Successes in Logic Programs [1]

Annalisa Bossi, Nicoletta Cocco
Dip. di Matematica Applicata e Informatica
Università di Venezia-Ca' Foscari - Italy
bossi, cocco@dsi.unive.it

1 Introduction

When developing a logic program we, more or less explicitly, have to analize it with respect to correctness, termination properties and the existence of successful computations. On correctness and termination verification there have been rather many proposals such as [15, 7, 3, 11, 6, 18]. While for distinguishing failing and successful computations or for ensuring the presence of successes in a computation, very little is available, both in terms of methodology and tools. We can illustrate the problem through a few simple examples.

```
1) reverse([], []).
   reverse([X |Xs], Zs) ← reverse(Xs, Ys), append(Ys, [X], Zs).
```

with directional type $reverse(+ : List, - : List)$ and $app(+ : List, + : List, - : List)$.
This well-known program shows the simplest case of a programs which can be queried without producing finitely failing derivations (FDs). In a previous work [8] we have given a sufficient condition for ensuring that a given program and query are without failures, namely they cannot have finitely failing derivations (noFD). Our condition is very restrictive, since we want it to be verifiable from the text of the program and failures are difficult to be syntactically characterized. The requirement of not having failures can be useful in program transformations and when reasoning on efficiency, but it is too strong in general. In fact in most cases we are satisfied when a program has at least a successful derivation for a given query. All programs using *test predicates* cannot be noFD, but they can have successes.

```
2) delete([], X, []).
   delete([X |Xs], X, Ys) ← delete(Xs, X, Ys).
   delete([X |Xs], Z, [X |Ys]) ← X ≠ Z, delete(Xs, Z, Ys).
```

with directional type $delete(+ : List, + : Any, - : List)$.
This program can produce FDs since it contains *unification tests*, namely $X \neq Z$. Nevertheless we can prove that such tests are *exaustive on the input types*, and, since the program universally terminates with the specified directional type, it can be queried with at least one successful derivation.

```
3) gcd(X, 0, X).
   gcd(X, Y, Gcd) ← mod(X, Y, Z), gcd(Y, Z, Gcd).
   mod(X, Y, X) ← X<Y.
   mod(X, Y, Z) ← X>=Y, X1 is X-Y, mod(X1, Y, Z).
```

with directional types $gcd(+ : Int^+, + : Int^+, - : Int^+)$ and $mod(+ : Int^+, + : Int^+, - : Int^+)$, where Int^+ is the set of non-negative integers.
Also this program can have FDs since it contains *arithmetic tests*, namely $X < Z$ and $X >= Y$. But they are also exaustive on the input types; then, as in the previous example, it can be queried with at least one successful derivation.
In [14] a non-failure analysis is presented which captures these cases. It is undecidable in general if some tests "cover" a type, but for unification or arithmetic tests a sofisticated algorithm is supplied, which makes use of Abstract Interpretation techniques. A report on its implementation is also given.
In the present work we try to extend the results in [8, 14] with a methodological attitude. We study a sufficient condition for ensuring that a universally terminating query has at least one successful derivation in a given program. This corresponds to a large class of programs with test predicates, which we call

[1] Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-based Program Synthesis and Transformation, 15-19 June 1998, Manchester, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html

successful programs.
In the Mercury language [16] a nondeterminism analysis is implemented which has some analogies with our work. In our classification noFD plus successful predicates correspond to “det” plus “multi” modes in Mercury, but we distinguish predicates also depending on the presence of FDs, since this can be relevant in program transformations, as shown in [8].

4)
```
split([], Set2, [], []).
split(Set1, [], Set1, []).
split([X |Xs], Set2, [X |Complement], Intersection) ← not member(X, Set2),
   split(Xs, Set2, Complement, Intersection).
split([X |Xs], Set2, Complement, [X |Intersection]) ← member(X, Set2),
   split(Xs, Set2, Complement, Intersection).
member(X, [X |Xs]).
member(X, [Y |Ys]) ← member(X, Ys).
```

with directional types $split(+ : GroundList, + : GroundList, - : GroundList, - : GroundList)$ and $member(+ : Ground, + : GroundList)$.
This program can have FDs since it contains *general ground tests predicates*, namely $member(X, Set2)$ and *not* $member(X, Set2)$. But such tests are clearly exaustive on the input types and the specified directional types guarantee termination. Then we can prove that *split* can be queried with at least one successful derivation.
5) Similarly for the following predicate since $even(X)$ and $odd(X)$ are exaustive tests.

```
doubleEven([], []).
doubleEven([X |Xs], [X |Ys]) ← odd(X), doubleEven(Xs, Ys).
doubleEven([X |Xs], [Y |Ys]) ← even(X), Y is X*2, doubleEven(Xs, Ys).
```

with directional types $doubleEven(+ : NatList, - : NatList)$ and $even(+ : Nat)$, $odd(+ : Nat)$.
6) Now let us consider:

```
posPivot(L, Pos) ← oddLength(L, Length), Pos is Length div 2 +1.
posPivot(L, 0) ← evenLength(L, Length).
oddLength(L, N) ← length(L, N), odd(N).
evenLength(L, N) ← length(L, N), even(N).
```

with directional types $posPivot(+ : List, - : Nat)$, $evenLength(+ : List, - : Nat)$, $oddLength(+ : List, - : Nat)$ and $even(+ : Nat)$, $odd(+ : Nat)$.
In this program the tests predicates $evenLength(L, N)$ and $oddLength(L, N)$ do not require ground input (it is sufficient that the list in input has fixed length) and they supply output. Such predicates are also exaustive tests on the input types, then there are FDs, but we can prove that *posPivot* can be queried with at least one successful derivation.
7) As a last example let us consider:

```
choose(Low, High, Low).
choose(Low, High, X) ← Low < High, L is Low +1, choose(L, High, Low).
```

with directional types $choose(+ : Int, + : Int, - : Int)$.
This program is non-deterministic and it can have FDs since it contains tests predicates. But, as we will see in the following, we can simplify it and immediately ensure that it can be queried with at least a success.
In Section 2 we set some notation and recall the definitions of noFD program and query. In Section 3 the *class of successful predicates* is defined. We can prove that *a universally terminating noFD query, defined by successful predicates, has at least one successful LD-derivation.* In section 4 we describe a methodology for verifying that a predicate is successful.

2 Basic Notions

The notation is more or less standard and essentially follows [17, 1]. We make use of the notion of modes and types introduced in [13, 5, 2]. We consider a combination of modes and types and adopt the following assumption: *every considered relation has a fixed mode and a fixed type associated with it.* This assumption allows us to talk about types of *input positions* and of *output positions of an atom.* For example, $app(+ : List, + : List, - : List)$ denotes a ternary relation *app* with the first two positions moded as input and typed as $List$ and the third position moded as output and typed as $List$. A similar denotation is called *a directional type* in [12]. From [5, 2] we take also the notion of *well-typed query and program* and from [4] the notion of *simply moded clause and query.* A complete treatment of these topics can be found in [3]. Besides we need the following definitions.

Definition 2.1 [**output-free sequence**] Let P be a well-typed program and $\mathbf{Q} = B_1, \ldots, B_m$ a well-typed query in P. *The input (output) terms of* $\mathbf{Q}$, denoted by $In(\mathbf{Q})(Out(\mathbf{Q}))$, are the terms in input (output) positions in $\mathbf{Q}$.
$\mathbf{Q}$ is *output-free* iff *output positions are filled in with distinct variables which do not appear leftmost*, namely $\forall j \in [1, m]$, $Out(B_j) = Var(Out(B_j))$; if $X_1, \ldots, X_k$ are the output terms in B_j, then $X_i \neq X_h$, for $i \neq h$, $i, h \in [1, k]$; and $Out(B_j) \cap (Var(B_1, \ldots, B_{j-1}) \cup Var(In(B_j))) = \emptyset$. □

Definition 2.2 [**input-correct and output-free instance**] Let A be an output-free atom and α a substitution. We say that $A\alpha$ is an *input-correct and output-free instance* of A iff $A\alpha$ is correctly typed in input and $Out(A\alpha)$ is a new variant of $Out(A)$. □

In [8] we have defined a class of programs and queries which cannot have finitely failing derivations.

Definition 2.3 [**noFD program**] Let P be a well-typed program.
A clause $c : H \leftarrow A_1, \ldots, A_n$. *in* P *is without failures (noFD clause)* iff

1. c is simply moded (namely the sequence of atoms in the body, $A_1, \ldots, A_n$, is output-free and $Var(In(H)) \cap Var(Out(A_1, \ldots, A_n)) = \emptyset$);

2. for all $i \in [1, n]$ and for any input-correct and output-free instance $A_i\alpha$ of A_i, there exists a clause in P whose head unifies with $A_i\alpha$.

A predicate p *in* P *is without failures (noFD predicate)* iff all the clauses in (the deductive closure of) its definition in P are noFD ones. The program defining p is then *a program without failures (noFD program).* □

Definition 2.4 [**noFD query**] Let $\mathbf{Q} = B_1, \ldots, B_m$ be a well-typed query.
$\mathbf{Q}$ *is without failures (noFD query) in a program* P iff

1. the query $\mathbf{Q}$ is simply moded (namely it is output-free);

2. for $j \in [1, m]$ and for any input-correct and output-free instance $B_j\alpha$ of B_j, there exists a clause in P whose head unifies with $B_j\alpha$. □

In [8] we also proved the "persistency" of the noFD property through LD-derivation and the main result on absence of FDs.

Lemma 2.5 *Let* P *be a noFD program and* $\mathbf{Q}$ *a noFD query in* P. *Let us consider one non-trivial LD-derivation step* $\mathbf{Q} \overset{\sigma}{\longmapsto}_P \mathbf{Q}'$. *The query* $\mathbf{Q}'$ *is also noFD.* □

Theorem 2.6 *Let* P *be a noFD program and* $\mathbf{Q}$ *a noFD query in* P. *Then* $\mathbf{Q}$ *cannot have finitely failing derivations.* □

3 Successful Programs

In [8] we have seen how using tests in a program prevents it to be noFD. Tests predicates sometimes produce inefficiency, as in generate-and-test programs, and then it is better to avoid them. But more often they are necessary, as in a sorting algorithm. We then define a class which is wider than the one of noFD programs and which characterizes a weaker property: the programs in the class can have FDs but they can be queried with at least one successful derivation. We call it *the class of successful programs.*

Definition 3.1 [**exaustive tests**] Let P be a well-typed program wrt the directional types T and let p be a predicate defined in P by the clauses $p(\mathbf{t_i}) \leftarrow \mathbf{T_i}, \mathbf{B_i}.$, with $i \in [1, m]$.
$\mathbf{T_1}, \ldots, \mathbf{T_m}$ *are exaustive tests for* p *wrt input types in* T iff
for any well-typed input substitution for p, θ, namely such that $p(\mathbf{X})\theta$ is an input-correct and output-free instance of $p(\mathbf{X})$, there exists i, $i \in [1, m]$, such that $\sigma = mgu(p(\mathbf{X})\theta, p(\mathbf{t_i}))$ and $\mathbf{T_i}\sigma$ has at least one successful LD-derivation. □

Example 3.2 Let us consider the following trivial program

```
p(X, Y, Z) ← X = Y, r(1, Z).
p(a, B, Z) ← any(B), r(2, Z).
p(b, a, Z) ← r(3, Z).
p(b, c, Z) ← r(4, Z).
any(b).
any(c).
r(X, Y) ← ...
```

with directional types: $T = p(+ : \{a, b\}, + : \{a, b, c\}, - : R), any(+ : \{a, b, c\}), r(+ : \{1, 2, 3, 4\}, - : R)$.
Then $\{X = Y, any(B), true, true\}$ are exaustive tests for p wrt input types in T. In fact well-typed input substitutions θ for $p(X1, X2, X3)$ are such that $X1\theta \in \{a, b\}$ and $X2\theta \in \{a, b, c\}$.
Let us consider all possible cases.
1) $\theta = \{X1/a, X2/a\}$ or $\theta = \{X1/b, X2/b\}$, then $\sigma = mgu(p(X, Y, Z), p(X1, X2, X3)\theta)$, with $\sigma = \{X/a, Y/a\}$ or $\sigma = \{X/b, Y/b\}$ and in both cases the test $(X = Y)\sigma$ is successful.
2) $\theta = \{X1/a, X2/b\}$ or $\theta = \{X1/a, X2/c\}$, then $\sigma = mgu(p(X, B, Z), p(X1, X2, X3)\theta)$, with $\sigma = \{X/a, B/b\}$ or $\sigma = \{X/a, B/c\}$ and in both cases the test $any(B)\sigma$ is successful.
3) $\theta = \{X1/b, X2/a\}$, then $\sigma = mgu(p(b, a, Z), p(X1, X2, X3)\theta)$, with $\sigma = \emptyset$ and the test $true\sigma$ is successful.
4) $\theta = \{X1/b, X2/c\}$, then $\sigma = mgu(p(b, c, Z), p(X1, X2, X3)\theta)$, with $\sigma = \emptyset$ and the test $true\sigma$ is successful.
Since for any well-typed input substitution θ the condition in Definition 3.1 is satisfied, the tests are exaustive. □

Note that *we can always introduce a dummy test "true" in a clause* in order to associate a set of exaustive tests to a predicate p with types in T. For example when p is defined also by facts, the introduction of the dummy test "true" is necessary for satisfying Definition 3.1.

Definition 3.3 [**successful predicate**] Let P be a well-typed program wrt the directional types T.
Let $p(\mathbf{t_i}) \leftarrow \mathbf{T_i}, \mathbf{B_i}.$, with $i \in [1, m]$, be the clauses defining a predicate p in P.
A predicate p in P is *successful* wrt the input types in T iff

1. $\mathbf{T_1}, \ldots, \mathbf{T_m}$ are exaustive tests for p wrt input types in T;
2. (*noFD conditions*) for $i \in [1, m]$
   - the sequence $\mathbf{B_i}$ is output-free;
   - $Var(Out(\mathbf{B_i}) \cap (Var(In(p(\mathbf{t_i}))) \cup Var(\mathbf{T_i})) = \emptyset$;
   - for all $D \in \mathbf{B_i}$ and for any input-correct and output-free instance $D\alpha$ of D, there exists a clause in P whose head unifies with $D\alpha$;

- all the predicates in $\mathbf{B_i}$ are successful. □

The intuitive idea is that if p is a successful predicate, for any query $p(\mathbf{t})$, input-correct and output-free, there is at least a clause in the definition of p which is applicable, whose tests are successful and which produces a non-finitely failing derivation (noFD conditions on the atoms in the body which follow the tests). (3), (4), (5) and (6) in the introduction are all successful predicates.
Note that we require that tests are leftmost in the clause bodies. This is clearly a restriction, since tests can appear also in other positions. For example let us consider a non-tail recursive definition of the maximum element in a list of integers. On the other hand this requirement simplifies the definition of successful predicates.
Note also that if p is a noFD predicate and for any well-typed input substitution θ, $p(\mathbf{X})\theta$ is a noFD query, then p is also a successful predicate. In fact we can always introduce the dummy test *true* in all the clauses defining p. The predicate is defined for any well-typed input, hence we have tests which are trivially exaustive. For this reason also (1) in the introduction is a successful predicate.
For successful predicates we can prove that any noFD query which universally terminates has at least one successful derivation. The proof is in [9].

Theorem 3.4 *Let* $\mathbf{Q}$ *be a noFD query in* P*. If* $\mathbf{Q}$ *is universally terminating in* P *and all the predicates in* $\mathbf{Q}$ *are successful in* P *wrt the input types, then* $\mathbf{Q}$ *has at least one successful LD-derivation in* P*.* □

4 How to Verify that a Program is Successful

The conditions in Definition 3.3 are very restrictive and not so easy to be satisfied. One reason is that often the directional type associated to a predicate is actually larger than the type for which the predicate has been defined. Hence we can extend Definition 3.3 by introducing the notion of *actual input types*.

Definition 4.1 [**actual input types**] Let P be a well-typed program and p a predicate defined in P by the clauses $p(t_{i1},\ldots,t_{in},s_{i1},\ldots,s_{ir}) \leftarrow \mathbf{B_i}.$, with $i \in [1,m]$, where t_{ik}, $k \in [1,n]$, are input terms and s_{ih}, $h \in [1,r]$, output terms. Let T_k, for $k \in [1,n]$, be the types associated to the input positions in the directional type of p.
The actual input types of p*,* AT_k*, for* $k \in [1,n]$, are given by the sets of all the instances of terms in the k-th input positions of the heads of the definitions of p which belongs to T_k:
$AT_k = (\cup_\beta\{t_{1k}\beta,\ldots,t_{mk}\beta\}) \cap T_k$. □

Example 4.2 Let us consider the predicate defining the last element of a list:

```
last([X], X).
last([X, Y |Xs], Z)  ←  last([Y |Xs], Z).
```

with the directional type $last(+ : List, - : Any)$. The actual input type of *last* is:
$AT_1 = (\cup_\beta\{[X]\beta, [X,Y \mid Xs]\beta\}) \cap List$, namely the set of non-empty lists: $List - \{[]\}$.
This predicate is noFD, as one can easily verify. Nevertheless it is not successful wrt its input type. Let us introduce two dummy tests *true*, leftmost in the clauses, as usual. Such tests are not exaustive, in fact Definition 3.1 requires that also for the query $last([], Y)$, which is well-typed in input, there is at least a unifying clause with a successful test. On the other hand *last* is successful wrt its actual input type. □

A major difficulty in using Definition 3.3 is the identification of exaustive tests. To make it easier we introduce a transformation for normalizing input arguments in a program.

Definition 4.3 [**input-normalized definition**] Let P be a well-typed program and $p(i_1,\ldots,i_n,o_1,\ldots,o_r)$ a predicate defined in P, where i_k, with $k \in [1,n]$, are the input positions and o_h, with $h \in [1,r]$ the output positions in p. Let $V = \{X_1,\ldots,X_n\}$ be a set of new variables wrt p, namely variables which do not appear in the definition of p.
The input-normalized definition of p*,* p^N, is the program obtained by substituting each clause in the definition of p: $p(t_1,\ldots,t_n,s_1,\ldots,s_r) \leftarrow \mathbf{D}$.
with the corresponding *input-normalized clause*: $p(X_1,\ldots,X_n,s_1,\ldots,s_r) \leftarrow X_1 = t_1,\ldots,X_n = t_n,\mathbf{D}$.

The program thus obtained is *input-normalized wrt p. V is the set of normalized input variables for p^N.* $X_k = t_k$, for $k \in [1, n]$, are *the input equalities in each clause in the definition of p^N*, with directional type $= (+ : T_k, - : T_k)$, where T_k is the type associated to the k-th input position. □

Input normalization preserves well-typing and produces a program which is clause-wise equivalent to the original one. We assume that *input-normalized programs always refer to the actual input types of the original program.*

Example 4.4 Let us consider the input-normalized version of the program in the example 4.2, $last^N$:

$$\texttt{last}(X_1,\ \texttt{X}) \leftarrow X_1 = \texttt{[X]}.$$
$$\texttt{last}(X_1,\ \texttt{Z}) \leftarrow X_1 = \texttt{[X, Y |Xs], last([Y |Xs], Z)}.$$

If the query has a non-empty list in input, namely its input term is in the actual input type of the original program, then one of the input equalities succeeds. □

We give here the definitions of exaustive tests and successful program extended in order to consider actual input types and normalization. These extended conditions are generally easier to verify that the ones in the previous Definitions 3.1 and 3.3.

Definition 4.5 [**exaustive tests (extended)**] Let P be a well-typed program, p a predicate defined in P and $AT = (AT_1, \ldots, AT_n)$ its actual input types. Let p^N be its input-normalized definition $p(X_1, \ldots, X_n, \mathbf{s_i}) \leftarrow \mathbf{T_i}, \mathbf{B_i}.$, with $i \in [1, m]$, and $V = \{X_1, \ldots, X_n\}$ be the set of normalized input variables for p^N.
$\mathbf{T_1}, \ldots, \mathbf{T_m}$ *are exaustive tests for p^N wrt AT* iff
for all well-typed input substitutions θ wrt AT, namely such that $Dom(\theta) = V$ and $X_k\theta \in AT_k$, for $k \in [1, n]$, there exists i, $i \in [1, m]$, such that $\mathbf{T_i}\theta$ has at least one successful LD-derivation. □

Definition 4.6 [**successful predicate (extended)**] Let P be a well-typed program and p a predicate in P with actual input types AT. Let $p(\mathbf{t_i}) \leftarrow \mathbf{T_i}, \mathbf{B_i}.$, with $i \in [1, m]$, be the clauses defining p^N in P.
p in P is *successful wrt its actual input types* iff

1. $\mathbf{T_1}, \ldots, \mathbf{T_m}$ are exaustive tests for p^N wrt AT;
2. (*noFD conditions*) for $i \in [1, m]$
   - the sequence $\mathbf{B_i}$ is output-free;
   - $Var(Out(\mathbf{B_i})) \cap (Var(In(p(\mathbf{t_i}))) \cup Var(\mathbf{T_i})) = \emptyset$;
   - for all $D \in \mathbf{B_i}$ and for any input-correct wrt AT and output-free instance $D\alpha$ of D, there exists a clause in P whose head unifies with $D\alpha$;
   - all the predicates in $\mathbf{B_i}$ are successful. □

With this extended definition, it is possible to prove that also (2) in the introduction is successful. In fact by the previous definition, we can determine tests predicates by composing the input equalities introduced by input-normalization with atoms in the clause bodies for which noFD conditions are not satisfied.
The property of having exaustive tests is a semantic condition and not decidable in general, as shown in [14], but in most practical cases the conditions in Definition 4.6 can be easily verified. In order to make easier to prove the property of being exaustive tests for a predicate, it is often useful to "simplify" the set of clauses we consider. To this purpose we introduce the following definition.

Definition 4.7 [**Clause input-subsumed by a fact or a clause**] Let P be a well-typed program. Let $c_1 : A.$ be a fact, $c_2 : H \leftarrow \mathbf{B}.$ and $c_3 : K \leftarrow \mathbf{A}.$ be clauses in P which define the same predicate, namely $A = p(\mathbf{t_1})$, $H = p(\mathbf{t_2})$ and $K = p(\mathbf{t_3})$.
Clause c_2 is input-subsumed by the fact c_1 iff there exists an input substitution θ such that $A\theta = H$.
Clause c_3 is input-subsumed by clause c_2 iff there exists an input substitution θ such that $H\theta = K$. □

We have extended the usual concept of subsumption to fit the verification of the successful property. We are not interested in c.a.s. but in characterizing the presence of at least one successful derivation for each well-typed input substitution. A fact always produces a successful derivation, then it can input-subsume a clause independently from its body. Similarly for two clauses, if the one accepting a more general input produces a successful derivation, then the other one becomes irrelevant. Input-subsumption can be used to prune the program. If we can prove that a pruned definition of a predicate is successful, then also the original definition is successful. This means that we may simplify the verification of Definition 4.6 by first pruning the program of some input-subsumed clauses. The simplification phase must be applied before input-normalization; after input-normalization input-subsumption becomes trivial and useless.
In (7) in the introduction, the second clause is input-subsumed by the first one which is a fact and then noFD. Hence we can conclude that also (7) is successful.

Given a program P, **the finite failure analysis** could proceed as follows:

- *if the predicate p is noFD (Definition 2.3) then OK;*
- *else repeat*
  - *prune p by eliminating some input-subsumed clauses and determine actual input types;*
  - *input-normalize the resulting program;*
  - *single out the tests in the bodies*
    {the input equalities plus the body atoms which do not satisfy the noFD conditions};

  until (such tests are exaustive) and (the noFD conditions in Definition 4.6 are verified).

The property of beeing successful, even if weaker than the one of being noFD, can be useful in the field of *program verification*. Let us assume to have a program which is partially correct wrt a specification and a query which is both universally terminating and partially correct wrt the specification. Then, if we prove that the query is noFD and the predicates in the query are successful, Theorem 3.4 ensures the existence of at least one correct c.a.s.

Also in the field of *program transformations* being successful can be useful in particular for switching atoms in the clause bodies. In [10] we consider left-terminating programs [6] and we define a sufficient condition for switching atoms while preserving left-termination. This condition for switching the atom A with B in a clause $c : H \leftarrow \mathbf{J}, A, B, \mathbf{K}$. depends on the property of A of being "non-failing", namely *for each grounding substitution θ, such that $Dom(\theta) = Var(In(H), \mathbf{J}, In(A))$ and $\mathbf{J}\theta \in M_P$, there exists γ such that $A\theta\gamma \in M_P$*. This intuitively means that if A is the selected atom in an LD-derivation, because c was used as input clause in a previous derivation step, then the computation of A will eventually succeed. This is a "semantic" condition which is not easy to verify in general. But *if we prove that the query A is universally terminating and noFD and the predicate in A is successful, then the non-failing condition is guaranteed.*

Example 4.8 Let us consider an example of transformation given in [10]. The following program defines the predicate $goodpath(X, Xs)$ which relates a node X with a list Xs of "good" nodes which are connected to X.

```
1: path(X, [X]).
2: path(X, [X |Xs]) ← arc(X, Y), path(Y, Xs).
3: goodlist([]).
4: goodlist([X |Xs]) ← good(X), goodlist(Xs).
d: goodpath(X, Xs) ← path(X, Xs), goodlist(Xs).
```

The directional types are $path(+ : Const, - : List), goodlist(+ : List), goodpath(+ : Const, - : List)$, $arc(+ : Const, - : Const)$, $good(+ : Const)$.
The predicates *good* and *arc* are test predicates defined in the program.
In [10] we consider a transformation which repeatedly unfold the body atoms in d until we get to:

```
d1:goodpath(X, [X]) ← good(X).
d2:goodpath(X, [X |Xs]) ← arc(X, Y),
      path(Y, Xs), good(X),
      goodlist(Xs).
```

In order to optimize the program by folding with d, we need to switch $path(Y, Xs)$ and $good(X)$ in $d2$. Hence we need to prove that $path(Y, Xs)$ is non-failing in d2, namely that

a) for each grounding substitution θ, such that $Dom(\theta) = \{X, Y\}$ and $Y\theta \in Const$, $path(Y, Xs)\theta$ is a noFD query and

b) *path* is a successful predicate.

(a) is trivial. For (b) let us consider the definition of *path*. We can prune 2, since it is input-subsumed by the fact 1. Hence we obtain a noFD program, which implies that *path* is successful. □

References

[1] K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[2] K. R. Apt. Declarative programming in Prolog. In D. Miller, editor, *Proceedings of the 1993 International Symposium on Logic Programming*, pages 12–35. The MIT Press, 1993.

[3] K. R. Apt. *From Logic Programming to Prolog.* Prentice Hall International Series in Computer Science, 1997.

[4] K. R. Apt and S. Etalle. On the unification free Prolog programs. In A. Borzyszkowski and S. Sokolowski, editors, *Proceedings of the Conference on Mathematical Foundations of Computer Science (MFCS 93)*, Lecture Notes in Computer Science, pages 1–19, Berlin, 1993. Springer-Verlag.

[5] K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6A):743–765, 1994.

[6] K. R. Apt and D. Pedreschi. Studies in Pure Prolog: termination. In J.W. Lloyd, editor, *Proceedings of the Simposium in Computational Logic*, pages 150–176, Berlin, 1990. Springer-Verlag.

[7] A. Bossi and N. Cocco. Verifying correctness of logic programs. In J. Diaz and F. Orejas, editors, *TAPSOFT '89, Barcelona, Spain, March 1989, (Lecture Notes in Computer Science, vol. 352)*, pages 96–110. Springer-Verlag, 1989.

[8] A. Bossi and N. Cocco. Programs without Failures. In N. Fuchs, editor, *Proceedings LOPSTR'97*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1997.

[9] A. Bossi and N. Cocco. Successful Logic Programs. Technical Report CS298, CS Dept. - Venice University, February 1998.

[10] A. Bossi, N. Cocco, and S. Etalle. Transformation of Left Terminating Programs: The Reordering Problem. In M. Proietti, editor, *Proceedings LOPSTR'95*, volume 1048 of *Lecture Notes in Computer Science*, pages 33–45. Springer-Verlag, Berlin, 1996.

[11] A. Bossi, N. Cocco, and M. Fabris. Norms on terms and their use in proving universal termination of a logic program. *Theoretical Computer Science*, 124:297–328, 1994.

[12] J. Boye and J. Maluszynski. Two Aspects of Directional Types. In Sterling, editor, *Proc. Int'l Conf. on Logic Programming*, pages 747–761. MIT Press, 1995.

[13] F. Bronsard, T. K. Lakshman, and U. S. Reddy. A framework of directionalities for proving termination of logic programs. In K. R. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 321–335. The MIT Press, 1992.

[14] S. Debray, P. Lopez-Garcia, and M.Hermenegildo. Non-Failure Analysis for Logic Programs. In *Proceedings of the International Symposium on Logic Programming*, pages 48–62, 1997.

[15] W. Drabent and J. Maluszynski. Inductive assertion method for logic programs. *Theoretical Computer Science*, 59:133–155, 1988.

[16] F. Henderson, T. Conway, Z.Somogyi, and D. Jeffery. The Mercury language reference manual. Technical Report TR 96/10, Dep. of Computer Science, University of Melbourne, 1996.

[17] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.

[18] D. De Schreye and S. Decorte. Termination of Logic Programs: the never-ending story. *Journal of Logic Programming*, 19-20:199–260, 1994.