Beyond Success and Failure

Sandro Etalle
Dept. of Computer Science, Universiteit Maastricht
P.O. Box 616, 6200 MD Maastricht, The Netherlands
etalle@cs.unimaas.nl

Femke van Raamsdonk
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
femke@cwi.nl

Abstract

We study a new programming framework based on logic programming where success and failure are replaced by predicates for adequacy and inadequacy. Adequacy allows to extract a result from a partial computation, and inadequacy allows to flexibly constrain the search space. In this parameterized setting, the classical result of independence of the selection rule does not hold. We show that, under certain conditions, whenever there exists an adequate derivation there is one in which only so-called needed atoms are selected. This result is applied in a practical setting where adequacy is expressed using a notion of request.

1 Introduction

In logic programming, success and failure determine when the computation should stop and return the result and when it should start backtracking. In this sense, success and failure form the cornerstones of control. In the traditional approach, a resolution sequence is considered to be successful if it ends in the empty query, and failing if the resulting query is non-empty but no clause can be applied to the selected atom. In the present paper we propose to take a more general point of view by parameterizing over notions of success and failure, which we then call *adequacy* and *inadequacy*.

The important point of using adequacy is that it permits to extract a result from an incomplete computation, which is particularly useful in the presence of potentially infinite data structures. As an example we consider the following program, which computes the list of Fibonacci numbers.

$$\begin{aligned} fib([x_n, x_{n+1}, x_{n+2}|l]) \quad \leftarrow \quad & x_{n+2} \ is \ x_n + x_{n+1}, \\ & fib([x_{n+1}, x_{n+2}|l]). \end{aligned}$$

Suppose we are interested in computing the fifth Fibonacci number. Then we start computing in the query $fib([0, 1, \_, \_, x|l])$ and specify that a computation is adequate if it computes the value of x. The derivation

$$fib([0,1,\_,\_,x|l]) \Rightarrow^* fib([1,1,\_,x|l]) \Rightarrow^* fib([1,2,x|l]) \Rightarrow^* fib([2,3,l]).$$

instantiates x to 3 and is hence considered to be adequate, although it is not successful in the traditional sense.

The inadequacy predicate can be used to cut away an infinite branch never leading to an adequate state. Consider for instance the program defined by the following clauses:

$$\begin{array}{rcl} p(f(y)) & \leftarrow & . \\ p(a) & \leftarrow & . \\ q & \leftarrow & q. \end{array}$$

Consider the query $p(x), q$ and suppose that a derivation is considered to be adequate if it computes the value of x. Consider the derivation

$$\langle p(x), q; \epsilon\rangle \Rightarrow \langle q; x \mapsto f(y)\rangle.$$

It is clear that there is no way to extend this derivation to an adequate one, because y will never be instantiated and the only possible extension is to enter the one-step loop $\langle q; x \mapsto f(y)\rangle \Rightarrow \langle q; x \mapsto f(y)\rangle$. At this point it becomes extremely useful to be able to backtrack instead of entering an infinite branch. This is possible by specifying an inadequacy predicate as follows: the inadequacy predicate $\mathcal{I}$ holds for a state $\langle Q; \sigma\rangle$ if $x\sigma$ is not ground and the variables in $x\sigma$ do not occur in Q.

It is easy to see that in this general setting independence of the selection rule does not hold. Hence it is important to investigate when appropriate selection rules can be devised. Following the approach initiated by Huet and Lévy in [5] in the context of rewriting, we introduce needed atoms, and we show that, under certain conditions, whenever there is an adequate derivation, there is one in which only needed atoms are selected. This result is obtained in a quite abstract setting. We then introduce for logic programs with guards and delay declarations the notion of request, which allows to formulate a notion of adequacy that is useful in practice, and we formulate conditions that permit to apply the result concerning needed atoms.

The remainder of the paper is organized as follows. In the next section we fix the notation. In Section 3 we discuss our approach of parameterizing over success and failure by means of predicates for adequacy and inadequacy. In Section 4 the notion of needed atoms is introduced, and we show that, under certain conditions, there is an adequate derivation in which only needed atoms are selected whenever there is an adequate derivation. Section 5 is concerned with logic programming with guards and delay declarations enriched with requests. Finally we discuss some other possible uses of the (in)adequacy predicates.

2 Preliminaries

We assume a set $\mathcal{V}$ of infinitely many *variables* written as $x, y, z, \ldots$. We write *function symbols* as $f, g, a, b, \ldots$ and *relation symbols* as $p, q, \ldots$. Sometimes we use more suggestive notation. Both function and relation symbols are supposed to have a fixed arity. If $\mathcal{F}$ is a set of function symbols, then *terms* over $\mathcal{F}$ are built from variables and function symbols in $\mathcal{F}$ in the usual way. If in addition $\mathcal{R}$ is a set of relation symbols, then *atoms* over $\mathcal{F}$ and $\mathcal{R}$ are built from terms over $\mathcal{F}$ and relation symbols in $\mathcal{R}$ in the usual way. Terms are written as $s, t, \ldots$ and atoms as $A, B, H, \ldots$. In the sequel, we won't specify the sets of function and predicate symbols if no confusion can arise. A *query* is defined as a finite sequence of atoms. Queries are denoted by $Q, \ldots$ or by specifying a finite sequence of atoms. A sequence of atoms $A_1, \ldots, A_n$ with $n \geq 0$ is often abbreviated to $\vec{A}$. The empty query is denoted by $\Box$. A *substitution* is a mapping from variables to terms, written as $\sigma, \tau, \ldots$, that is extended to terms and atoms homeomorphically. The identity substitution is denoted by ϵ. We will consider pairs consisting of a query and a substitution, written as $\langle Q; \sigma \rangle$, and call such pairs *states*. Instead of states one often considers queries only. The dynamics of logic programming is defined using clauses. A *clause* is an expression of the form $H \leftarrow B_1, \ldots, B_m$ with $H, B_1, \ldots, B_m$ atoms. A set of clauses is also called a *program*.

3 Success and Failure Revisited

In the approach we propose, the control part of a logic program is specified by two *control predicates:* one for *adequacy* and one for *inadequacy*. The adequacy predicate is used to express that a computation has yield sufficient information, and the inadequacy predicate to express that it is no use to continue the computation under consideration. Before we discuss the meaning and use of these predicates, we first need to introduce some more notions.

In logic programming languages with a dynamic selection rule there is often a way to express whether an atom can be selected. This can for instance be done by means of delay declarations, or, somewhat more implicitly, using guards. These two concepts will be considered in Section 5. Until then, it is sufficient to consider an abstract predicate for selectability.

Definition 3.1 Let a program $\mathcal{P}$ be given. A predicate $\mathcal{S}$ defined on atoms is said to be a *selectability predicate* if it satisfies the following requirements:

1. if $\mathcal{S}(A)$ holds then for every clause $H \leftarrow \vec{B}$ in $\mathcal{P}$ it is decidable whether A and H unify,
2. if $\mathcal{S}(A)$ holds then $\mathcal{S}(A\sigma)$ holds for every substitution σ. $\Box$

The first requirement makes sense for instance when considering guarded unification; note further that the selectability predicate is required to be monotonic.

A *selection rule* is a mapping that given a state returns a selectable atom in that state whenever the state contains a selectable atom, and returns δ otherwise.

In the following we will consider predicates $\mathcal{A}$ for adequacy and $\mathcal{I}$ for inadequacy. These predicates depend on the program and the selection rule under consideration; note that the latter in its turn depend on the selectability predicate. In the notation these parameters are suppressed: we denote by $\mathcal{A}(\langle Q; \sigma \rangle)$ that a state $\langle Q; \sigma \rangle$ is adequate and by $\mathcal{I}(\langle Q; \sigma \rangle)$ that it is inadequate. The (in)adequacy predicates are subject to the following three assumptions. We will suppose that

1. *(in)adequacy predicates are invariant under renaming,*
2. *there is no state in which both $\mathcal{A}$ and $\mathcal{I}$ hold,*
3. *if there is no clause the head of which unifies with the selected atom, then $\mathcal{I}$ holds.*

If a state is neither adequate nor inadequate and applying the selection rule to it yields δ, then we say that it is in *deadlock*. Note that it can be the case that inadequacy is defined to hold if the selection rule yields δ; in that case no state is in deadlock. By definition, adequacy, inadequacy and deadlock are mutually exclusive.

Now the dynamics are defined as follows.

Definition 3.2 Let $\mathcal{P}$ be a program. Suppose that

1. $\langle Q; \sigma \rangle$ is a state that is neither adequate, nor inadequate, nor in deadlock,
2. $Q = A_1, \ldots, A_{m-1}, A_m, A_{m+1}, \ldots, A_n$ with A_m the selected atom,
3. $c = H \leftarrow \vec{B}$, is a suitable renaming of a clause in $\mathcal{P}$,
4. τ is a most general unifier of A_m and H.

Then we have a *derivation step*

$$\langle Q; \sigma \rangle \overset{\tau}{\underset{c}{\Longrightarrow}} \langle Q'; \sigma\tau \rangle$$

with $Q' = (A_1, \ldots, A_{m-1}, \vec{B}, A_{m+1}, \ldots, A_n)\tau$.

We say that the clause c is *applied* to A_m in $\langle Q; \sigma \rangle$. For every $i \in \{1, \ldots, m-1, m+1, \ldots, n\}$, the atom $A_i\tau$ in Q' is said to be the *descendant* of the atom A_i in the query Q. The atoms $\vec{B}\tau$ in Q' are said to be *created* in the derivation step. □

We omit the labels specifying the most general unifier and/or the clause whenever possible. Further, we usually specify only the relevant part of a substitution in a state. A sequence of derivation steps is called a *derivation*. Derivations can be finite or infinite. The definition of descendant is extended in a natural way to derivations consisting of more than one derivation step. We make use of the following terminology.

Definition 3.3 Let $\mathcal{P}$ be a program and let an adequacy predicate $\mathcal{A}$ and an inadequacy predicate $\mathcal{I}$ be given. Let

$$\xi : \langle Q_1; \sigma_1\rangle \Rightarrow^* \langle Q_m; \sigma_m\rangle$$

be a finite derivation.

1. ξ is said to be *adequate* if $\mathcal{A}(\langle Q_m; \sigma_m\rangle)$ holds.
2. ξ is said to be *inadequate* if $\mathcal{I}(\langle Q_m; \sigma_m\rangle)$ holds.
3. ξ is said to *deadlock* if $\langle Q; \sigma\rangle$ is in deadlock. □

The standard approach to logic programming is obtained by specifying that $\mathcal{A}$ coincides with the usual notion of *success*, that is, it holds for $\langle Q; \sigma\rangle$ and $\mathcal{P}$ if and only if $Q = \Box$, and that $\mathcal{I}$ coincides with the usual notion of *failure*, that is, it holds for $\langle Q; \sigma\rangle$ and $\mathcal{P}$ if and only if $Q \neq \Box$ and no clause in $\mathcal{P}$ can be applied to the selected atom. The general, parameterized, setting permits however to express many more notions of (in)adequacy, as discussed in the introduction.

Non-determinism. As is well-known, a derivation step is subject to four dimensions of non-determinism (c.f. Section 3.5 of [2]):

1. the choice of a renaming of the program clause used,
2. the choice of a most general unifier,
3. the choice of an atom,
4. the choice of a program clause.

Since (in)adequacy predicates are supposed to be invariant under renaming, it follows as for the classical approach to logic programming that neither the choice of a renaming of a clause nor the choice of a most general unifier is relevant. Further, the choice of a program clause yields non-determinism as in the classical approach.

The essential difference with the classical approach lies in the third dimension of non-determinism, the choice of an atom. In the classical approach, an important result is the independence of the selection rule. More precisely, it is shown (see [2]), that for every successful derivation ξ and for every selection rule R, there exists a derivation that is via R, successful, has the same length as ξ and that yields the same computer answer substitution as ξ. This result fails in the present, generalized setting. Consider for instance the program defined by the following clauses:

$$\begin{array}{rcl} q & \leftarrow & q. \\ p(a) & \leftarrow & . \end{array}$$

Suppose that we specify our adequacy condition to hold if and only if x is instantiated to a ground term. Then

$$\langle q, p(x); \epsilon\rangle \Rightarrow \langle q; x \mapsto a\rangle$$

is an adequate derivation, but there is no adequate derivation using the leftmost selection rule.

4 Call by Need

In the light of the negative result concerning independence of the selection rule, it is important to identify useful selection rules. Our approach is inspired by a line of research initiated by Huet and Lévy in [5] (for more recent contributions concerning also notions of partial result see for instance [6, 4]). They show in [5] that in an orthogonal term rewriting system, every term not in normal form contains a needed redex, and that repeatedly contracting needed redexes yields a rewrite sequence ending in normal form, whenever the initial term has a normal form. Intuitively speaking, the contraction of a needed redex cannot be avoided and needed rewriting is normalizing. Needed redexes are not computable.

In this section we define needed atoms as atoms that must be resolved in order to obtain an adequate state. We identify conditions on the selectability predicate and the (in)adequacy predicates that guarantee that if a state admits an adequate derivation, then it admits one in which only needed atoms are selected. We call such a derivation a *call by need* derivation. Note that due to the fourth dimension of non-determinism (the choice of the program clause), we cannot hope that every derivation via the call by need selection rule is adequate.

Needed atoms. Needed atoms are defined using the notion of descendant (see Definition 3.2).

Definition 4.1 Let $\mathcal{P}$ be a program and let $\mathcal{A}$ be an adequacy predicate. An atom A in a state $\langle Q; \sigma\rangle$ is *needed* with respect to $\mathcal{A}$ if for every adequate derivation $\langle Q; \sigma\rangle \Rightarrow^* \langle Q'; \sigma'\rangle$ we have that Q' does not contain a descendant of A. □

Note that in a state that is not the starting point of an adequate derivation, every atom is needed.

Example 4.2

1. If we take for adequacy the traditional notion of success, then in any state $\langle Q;\sigma\rangle$ every atom in Q is needed.
2. If $\mathcal{A}(\langle A;\sigma\rangle)$ does not hold, then A is needed. □

The first part of the previous example reveals that needed atoms are simply not interesting in the traditional approach to logic programming.

Requirements. What we aim at now, is to identify requirements that guarantee the following property to hold: if there is an adequate derivation, then there is one in which only needed atoms are selected. This is not the case in general. It can be the case that needed atoms do not exist. Moreover, problems occur when atoms that are not needed can make needed atoms available. This can happen in three ways: by creating an atom (in the sense of Definition 3.2), by turning an atom that is not needed into a needed one, and by turning a non-selectable needed atom into a selectable one. This is illustrated in the following example.

Example 4.3

1. Consider the program $\mathcal{P}$ defined by the clauses

$$\begin{array}{lcl} a & \leftarrow & c \\ b & \leftarrow & c \end{array}$$

Let the adequacy predicate $\mathcal{A}$ be defined as follows: $\mathcal{A}(\langle Q;\sigma\rangle)$ holds if and only if Q contains an atom c. Consider the state $\langle a,b;\epsilon\rangle$. Neither a nor b is a needed atom, since both $\langle a,b;\epsilon\rangle \Rightarrow \langle c,b;\epsilon\rangle$ and $\langle a,b;\epsilon\rangle \Rightarrow \langle a,c;\epsilon\rangle$ are adequate derivations. Hence the state $\langle a,b;\epsilon\rangle$ does not contain a needed atom.

2. Consider the program $\mathcal{P}$ defined by the clauses

$$\begin{array}{lcl} a & \leftarrow & a' \\ a' & \leftarrow & a'' \\ b & \leftarrow & b' \\ b' & \leftarrow & b'' \end{array}$$

Let the adequacy predicate $\mathcal{A}$ be defined as follows: $\mathcal{A}(\langle Q;\sigma\rangle)$ holds if and only if $Q \in \{(a'',b),(a,b'')\}$. Then the state $\langle a,b;\epsilon\rangle$ does not contain a needed atom, since we have adequate derivations $\langle a,b;\epsilon\rangle \Rightarrow \langle a',b;\epsilon\rangle \Rightarrow \langle a'',b;\epsilon\rangle$ and $\langle a,b;\epsilon\rangle \Rightarrow \langle a,b';\epsilon\rangle \Rightarrow \langle a,b'';\epsilon\rangle$. Note that in this case resolving an unneeded atom creates a needed atom.

3. Consider the program defined by the clauses

$$\begin{array}{rl} q(a,b) & \leftarrow \\ q(a',b') & \leftarrow \\ p(a) & \leftarrow \\ p'(a') & \leftarrow \end{array}$$

Suppose that $\mathcal{S}(q(s,t))$ holds only if s is a ground term. Suppose further that a derivation starting in $\langle p(x), p'(x), q(x,y); \epsilon\rangle$ is adequate only if y is instantiated to a ground term. The query $p(x), p'(x), q(x,y)$ does not contain a needed atom, essentially because there is more than one way to make a descendant of $q(x,y)$ selectable. □

This example motivates the following two definitions; the first one concerns the predicates for (in)adequacy and the second one concerns the selectability predicate.

In the rest of this section we refer to an fixed unspecified program $\mathcal{P}$ and set of queries $\mathcal{Q}$, which we assume to be *closed* with respect to $\mathcal{P}$, i.e. that for every derivation step $\langle Q;\sigma\rangle \Rightarrow \langle Q';\sigma\tau\rangle$, using a clause of $\mathcal{P}$ we have that $Q \in \mathcal{Q}$ implies $Q' \in \mathcal{Q}$.

Definition 4.4 (In)adequacy predicates $\mathcal{A}$ and $\mathcal{I}$ are said to be *serializable* if for each $Q \in \mathcal{Q}$ the following conditions are satisfied.

1. If $\xi = \langle Q;\sigma\rangle \Rightarrow \langle Q';\sigma'\rangle$ is a derivation step with $\mathcal{A}(\langle Q';\sigma'\rangle)$ and not $\mathcal{A}(\langle Q;\sigma\rangle)$, then the atom selected in ξ is needed.
2. Let $\xi = \langle Q;\sigma\rangle \Rightarrow \langle Q';\sigma'\rangle$ be a derivation step in which the selected atom is not needed, and suppose that $\langle Q';\sigma'\rangle$ admits an adequate derivation. Then all needed atoms in $\langle Q';\sigma'\rangle$ are descendants of needed atoms in $\langle Q;\sigma\rangle$. □

Definition 4.5 A selectability predicate $\mathcal{S}$ is said to be *serializable* if for every $Q \in \mathcal{Q}$ the following requirement is satisfied. Let $\xi = \langle Q;\epsilon\rangle \Rightarrow \langle Q';\sigma'\rangle$ be a derivation step with selected atom A, and let B be an atom in Q that is not selectable in $\langle Q;\epsilon\rangle$, but which descendant B' is selectable in $\langle Q';\sigma'\rangle$. Then for every derivation $\langle Q;\epsilon\rangle \Rightarrow^* \langle Q'';\sigma''\rangle$ such that the descendant of B is selectable in $\langle Q'';\sigma''\rangle$, we have that Q'' does not contain a descendant of A. □

Call by need is adequate. Now we have the following result.

Theorem 4.6 *If the (in)adequacy predicates $\mathcal{A}$ and $\mathcal{I}$ and the selectability predicate $\mathcal{S}$ are serializable, then for each $Q \in \mathcal{Q}$.*

1. *if a state $\langle Q;\sigma\rangle$ is not adequate then it contains a needed atom,*
2. *if there is an adequate derivation starting in $\langle Q;\sigma\rangle$ consisting of n derivation steps, then there is an adequate derivation starting in $\langle Q;\sigma\rangle$ in which only needed atoms are selected and which does not contain more than n steps.*

Proof. Let

$$\xi \equiv \langle Q;\sigma\rangle = \langle Q_0;\sigma_0\rangle \Rightarrow^* \langle Q_n;\sigma_n\rangle$$

be an adequate derivation of n steps starting in $\langle Q;\sigma\rangle$. We proceed by induction on n.

1. Suppose that ξ consists of one step: $\xi \equiv \langle Q_0;\sigma_0\rangle \Rightarrow \langle Q_1;\sigma_1\rangle$. By the first property, we have that the selected atom in ξ is needed.
2. Suppose that $\xi \equiv \langle Q_0;\sigma_0\rangle \Rightarrow \langle Q_1;\sigma_1\rangle \Rightarrow^* \langle Q_n;\sigma_n\rangle$ consists of more than one step. By the induction hypothesis, there is a call by need derivation ξ_1 issuing from $\langle Q_1;\sigma_1\rangle$ that is adequate. Let A_0 be the atom that is selected in the first step of ξ. Two cases are distinguished.
   (a) A_0 is needed. Then the derivation step $\xi_0 \equiv \langle Q_0;\sigma_0\rangle \Rightarrow \langle Q_1;\sigma_1\rangle$ followed by ξ_1 is an adequate call by need derivation.
   (b) A_0 is not needed. We construct a call by need derivation that is adequate, using serializability.

This result is quite abstract. In the following section we discuss a concrete application.

5 Using (In)adequacy Predicates

In this section we consider logic programs with guards and delay declarations. We extend the language with so-called *requests*, which permits to formalize a simple notion of adequacy which is useful practice. The first subsection contains preliminaries, in the second subsection the notion of request is introduced.

5.1 Guards and Delay Declarations

Guards. We roughly follow the approach described by Shapiro in [8]. A *guard* is a conjunction of atoms built from function symbols and *guard relation symbols*, which possibly don't coincide with the relation symbols used in the program. Guards are written as $G_1,\ldots,G_k$, which can be abbreviated to $\vec{G}$. A *guarded clause* is an expression of the form

$$H \leftarrow G_1,\ldots,G_k \mid B_1,\ldots,B_n.$$

We assume that it is decidable whether a ground instance of a guard holds. The operational semantics as given in Definition 3.2 changes as follows. Let A be the selected atom in some query and let τ be a most general unifier of the head of a clause $c = H \leftarrow \vec{G} \mid \vec{B}$ and A. If $\vec{G}\tau$ is ground and holds, then c can be applied to A. If $\vec{G}\tau$ is ground and does not hold, then the computation fails. If $\vec{G}\tau$ is not ground, then the computations suspends. As in the case without guards, with classical notions of success and failure, the computations also fails if A does not unify with the head of any clause of the program.

Delay Declarations. Logic programming languages which employ a dynamic selection rule need a mechanism for determining when an atom is selectable. Here we use delay declarations as introduced by Naish in [7]. We roughly follow the approach described by Apt and Luitjes in [1]. A *delay condition* is a conjunction built from delay base conditions of the following form:

- *ground*(s), which holds iff s is a ground term,
- *nonvar*(s), which holds iff s is a non-variable term

A difference with the delay declarations as in [1] is that we do not consider delay conditions that are disjunctions.

Now a *delay declaration* for a relation symbol p is an expression of the form

delay A until c

with A an atom with relation symbol p and c a delay condition. We suppose that a program is equipped with delay declarations. We admit the possibility that there is more than one delay declaration for a relation symbol; in this respect our approach differs from the one in [1].

Delay declarations influence the operational semantics as follows. An atom B is said to be *delay-safe* if for every delay declaration *delay A until c* and for every substitution σ we have that $A\sigma = B$ implies that $c\sigma$ holds. In this setting we make use of the following notion of selectability: $\mathcal{S}(A)$ iff A is delay-safe. Note that this selectability predicate is monotonic. Now in order for the first requirement of Definition 3.1 to hold, we assume the following relation between guards and delay declarations:
for every delay-safe atom A and every guarded clause $H \leftarrow \vec{G} \mid \vec{B}$ of $\mathcal{P}$ we have that if A and H have a most general unifier θ then $\vec{G}\theta$ is ground.

Natural Programs. A *moding* is a function assigning to every argument of a predicate symbol *In* or *Out*. In the first case an argument is said to be *input* and in the second case it is said to be *output*. In the remainder of this paper we assume every predicate symbol to have a unique moding. An argument of a predicate symbol occurring in a clause (query) is said to be *producing* either if it occurs in the head and is input or if it occurs in the body and is output. An argument of a predicate symbol is said to be *consuming* otherwise.

Definition 5.1

1. A clause is said to be *natural* if the family of terms in its producing positions is linear, i.e. no variable appears more than once. A program is *natural* if every clause is natural.

2. A delay declaration *delay A until c* is *natural* if every variable of c occurs in an input position of A, and moreover A has only fresh distinct variables in its output positions. □

In the remainder of this section we consider natural programs with guards and delay declarations as described in this subsection.

5.2 Requests

In the examples we have seen that some reasonable notions of (in)adequacy depend not only on the program and the state under consideration, but also on the the degree of instantiation of the variables of the initial state. This is for instance the case if we start computing the list of prime numbers with the aim to find only the first one (or the first few ones). Then we would like to start in $\langle prime(x); \epsilon\rangle$ and specify the adequacy predicate to hold on $\langle Q; \sigma\rangle$ and $\mathcal{P}$ if the first element of $x\sigma$ is specified. In this subsection we introduce a way to formalize such notions of (in)adequacy by means of requests.

A *request condition* or shortly a *request* is a conjunction built from the following request base conditions:

- $Val(x)$, which holds in a state $\langle Q; \sigma\rangle$ if $Var(x\sigma) \cap Var(\vec{A}) = \emptyset$,
- $Root(x)$, which holds in a state $\langle Q; \sigma\rangle$ if either $x\sigma$ is not a variable or $Var(x\sigma) \cap Var(\vec{A}) = \emptyset$,

Definition 5.2 A *request configuration* is an expression of the form $r : \langle Q; \sigma\rangle$ with r a request.

The definition of a derivation step is extended to request configurations as follows: if $\langle Q;\sigma\rangle \Rightarrow \langle Q';\sigma'\rangle$, then $r : \langle Q;\sigma\rangle \Rightarrow r : \langle Q';\sigma'\rangle$. A request configuration $r : \langle Q;\sigma\rangle$ is natural if $\langle Q;\sigma\rangle$ is natural. □

Other notions concerning derivations are extended in the natural way.

Natural request configurations are closed under resolution with natural clauses.

In this setting we will consider the following notion of adequacy (now defined on a request configuration and a program): $\mathcal{A}(r : \langle Q;\sigma\rangle)$ holds if and only if r holds in $\langle Q;\sigma\rangle$. We take for inadequacy the usual notion of failure.

Example 5.3 The following program implements the sieve of Eratosthenes.

$$\begin{array}{rcl}
primes(x) & \leftarrow & from(2,y), sieve(y,x). \\
from(x,y) & \leftarrow & from(x+1,z), y=[x|z]. \\
sieve([x|y],[x|z]) & \leftarrow & filter(x,y,u), sieve(u,z). \\
filter(x,[y|z],u) & \leftarrow & div(x,y) \mid filter(x,z,u). \\
filter(x,[y|z],u) & \leftarrow & nondiv(x,y) \mid filter(x,z,u'), u=[y|u'].
\end{array}$$

We have the following derivation:

$$\begin{array}{rl}
Val(x) : \langle primes([x|xs]); \epsilon\rangle & \Rightarrow \\
Val(x) : \langle from(2,y), sieve(y,[x|xs]); \epsilon\rangle & \Rightarrow \\
Val(x) : \langle from(3,z), y = [2|z], sieve(y,[x|xs]); \epsilon\rangle & \Rightarrow \\
Val(x) : \langle from(3,z), sieve([2|z],[x|xs]); \epsilon\rangle & \Rightarrow \\
Val(x) : \langle from(3,z), filter(2,z,u), sieve(u,xs); x \mapsto 2\rangle &
\end{array}$$

This derivation is adequate because in the last request configuration the request is satisfied. □

5.3 Call by need

The aim of the present subsection is to apply Theorem 4.6 in the practical setting of logic programs with guards and delay declarations, with the notion of request as defined above. So we need to show that the adequacy predicate using requests and the selectability predicate using delays are serializable. This can be shown provided we impose one further requirement on the programs we consider; this requirement is defined as follows.

Definition 5.4 We say that $\mathcal{P}$ is *non-destructive* if for every selectable atom A, for every clause $H \leftarrow \vec{G} \mid \vec{B}$ in $\mathcal{P}$ and for every most general unifier σ of H and A, we have that $A\sigma$ is in its input positions a variant of A. □

Thus, a non-destructive program is a program which never instantiates the input positions of the selected atoms. A large class of programs satisfies this requirement, as is argued in [3].

Assuming programs to be non-destructive, we can show that the selectability predicate and the (in)adequacy predicates of the present setting are serializable. Hence we have the following corollary of Theorem 4.6

Corollary 5.5 *Let $\mathcal{P}$ be a natural and non-destructive program. Every request configuration that is not adequate contains a needed atom, and if a request configuration admits an adequate derivation of length n then it admits an adequate call by need derivation of length not more than n.*

Finding needed atoms. We now discuss how to find needed atoms.

Definition 5.6 Let $r : \langle Q; \sigma\rangle$ be a request configuration.

1. If $r = Val(x)$, then we say that r *demands* an atom B in Q if a variable in $x\sigma$ occurs in a producing position of B.
2. If $r = Root(x)$ then we say that r *demands* an atom B in Q if $x\sigma$ is a variable occurring in a producing position of B.

3. If $r = r_1 \wedge r_2$, then we say that r *demands* an atom B in Q either if r_1 demands B or if r_2 demands B. □

We need one last concept.

Definition 5.7 Let A be an atom and $x \in Var(A)$. We say that A *is locked on* x if for every ground substitution σ not having x in its domain, we have that $A\sigma$ is not selectable. □

It can be proven that if A is not selectable then there is a variable x in its input positions such that A is locked on x. There might be more than one such variable.

Definition 5.8 (Demanded Atom) Let $r : \langle \vec{A}; \sigma \rangle$ be a request configuration. The set of *demanded atoms* is the minimal subset of $\vec{A}$ such that the following conditions are satisfied.

- if $r\sigma$ demands x and x occurs in an output position of A_i, then A_i is demanded,
- if A_i is demanded and it is locked on a variable x, and x occurs in an output position of A_j, then A_j is demanded. □

Now we have the following result.

Theorem 5.9 *In a natural and non-destructive program, demanded atoms are needed. Moreover, every state that is not adequate contains a demanded atom.*

Example 5.10 Consider the program quicksort using difference lists:

$$\begin{array}{rcl}
qsort(x, y) & \leftarrow & qsort\_dl(x, y, []). \\
qsort\_dl([x|x'], y, z) & \leftarrow & partition(x', x, ls, bs), \\
 & & qsort\_dl(ls, y, [x|y']),\ qsort\_dl(bs, y', z). \\
partition(x, [y|ys], [y|ls], bs) & \leftarrow & x \geq y \mid partition(x, ys, ls, bs). \\
partition(x, [y|ys], ls, [y|bs]) & \leftarrow & x < y \mid partition(x, ys, ls, bs). \\
partition(x, [\,], [\,], [\,]). & & \\
qsort\_dl([], x, x). & &
\end{array}$$

Let l be a ground list. Using the specification of the adequacy predicate as above, the request configuration

$Val(x) : \langle qsort(l, [x|x']); \epsilon \rangle$

is the starting point of an adequate derivation that ends in a request configuration in which the value of the first element of l is known. It is important to observe that in order to obtain the desired result the computation does *not* have to order the whole list, and it will not do so, provided that the selection rule is sufficiently "intelligent", which is the case if a call-by-need selection rule is employed; in this case it is worth noticing that the above request configuration generates a derivation in which only half of the atoms of the form $qsort\_dl(xs, ys, zs)$ which are introduced along the derivation will eventually be resolved. □

6 Concluding remarks

In this section we discuss some possible other uses of the parameterization over success and failure.

Interactive Computing

The computation with requests can be extended such that one is allowed to add requests on the fly. In this way we obtain in a very natural way a notion of interactive computing. The interaction then is between a *requesting* and a *resolving* agent. The requesting agent specifies a request of the form $r_0 : \langle Q; \epsilon\rangle$. Then it is the turn of the resolving agent, who will try to construct a derivation ending in a state that is adequate with respect to the request r_0. Having reached this point, it is again the turn of the requesting agent, who might decide either to be satisfied, or to add another request r_1. One can for instance think of a computation that is supposed to yield an infinite list; first one asks for the first value and having got that one, one may decide to ask in addition for the second one.

Expert Systems

As yet we did not fully exploit the possibility to specify an inadequacy predicate other than the usual notion of failure in logic programming. A domain where it is useful is in that of expert systems. Suppose we are in presence of a diagnosis system based on backward chaining. Then we can be interested in knowing whether the system, in order to provide a certain explanation δ, has been used a certain information (or inference rule) *inf* (or, equivalently, if the explanation δ would hold also if *inf* was not present). Generally, this can be done by removing *inf* from the knowledge base; in our framework this can be simply achieved by specifying together with the query an appropriate inadequacy predicate which triggers every time that some information in *inf* is being used. This has the advantage of not requiring internal modifications. Furthermore, it could allow arbitrarily sophisticate control over the inference mechanism, by forbidding certain "undesired" forms of reasoning.

Acknowledgments

We thank Krzysztof Apt for feed-back.
This research was supported by NWO/SION project number 612-33-003, entitled 'Parallel declarative programming: transforming logic programs to lazy functional programs'. The first author was working at the faculty of mathematics, computer science, physics, and astronomy of the University of Amsterdam when this research was initiated.

References

[1] K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In A. Borzyszkowski and S. Sokolowski, editors, *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, (AMAST'95)*, Lecture Notes in Computer Science, pages 1–19, Berlin, 1995. Springer-Verlag.

[2] K.R. Apt. *From Logic Programming to Prolog.* Prentice-Hall, 1997.

[3] 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)*, number 711 in Lecture Notes in Computer Science, pages 1–19, Gdańsk, Poland, 1993. Springer Verlag.

[4] J. Glauert and Z. Khasidashvili. Relative normalization in orthogonal Expression Reduction Systems. In N. Dershowitz and N. Lindenstrauss, editors, *Proceedings of the 4th International Workshop on Conditional and Typed Rewriting Systems (CTRS '94)*, number 968 in Lecture Notes in Computer Science, pages 144–165, Jerusalem, Isreal, 1994. Springer Verlag.

[5] G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems, I and II. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in honor of Alan Robinson*, pages 395–443. MIT Press, Cambridge, Massachusetts, 1991.

[6] A. Middeldorp. Call by need computations to root-stable form. In *Proceedings of the 24th Symposium on Principles of Programming Languages (POPL '97)*, pages 94–105, Paris, France, January 1997. ACM Press.

[7] L. Naish. An introduction to mu-prolog. Technical Report 82/2, The University of Melbourne, 1982.

[8] E. Y. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.