. Applications of such non-failure information include programming error detection, program transformations and parallel execution optimization, avoiding speculative parallelism and estimating lower bounds on the computational costs of goals, which can be used for granularity control. Finally, we report on an implementation of our method and show that better results are obtained than with previously proposed approaches.

1 Introduction

There are two important motivations for considering compile-time analyses to identify non-failure in logic programs. The first is that it is usually very useful to be able to identify badly-behaved programs where possible. For example, in statically typed languages, the behavior one expects is that program components will be used in a way consistent with their types, and compile-time type checking is used to detect departures from this expected behavior. While this does not rule out programming errors, it makes it a lot simpler to identify and localize certain kinds of common programming errors. Similarly, in logic programs, the usual expectation is that a predicate will succeed and produce one or more solutions. In most logic programming systems, however, the only checking that is done is a rather simplistic—though useful—check about the naming of singleton variables. The second reason is that knowledge of non-failure can be used to aid a number of program transformations and optimizations. For example, we may want to execute possibly-failing goals ahead of non-failing goals where possible; and in parallel systems, knowledge of non-failure can be used to avoid speculative parallelism and to estimate lower bounds on the computational costs of goals [7, 5], which can be used for granularity control of parallel tasks [9].

The problem with naive attempts to infer non-failure is that, in general, it is always possible for a goal to fail because "bad" argument values cause a failure during head unification. An obvious solution would be to try and rule out such argument values by considering the types of predicates. However, most existing type analyses provide upper approximations, in the sense that the type of a predicate is a superset of the set of argument values that are actually encountered at runtime. Unfortunately, straightforward attempts to address this issue, for example by trying to infer lower approximations to the calling types of predicates, fail to yield nontrivial lower bounds for most cases.

In this paper, we show how, given mode and (upper approximation) type information, we can detect procedures and goals that can be guaranteed to not fail. Our technique is based on an intuitively very simple notion, that of a (set of) tests "covering" the type of a variable. We show that the problem of determining a covering is undecidable in general, and give decidability and complexity results for the common cases involving the Herbrand and linear arithmetic constraint systems. We then give an algorithm for checking whether a set of tests covers a type, that is efficient in practice. Based on this information, we show how to identify goals and procedures that can be guaranteed to not fail at runtime.

Space limitations prevent us from discussing several issues completely or including some of the longer proofs. We refer the reader to [4] for details.

2 Preliminaries

We assume an acquaintance with the basic notions of logic programming. In order to reason about non-failure, 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 programs are *moded*, 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 ground, and allows an output argument to occur as a subterm of an input argument). Where it is necessary to emphasize the input tests in a clause, we write the clause in "guarded" form, as

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

Consider a predicate defined by the clauses

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

Suppose we know that this predicate will always be called with its first argument bound to an integer. Obviously, for any particular call, one or the other of the tests '$X \geq 0$' and '$X < 0$' may fail; however, taken together, one of them will succeed. This shows that we cannot rely on examining the tests of each clause separately: it is necessary to collect them together and examine the behavior of the collection as a whole. When collecting tests together, however, we must be careful to make sure that we do not get confused by different variable names in different clauses. For example, in the *abs* predicate defined above, we need to make sure that (i) we notice that the variable X in the first clause and the variable Y in the second clause actually refer to the same component of the arguments to the predicate; and (*ii*) we do not confuse the variable Y in the first clause with the variable Y in the second clause.

These pitfalls can be avoided by normalizing clauses so that they use variable names consistently and according to a predefined convention. We rely on the usual approach of using sequences of integers to encode paths in ordered trees: the empty sequence ε corresponds to the root node of the tree, and if a sequence π corresponds to a node N, then the sequences $\pi 1, \dots, \pi k$ correspond to its children $N_1, \dots, N_k$ taken in order. We adopt the convention that a variable in a clause is designated as X_π, where π is (the sequence encoding) the path from the root of the clause, labeled `:-`/2, to the leftmost occurrence of that variable. To enhance readability, the examples used in this paper will not resort to explicitly naming variables in this way unless necessary, with the understanding that the algorithms are defined with respect to normalized clauses.

3 Types, Tests, and Coverings

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 [13]). 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 $\mathsf{type}[p]$. The non-failure 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*.

A more detailed treatment of these issues may be found in papers on type analysis, e.g., [3, 13]. Due to space limitations we do not pursue them further here. 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, a *type assignment over* V is a mapping from V to a set of types. A type assignment ρ over a set of variables $\{x_1, \dots, x_n\}$ is written as $(x_1 : a_1, \dots, x_n : a_n)$, where $\rho(x_i) = a_i, 1 \leq i \leq n$, and a_i is a type representation. Given a term t and a type representation T, we abuse of terminology and say $t \in T$, meaning that t belongs to the set of terms denoted by T.

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". 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.

Fundamental to our approach to detecting non-failure is the notion of a test "covering" a type assignment:

Definition 3.1 A test $S(\bar{x})$ *covers* a type assignment $\bar{x} : \bar{T}$, where $\bar{T}$ is a tuple of nonempty types, if for every $\bar{t} \in \bar{T}$ it is the case that $\bar{x} = \bar{t} \models S(\bar{x})$. ∎

Consider a predicate p defined by n clauses, with input tests $\tau_1, \dots, \tau_n$:

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

$$\dots$$

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

We refer to the test $\tau(\bar{x}) \equiv \tau_1(\bar{x}) \vee \cdots \vee \tau_n(\bar{x})$ as the *input test of* p. Suppose that the predicate p has type $\mathsf{type}[p]$: in the interest of simplicity, we sometimes abuse terminology and say that the predicate p covers the type $\mathsf{type}[p]$ if the input test $\tau(\bar{x})$ of p covers the type assignment $\bar{x} : \mathsf{type}[p]$.

Define the "calls" relation between predicates in a program as follows: p calls q, written $p \leadsto q$, if and only if a literal with predicate symbol q appears in the body of a clause defining p, and let $\leadsto^\star$ denote the reflexive transitive closure of $\leadsto$. The importance of the notion of covering is expressed by the following result:

Theorem 3.1 *A predicate p in the program is non-failing if, for each predicate q such that $p \leadsto^\star q$, q covers* $\mathsf{type}[q]$.

Proof Assume that p can fail, i.e., there is a goal $p(\bar{t})$, with $\bar{t} \in \mathsf{type}[p]$, that fails. It is a straightforward induction on the number of resolution steps to show that there is a q such that $p \leadsto^\star q$ and q does not cover its type. ■

Note that non-failure does not imply success: a predicate that is non-failing may nevertheless not produce an answer because it does not terminate. This is illustrated by the predicate, defined by the single clause given below, which is non-failing and—on most existing Prolog systems—non-terminating:

$$p(X) \ :- \ p(X).$$

Ideally, we would like to have a decision procedure to determine whether a test covers a given type assignment. Unfortunately, this is impossible in general, as the following result shows:

Theorem 3.2 *Given an arbitrary test and type assignment, it is in general undecidable whether the test covers the type assignment.*

Proof The proof is straightforward from a result, due to Matijasevič, that shows that determining the existence of (integer) solutions for arbitrary Diophantine equations is undecidable [16]. Given an arbitrary polynomial $\phi(x_1, \ldots, x_n)$, consider the test $\phi(x_1, \ldots, x_n) \neq 0$. This test covers the type assignment $(x_1 : \mathtt{integer}, \ldots, x_n : \mathtt{integer})$ if and only if every possible assignment of integers to the variables $x_1, \ldots, x_n$ causes the polynomial ϕ to take on a non-zero value, i.e., if and only if the Diophantine equation $\phi(x_1, \ldots, x_n) = 0$ has no integer solutions. But since the problem of determining the existence of integer roots for an arbitrary Diophantine equation is undecidable, it follows that the problem of determining whether an arbitrary test covers an arbitrary type assignment is also undecidable. ■

We are therefore forced to resort to sound (but, necessarily, incomplete) algorithms to determine coverings. In the remainder of this section we show that covering problems are decidable for most cases arising in practice—in particular, for equality and disequality tests over the Herbrand domain and for linear arithmetic tests—and give algorithms for deciding covering for these cases. Given a test and a type assignment that we want to check for covering, our approach is to first partition the test such that tests in different resulting partitions involve different constraint systems, and then apply to each partition a covering algorithm particular to the corresponding constraint system. In this paper we consider two commonly encountered constraint systems: first order terms with equality and disequality tests, on variables with *tuple-distributive regular types* [3] (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.

3.1 Covering in the Herbrand Domain

3.1.1 Decidability and Complexity

While covering is undecidable in the presence of arbitrary arithmetic operations, it turns out to be decidable if we restrict ourselves to equations and disequations over Herbrand terms. Before discussing the algorithm for this, we give a result on the complexity of the covering problem for Herbrand:

Theorem 3.3 *The covering problem for the Herbrand domain is co-NP-hard. It remains co-NP-hard even if we restrict ourselves to equality tests.*

Proof By reduction from the problem of determining whether a propositional formula in disjunctive normal form, containing at most 3 literals per disjunct, is a tautology ([10], problem LO8). ∎

3.1.2 A Decision Procedure

The decision procedure presented here is inspired by a result, due to Kunen [14], 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 covering 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 in the following) of *type-annotated term*, and in general *elementary set*, as representations which denote some subsets of $\mathcal{H}^n$ (for some $n \geq 1$). Given a representation S (elementary set or type-annotated term), $Den(S)$ refers to the subset of $\mathcal{H}^n$ denoted by S.

Definition 3.2 [type-annotated term] A *type-annotated term* is a pair $M = (\bar{t}, \rho)$, where $\bar{t}$ is a tuple of terms, and ρ is a type assignment $(x_1 : T_1, \ldots, x_k : T_k)$. To enhance readability, the type of x_i in M, i.e., T_i, will sometimes be written as $type(x_i, M)$ or as $type(x_i, \rho)$. Also, given a type-annotated term M, we denote its tuple of terms and its type assignment as $\bar{t}_M$ and ρ_M respectively. A type-annotated term denotes the set of all the ground terms $\theta(\bar{t})$ such that $\theta(x) \in type(\bar{t}, \rho)$ for each variable in $\bar{t}$. ∎

Given a type-annotated term $(\bar{t}, \rho)$, the tuple of terms $\bar{t}$ can be regarded as a *type term* and ρ can be considered to be a type substitution. This is useful for using an algorithm described by Dart and Zobel [3] to compute the "intersection" and "inclusion" of type-annotated terms, to be defined later. Let $\top$ denote the type of the entire Herbrand universe. When we have a type-annotated term $(\bar{t}, \rho)$ such that $\rho(x) = \top$ 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 3.3 [elementary set] An elementary set is defined as follows:

- Λ is an elementary set, and denotes the set $\emptyset$ (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 $Den(A) \subseteq Den(B)$. $A \simeq B$ iff $Den(A) = Den(B)$.

Definition 3.4 [cobasic set] A cobasic set is an elementary set of the form $comp(B)$, where B is a tuple of terms. ∎

Definition 3.5 [minset] A *minset* is either Λ or an elementary set of the form $X \otimes comp(Y_1) \otimes \cdots \otimes comp(Y_p)$, for some $p \geq 0$, where X is a tuple of terms, $comp(Y_1), \ldots, comp(Y_p)$ are cobasic sets, and for all $1 \leq i \leq p$, $Y_i = X\theta_i$ and $X \not\sqsubseteq Y_i$ for some substitution θ_i. ∎

For brevity, we write a minset of the form $X \otimes comp(Y_1) \otimes \cdots \otimes comp(Y_p)$ as X/C, where $C = \{Y_1, \ldots, Y_p\}$ (we say that C is the set of cobasic sets of the minset, although syntactically $Y_1, \ldots, Y_p$ are tuples of terms).

Definition 3.6 [type-annotated term instance] We say that the type-annotated term I is an instance of the type-annotated term R if $Den(I) \subseteq Den(R)$ and there is a substitution θ such that $\bar{t}_I = \bar{t}_R\theta$. ∎

Consider a predicate p defined by n clauses, with input tests $\tau_1(\bar{x}), \ldots, \tau_n(\bar{x})$: Suppose that the predicate p has type $\mathsf{type}[p]$. Testing whether the input test of p, $\tau(\bar{x})$, covers the type assignment $\bar{x} : \mathsf{type}[p]$ can be reduced to test whether $M \sqsubseteq S_1 \oplus \cdots \oplus S_n$, where M is a type-annotated term which is a representation of $\bar{x} : \mathsf{type}[p]$, and each S_i is a minset, which is the representation of $\tau_i(\bar{x})$. $\tau_i(\bar{x})$ can be transformed into the minset S_i as follows:

1. Assume that the test $\tau_i(\bar{x})$ is of the form $E_i \wedge D_i^1 \wedge \cdots \wedge D_i^m$, where E_i is the conjunction of all unification tests of $\tau_i(\bar{x})$ (i.e., a system of equations) and each D_i^j a disunification test (i.e., an disequation).
2. Let θ_i be the substitution associated with the solved form of E_i (this can be computed by using the techniques of Lassez et al. [15]).
3. Let θ_i^j, for $1 \leq j \leq m$, be the substitution associated with the solved form of $E_i \wedge N_i^j$, where N_i^j is the negation of D_i^j.
4. $S_i = B_i \otimes comp(B_i^1) \otimes \cdots \otimes comp(B_i^m)$, where $B_i = (\bar{x})\theta_i$ and $B_i^j = (\bar{x})\theta_i^j$, for $1 \leq j \leq m$.

Then, we have that $M \sqsubseteq S_1 \oplus \cdots \oplus S_m$ iff $M \otimes comp(S_1) \otimes \cdots \otimes comp(S_m) \simeq \Lambda$. We then can write $comp(S_1) \otimes \cdots \otimes comp(S_m)$ into disjunctive normal form as $M_1 \oplus \cdots \oplus M_u$, where each M_i is a minset. [1] Since $M \otimes (M_1 \oplus \cdots \oplus M_u) \simeq M \otimes M_1 \oplus \cdots \oplus M \otimes M_u$, we have that $M \sqsubseteq S_1 \oplus \cdots \oplus S_m$ iff $M \otimes M_i \simeq \Lambda$ for all $1 \leq i \leq u$. Thus, 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 $empty(M, S)$, defined in Figure 1.[2] First, it performs the "intersection" of M and the tuple of terms of the minset S. This intersection is implemented by the function $intersection(R, Cob)$, which returns $R \otimes Cob$ (recall that a tuple of terms is a type-annotated term), and is a straightforward adaptation of the function $unify(\tau_1, \tau_2, T, \Theta)$ described in [3], that performs a *type unification* where τ_1 and τ_2 are type terms, Θ a type substitution for the variables in τ_1 and τ_2, and T a set of type rules defining τ_1, τ_2, and Θ. Then, if the mentioned intersection is not empty, nor A ($S = A/C$) is "included" in R, it calls $empty1(C, R, \emptyset)$, which checks whether $R/C \simeq \Lambda$. This is done by checking if R is "included" in some cobasic set in C (in which case $R/C \simeq \Lambda$). For this purpose, it uses the function $included(R, Co)$, which is a straightforward adaptation of the function $subset_T(\tau_1, \tau_2)$ described in [3], that determines whether the type denoted by one pure type term is a subset of the type denoted by another (i.e., $included(R, Co)$ returns **true** if and only if $R \sqsubseteq Co$).

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.

---

[1] Note that $\oplus$, $\otimes$, and *comp* constitute a Boolean algebra, and the operation $\otimes$ is computable for type-annotated terms.

[2] 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.

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 $R/C \simeq \Lambda$). If a cobasic set Cob in C is "disjoint" with R, then it is useless (however, there can be useless cobasic sets in C which are not disjoint with R). If R is not "included" in none 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 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 $expansion(R, Cob)$, which takes a type-annotated term R and a cobasic set Cob and returns a pair $(R', Rest)$ (which is a "partition" of R) such that: R' is a type-annotated term; $Rest$ is a set of type-annotated terms; for all $x \in vars(R')$, $x\theta$ is a variable, where $\theta = mgu(\bar{t}_{R'}, Cob)$, or $type(x, R') = \top$; $(\cup_{X \in Rest} Den(X)) \cup Den(R') = Den(R)$; and for all $X \in Rest$, $X \otimes Cob \simeq \Lambda$. 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. The function $mgu(A, B)$ returns an (idempotent) most general unifier θ of the tuples of terms A and B. For example, assume that $R = ((X, Y), (X : list, Y : list))$ and $C = \{C_1, C_2\}$, where $C_1 = ([H|L], Z)$ and $C_2 = ([\,], Z)$. R is not included in none 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 $R_1 = (([H1|L1], Y1), (H1 : \top, L1 : list, Y1 : list))$ and $R_2 = (([\,], Y2), (Y2 : list))$, we have that R_1 and R_2 are included in C_1 and C_2 respectively. However, in other situations, the problem cannot be solved by expanding R: assume, for example, that now $C = \{(Z, Z)\}$, in this case, R is not included in (Z, Z) because this cobasic set 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"). In step 6, these aliased variables are computed by using the function $aliased(R, Cob)$, which takes a cobasic set Cob and a type-annotated term R such that for all $x \in vars(R)$, $x\theta$ is a variable, where $\theta = mgu(\bar{t}_R, Cob)$, and returns a set of variables $AlVars$ such that $v \in AlVars$ iff $v \in vars(R)$ and exists $v' \in vars(R)$ such that $v\theta \equiv v'\theta$. If for some $x \in vars(R')$, it holds that $type(x, R') = \top$ and either $x \in AVars$, or $x\theta'$ is not a variable, then we can say that Cob useless. This can be proved by using the variable x to construct an instance S such that: assuming that there exists an instance I of R, such that $I \otimes Cs \simeq \Lambda$ for all $Cs \in Cset$, where $Cset = C' \cup \{CS \mid (B, A, CS) \in AL\}$, then, S can be constructed from I so that $S \otimes C_2 \simeq \Lambda$ for all $C_2 \in \{Cob\} \cup Cset$.

The function $empty1(C, R, AL)$ performs a "first round" over the cobasic sets in C. After this round (whose end is detected in step 2 by the condition $C'' \equiv \emptyset$), 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 bounded 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 $empty2(AL, R)$ selects a cobasic set Cob in AL, and, if R is not included in it, 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 Cob or "disjoint" with it. It also ensures that R is not infinitely expanded (note that the type of such variables is finite).

Example 3.1 Consider the predicate `reverse/2`:

```
reverse(X,Y)  :- X = []      ‖ Y = [].
reverse(X,Y)  :- X = [X1|X2] ‖ reverse(X2,Y2), append(Y2,[X1],Y).
```

and the type assignment $\rho \equiv (X : \mathit{list})$, where $\mathit{list} ::= [\,] \mid [\top|\mathit{list}]$. Let τ be the input test of the predicate `reverse/2`, i.e., $\tau \equiv X = [\,] \vee X = [X1|X2]$. Let M be the type-annotated term which is a representation of ρ, i.e., $M = ((X), (X : \mathit{list}))$. The minset representations of $X = [\,]$ and $X = [X1|X2]$ are $([\,])$ and $([X1|X2])$ respectively (in this example we deal with unary tuples).

We have that τ covers ρ iff $((X), (X : \mathit{list})) \sqsubseteq ([\,]) \oplus ([X1|X2])$ iff $((X), (X : \mathit{list})) \otimes \mathit{comp}(([\,]) \oplus ([X1|X2])) \simeq \Lambda$ iff $((X), (X : \mathit{list})) \otimes \mathit{comp}(([\,])) \otimes \mathit{comp}(([X1|X2])) \simeq \Lambda$. The disjunctive normal form of $\mathit{comp}(([]))\otimes \mathit{comp}(([X1|X2]))$ is $(X3) \otimes \mathit{comp}(([\,])) \otimes \mathit{comp}(([X1|X2]))$, which has only one minset. Now, we have to prove that $((X), (X : \mathit{list})) \otimes (X3) \otimes \mathit{comp}(([\,])) \otimes \mathit{comp}(([X1|X2])) \simeq \Lambda$, i.e., whether the call $\mathit{empty}(M, S)$, where $M = (\bar{t}_M, \rho_M)$, $\bar{t}_M \equiv (X)$, $\rho_M \equiv (X : \mathit{list})$, and $S \equiv (X3)/\{([\,]), ([X1|X2])\}$ returns **true**. This call proceeds as follows (and in fact returns **true**):

1. $\mathit{intersection}(M, (X3))$ returns the type-annotated term $((X4), (X4 : \mathit{list}))$.
2. Since this intersection is not "empty" and $(X3)$—which represents the type-annotated term $((X3), (X3 : \top))$ —is not "included" in $((X4), (X4 : \mathit{list}))$, the call $\mathit{empty1}(\{([\,]), ([X1|X2])\}, ((X4), (X4 : \mathit{list})), \emptyset)$ is performed. This calls returns **true** and the computation is as follows:
   (a) We have that $((X4), (X4 : \mathit{list}))$ is not "included" in none of the cobasic sets in $\{([\,]), ([X1|X2])\}$. Thus, a cobasic set is selected from this set. Assume that $([X1|X2])$ is the selected cobasic set;
   (b) $(R', \mathit{Rest}) = \mathit{expansion}(((X4), (X4 : \mathit{list})), ([X1|X2]))$, where $R' = (([X5|X6]), (X5 : \top, X6 : \mathit{list}))$, and $\mathit{Rest} = \{(([\,]), \emptyset)\}$ ($\emptyset$ denotes an empty type assignment, since $([\,])$ has no variables).
   (c) The call $\mathit{included}(R', ([X1|X2]))$ returns **true**, and thus the call $\mathit{empty1}(\{([\,])\}, (([\,]), \emptyset), \emptyset)$ is performed. This call also returns **true**, because $(([\,]), \emptyset) \sqsubseteq ([\,])$. Thus, the initial call returns **true**. □

The covering algorithm we present is complete for *tuple-distributive regular types*:

Theorem 3.4 *Let M be a type-annotated term in which all types are tuple-distributive regular types, and S a minset. Then $\mathit{empty}(M, S)$ terminates, and returns* **true** *iff $M \otimes S \simeq \Lambda$.*

While sound, the algorithm is not complete for regular types in general (though we believe it is fairly accurate in practice):

Theorem 3.5 *Let M be a type-annotated term where all types are regular types, and S a minset. Then $\mathit{empty}(M, S)$ terminates, and if it returns* **true** *then $M \otimes S \simeq \Lambda$.*

For each of these theorems, correctness can be argued by induction on the depth of recursion of functions *empty1* and *empty2* upon termination; the termination arguments follow standard lines. Complete proofs can be found in [4].

One reason for imprecision in the case of non tuple-distributive regular types is that the function $\mathit{intersection}(M, A)$ described above computes a superset of the exact intersection when we deal with general regular types (this result can be derived from the work of Dart and Zobel [3]). Another reason comes from the use of the function $\mathit{expansion}(R, \mathit{Cob})$ to partition the type-annotated term R in the boolean

$\boxed{empty(M,S):}$

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 tuples of terms).

Output: a boolean value denoting whether $M \otimes S \simeq \Lambda$.

Process:

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

$\boxed{empty1(C, R, AL):}$

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,
- For all $x \in vars(B)$, $x\theta$ (where $\theta = mgu(\bar{t}_B, CS)$) is a variable, and,
- $v \in AV$ iff $v \in vars(B)$ and exists $v' \in vars(B)$ such that $v\theta \equiv v'\theta$ (i.e., AV is the set of variables in $vars(B)$ which are aliased with some other variable in $vars(B)$ by θ).

Output: a boolean value denoting whether $R/C_1 \simeq \Lambda$, where $C_1 = C \cup \{Cob \mid (B, A, Cob) \in AL,\ for\ some\ B\ and\ A\}$.

Process:

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

$\boxed{empty2(AL, R):}$

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

Figure 1: $\boxed{empty(M,S)}$

function *empty1*$(C, R, \emptyset)$. Given a pair $(R', Rest)$ where R' is a type-annotated term, and $Rest$ is a set of type-annotated terms, we assume that all type-annotated terms in $Rest$ are disjoint with the cobasic set Cob, but this is not true for general regular types, and, consequently, precision may be lost. A possible solution in order to obtain a complete algorithm for general regular types would be to rewrite the type annotated term which represents the input type of a predicate as a union of type annotated terms containing only tuple-distributive types, and then apply the above described covering algorithm for each of the elements of the union.

3.2 Covering in Linear Arithmetic over Integers

In this section, we consider linear arithmetic tests over integers (the ideas extend directly to linear tests over the reals, which turn out to be computationally somewhat simpler). Without loss of generality, assume that the tests are in disjunctive normal form, i.e., they are of the form $\Phi(\bar{x}) = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m} \phi_{ij}(\bar{x})$ where each of the tests $\phi_{ij}(\bar{x})$ is of the form $\phi_{ij}(\bar{x}) \equiv a_0 + a_1 x_1 + \cdots + a_k x_k \textcircled{?} 0$, with $\textcircled{?} \in \{=, \neq, <, \leq, >, \geq\}$. Determining whether $\Phi(\bar{x})$ covers the type assignment of `integer` to each variable in $\bar{x}$ amounts to determining whether $\models (\forall \bar{x})\Phi(\bar{x})$. This is true if and only if $(\exists \bar{x})\neg\Phi(\bar{x})$ is unsatisfiable. In other words, we need to determine the unsatisfiability of

$$\neg\Phi(\bar{x}) \;=\; \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m} \neg\phi_{ij}(\bar{x}) \;=\; \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m} \psi_{ij}(\bar{x}),$$

where $\psi_{ij}(\bar{x})$ is derived from $\neg\phi_{ij}(\bar{x})$ as follows: let $\phi_{ij}(\bar{x}) = \sum_{i=0}^{k} a_i x_i \textcircled{?} 0$. If $\textcircled{?}$ is a comparison operator other than '=', $\psi_{ij}(\bar{x})$ is simply $\sum_{i=0}^{k} a_i x_i \overline{\textcircled{?}} 0$, where $\overline{\textcircled{?}}$ is the complementary operator to $\textcircled{?}$, e.g., if $\textcircled{?} \equiv$ '$>$' then $\overline{\textcircled{?}} \equiv$ '$\leq$'. If $\textcircled{?} \equiv$ '$=$', the corresponding complementary operator is '$\neq$', but this can be written in terms of two tests involving the operators '$>$' and '$<$':

$$\psi_{ij}(\bar{x}) = (\textstyle\sum_{i=0}^{k} a_i x_i > 0) \;\vee\; (\sum_{i=0}^{k} a_i x_i < 0).$$

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

Since determining whether an integer programming problem is solvable is NP-complete [10], the following complexity result is immediate:

Theorem 3.6 *The covering 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 covering analysis, are therefore efficiently decidable.

3.3 Covering Analysis: Putting it Together

Let τ be the input test of predicate p and ρ a type assignment. Consider the type assignment ρ written as a type-annotated term M, and τ written in disjunctive normal form, i.e., $\tau = \tau_1 \vee \cdots \vee \tau_n$, where each τ_i is a conjunction of primitive tests (recall that primitive tests are unification, disunification, etc.). Consider the test τ_i 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 τ_i^H written as a minset D_i (recall that D_i is the intersection (conjunction) of a tuple of terms, and zero or more cobasic sets). Let D be the union (disjunction) of the these minsets.

Example 3.2 Let p be the predicate `partition`/4 from the familiar quicksort program. Let τ be $X = [\,] \vee (X = [H|L] \wedge H > Y) \vee (X = [H|L] \wedge H \leq Y)$ and let ρ be $(X : \mathit{intlist}, Y : \mathit{integer})$, where $\mathit{intlist} ::= [\,] \mid [\mathit{integer}|\mathit{intlist}]$. In this case, we have that M is $((X, Y), (X : \mathit{intlist}, Y : \mathit{integer}))$. $\tau_1 \equiv X = [\,]$, $\tau_2 \equiv X = [H|L], H > Y$, and $\tau_3 = X = [H|L], H \leq Y$. τ_1 can be written as $\tau_1^H \wedge \tau_1^A$, where $\tau_1^H \equiv X = [\,]$ and $\tau_1^A \equiv \mathit{true}$. Similarly, $\tau_2^H \equiv X = [H|L]$ and $\tau_2^A \equiv H > Y$, and $\tau_3^H \equiv X = [H|L]$ and $\tau_3^A \equiv H \leq Y$. $D = D_1 \oplus D_2 \oplus D_3$, where $D_1 \equiv ([\,], Y)$, $D_2 \equiv ([H|L], Y)$ and $D_3 \equiv ([H|L], Y)$. $\square$

To test whether τ covers ρ, we first test that D covers M, ignoring the arithmetic tests. If D does not cover M, then obviously, the (whole) input test of p, τ, does not cover M, and we report failure. Otherwise, we create (zero or more) covering subproblems, each of them containing only arithmetic tests, as follows:

1. Let A be the set of all the tuples of terms and negations of cobasic sets appearing in D (note that the negation of a cobasic set is a tuple of terms, thus A is a set of tuples of terms), and let $A' = \{b \in A \mid M \otimes b \not\approx \Lambda\}$.
2. For each tuple of terms b in A':
   (a) Let $I = M \otimes b$ and $\theta = mgu(\bar{t}_M, \bar{t}_I)$;
   (b) Let $\tau_b = \bigvee_{j=1}^{m} r_j$, where $\{r_1, \ldots, r_m\} = \{t_i \mid b \sqsubseteq D_i$ for some $1 \leq i \leq n$ and t_i is the result of applying θ to τ_i^A (this is done to take into account possible variable aliasing)$\}$. Note that there is an algorithm to test whether $b \sqsubseteq D_i$ in [14].
   (c) Test whether τ_b covers ρ_I (recall that ρ_I refers to the type assignment of I):
      i. Assume that $\tau_b = s_1 \vee \cdots \vee s_n$ and each s_i is a conjunction of primitive arithmetic tests. If $\tau_b \equiv \mathbf{true}$ then report success;
      ii. otherwise, if for some variable x appearing in all s_i, $1 \leq i \leq n$, it holds that $\mathit{type}(x, \rho_I)$ is not a numeric type, then report failure;
      iii. otherwise, use the algorithm described in section 3.2 to test whether τ_b covers ρ_I.

Theorem 3.7 *If D covers M and for each $b \in A'$, τ_b covers I, then the input test of p, τ, covers M.*

Proof It is clear that if D covers M, then the disjunction of all the basic sets in A' also covers M. Thus, for any tuple of terms $\bar{x}$ which is an instance of M, there is at least a $b \in A'$, such that $\bar{x}$ is an instance of b, and all the tests τ_i^H such that $b \sqsubseteq D_i$, will succeed for $\bar{x}$. If τ_b covers M, then at least one of the tests t_i in τ_b will succeed for $\bar{x}$. Thus, by the construction of τ_b, at least one τ_i will succeed for $\bar{x}$, and we conclude that τ covers M. ■

Example 3.3 Consider Example 3.2. It is clear that D covers M, thus we proceed as follows:

1. $A = \{([\,], Y), ([H|L], Y)\}$, and $A' = A$.
2. Let $b1 = ([], Y)$ and $b2 = ([H|L])$. Then $\tau_{b1} \equiv \mathbf{true}$ and $\tau_{b2} \equiv H > Y \vee H \leq Y$.
3. We have that **true** covers $(([], Y), (Y : \mathit{integer}))$, and also that $H > Y \vee H \leq Y$ covers $(L : \mathit{intlist}, H : \mathit{integer}, Y : \mathit{integer})$, thus τ covers M. $\square$

4 Non-Failure Analysis

4.1 The Analysis Algorithm

Once we have determined which predicates cover their types, determining non-failure is straightforward: from Theorem 3.1, analysis of non-failure reduces to the determination of reachability in the call graph of the program. In other words, a predicate p is non-failing if there is no path in the call graph of the program from p to any predicate q that does not cover its type. It is straightforward to propagate this reachability information in a single traversal of the call graph in reverse topological order. The idea can be illustrated by the following example.

Example 4.1 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).
```

Suppose that `qs`/2 has mode (`in`, `out`) and type (`intlist`, `-`), and suppose we have already shown that `part`/4 and `app`/3 cover the types `(int, intlist, -, -)` and `(intlist, intlist, -)` induced for their body literals in the recursive clause above. The input tests for `qs`/2 are `X1 = []` $\vee$ `X1 = [H|L]`, and this covers the type `intlist`, which means that head unification will not fail for `qs`/2. It follows that a call to `qs`/2 with the first argument bound to a list of integers will not fail. $\square$

4.2 A Prototype Implementation

In order to evaluate the effectiveness and efficiency of our approach to non-failure analysis we have constructed a relatively complete prototype which performs such analysis in an automatic way. The system takes Prolog programs as input, which include a module definition in the standard way. In addition, the types and modes of the arguments of exported predicates are given, as well as the required type definitions. The system uses the PLAI analyzer to derive mode information, using the Sharing+Freeness domain [17], and an adaptation of Gallagher's analysis to derive the types of predicates [8]. The resulting type- and mode-annotated programs are analyzed using the algorithms presented for Herbrand and linear arithmetic tests.

Herbrand covering is checked by a naive direct implementation of the analyses presented. Testing of covering for linear arithmetic tests is implemented directly using the Omega test [18]. 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. Some relevant results of these tests are presented in Table 1. **Program** lists the program names, **N** the number of predicates in the program, **F** the number of predicates detected by the analysis as non-failing, **Cov** the number of predicates detected to cover their type, **C** the number of non-failing predicates detected in [2], $\mathbf{T}_F$ the time required by the covering analysis (SPARCstation 10, 55MHz, 64Mbytes of memory), $\mathbf{T}_M$ 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 significantly more powerful than those previously reported in non-failure detection (the experimental results presented in [2] suggest that it is more appropriate for detecting determinacy than for non-failure). It is pointed out in [2] that the sure success information can be improved by using a more sophisticated type domain. However, this is also applicable to our analysis, and the types inferred by our system are similar to those used in [2]. Much of the power of our algorithm comes from the use of the notion of covering, which allows detecting when at least one of the clauses (not necessarily the same) defining a predicate will not fail for all possible calls. The cardinality analysis detects non-failure only when at least one of the clauses (always the same) defining a predicate will not fail for all the possible calls. The non-failure analysis times are quite good, despite the currently 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 quite acceptable, even when including the type and mode analysis times, which are in any case very useful in other parts of the compilation process.

The Mercury system [11] allows the programmer to declare that a predicate will produce at least one solution, and attempts to verify this with respect to the Herbrand terms with equality tests. As far as we know, the Mercury compiler does not handle disequality 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.

Program	**N**	**F (%)**	**Cov (%)**	**C**	$\mathbf{T}_F$	$\mathbf{T}_M$	$\mathbf{T}_T$
Hanoi	2	2 (100)	2 (100)	N/A	60	860	920
Deriv	1	1 (100)	1 (100)	N/A	80	940	1,020
Fib	1	1 (100)	1 (100)	N/A	20	90	110
Mmatrix	3	3 (100)	3 (100)	N/A	90	350	440
Tak	1	1 (100)	1 (100)	N/A	10	110	120
Subs	1	1 (100)	1 (100)	N/A	50	90	140
Reverse	2	2 (100)	2 (100)	N/A	10	100	110
Qsort	3	3 (100)	3 (100)	0 (0)	80	440	520
Qsort2	5	3 (60)	3 (60)	0 (0)	100	390	490
Queens	5	2 (40)	2 (40)	0 (0)	120	360	480
Gabriel	20	3 (15)	10 (50)	0 (0)	420	1,860	2,280
Read	38	8 (21)	19 (50)	8 (21)	540	12,240	12,780
Kalah	44	18 (40)	29 (65)	6 (13)	1,500	14,570	16,070
Plan	16	4 (25)	11 (68)	0 (0)	810	7,000	7,810
Credit	25	10 (40)	18 (72)	0 (0)	4,720	1,470	6,190
Pg	10	2 (20)	6 (60)	0 (0)	540	1,600	2,140
Mean	–	36%	63%	3%	51 (/p)	239 (/p)	291 (/p)

Table 1: Accuracy and efficiency of the non-failure analysis (times in mS).

5 Applications

There are several applications of this analysis. The first application is implementing granularity control in parallelizing compilers. All of the work that we know of in this context involves estimating upper bounds to the cost of goals (see, for example, [6]). The use of upper bounds allows us to guarantee that, given a program that is already parallelized, we can make it run more efficiently by running some of the parallel goals sequentially. However, the problem faced by parallelizing compilers is in fact exactly the converse of the one tackled above: what needs to be guaranteed is that the parallel execution will be more efficient than the sequential one, rather than the other way around. This type of granularity control can be solved using essentially the same general approach, but we need a *lower bound* on the cost of each goal. The detection of such lower bounds is not too different from that of upper bounds, except that it requires knowledge of non-failure, since otherwise only a trivial lower bound of zero can be derived [7]. The techniques presented in the paper directly address this problem. In fact, the usefulness of lower bounds was already clear when the work presented in [6] was developed, but the determination of useful lower bounds was deemed too difficult at the time. This approach allows us to guarantee that, given a sequential program, it will run more efficiently by running some of the goals in parallel. This in effect allows obtaining *guaranteed speedups* (or, at least, ensuring that no *slow-downs* will occur) from automatic parallelization, even in architectures for which parallel execution involves a significant overhead. We know of no other approach which can achieve this.

The second application has to do again with (and-)parallelism, in particular with the avoidance of speculative computation. Consider a number of goals in a resolvent which are determined to be independent. As shown in [12], and ignoring parallelization overheads (which can be dealt with as illustrated above), the time involved in their parallel execution can be guaranteed to be smaller or equal to that of the corresponding sequential execution. However, it is impossible to guarantee that no more work will be performed. This is due to the possibility of failure of one of the goals. Consider two goals p and q so that q is executed after p in the sequential execution. Assume also that p fails (both in the sequential and, correspondingly, in the parallel execution). If p and q are scheduled for execution in parallel, a part of q may be executed until the point in which p fails (the execution of q will normally be killed at this point). Although not producing a slow-down, this constitutes unnecessary computation which steals computing resources from any useful work that may exist in the system (and therefore does reduce speedup). Determining that goals in a conjunction will not fail (at least all but the rightmost one – note that failure of q in the example above does not have these ill-effects) thus allows guaranteeing avoidance of speculative computation.

A third application is in the general area of program transformation, where information about non-failure can be used in determining the order of execution of literals in a clause. Consider a clause

$$H \ :- \ B_1, p(X), B_2, q(X), B_3$$

where B_1, B_2, B_3 are sequences of literals, $p(X)$ produces bindings for X, and $q(X)$ is the left-most body goal that has X as an input argument. If p is known to be non-failing, it may be possible to transform this clause to

$$H \ :- \ B_1, B_2, p(X), q(X), B_3.$$

The resulting code may be more efficient than the original if a goal in B_2 can fail.

Finally, among the most important applications of non-failure we envision is in speeding up program development by assisting programmers by reporting predicates that are not guaranteed to not fail. This can help in detecting programming errors at compile time, in much the same way as type checking does in statically typed languages, since in logic programs the usual expectation is that a predicate will succeed and produce one or more solutions. In most logic programming systems, however, little compile-time checking is performed. The system is currently integrated in the CIAO system and used for these purposes (as well as for optimization).

Acknowledgements

The work of S. Debray was supported in part by the National Science Foundation under grant CCR-9123520. The work of M. Hermenegildo and P. López-García was supported in part by ESPRIT project LTR 22532 "DiSCiPl" and CICYT proyect number TIC96-1012-C02-01.

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 F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. MIT Press, 1987.

[4] S. Debray, P. López García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. Technical Report TR Number CLIP20/96.0, T.U. of Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, November 1996.

[5] S. Debray, P. López García, M. Hermenegildo, and N. Lin. Lower Bound Cost Estimation for Logic Programs. Technical Report CLIP20/95.0, T.U. of Madrid (UPM), Facultad Informática UPM, Madrid, Spain, December 1995.

[6] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[7] S.K. Debray, P. López García, M. Hermenegildo, and N.W. Lin. Estimating the Computational Cost of Logic Programs. In Springer-Verlag, editor, *Static Analysis Symposium, SAS'94*, LNCS vol. 864, pages 255–265, Namur, Belgium, September 1994.

[8] J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.

[9] P. López García, M. Hermenegildo, and S.K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computing, Special Issue on Parallel Symbolic Computation*, 1996. In press.

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

[11] F. Henderson, Z. Somogyi, and T. Conway. Determinism analysis in the Mercury compiler. In *Proc. Australian Computer Science Conference*, Melbourne, Jan. 1996.

[12] 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.

[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] 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.

[15] 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.

[16] Ju. V. Matijasevič, "Enumerable Sets are Diophantine", *Doklady Akademii Nauk SSSR*, 191 (1970), 279-282 (in Russian; English translation in *Soviet Mathematics—Doklady*, 11 (1970), 354-357).

[17] 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.

[18] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.