

Logic Programming with Requests

Sandro Etalle

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

Femke van Raamsdonk

Department of Theoretical Computer Science, Vrije Universiteit,
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
femke@cs.vu.nl
CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands.

Abstract

We propose an extension of logic programming where the user can specify, together with the initial query, the information he is interested in by means of a request. This allows one to extract a result from an incomplete computation, such as the prefix of an infinite derivation. The classical property of independence of the selection rule doesn't hold anymore. It is shown that under mild conditions a class of selection rules can be identified for which independence holds. A model-theoretic semantics for the language is given.

1 Introduction

The purpose of this paper is to present an extension of logic programming where it is possible to express a form of partial result. This is done by considering instead of the traditional successful derivations, where all subgoals are resolved, so-called adequate derivations, where possibly some subgoals remain. The intuition is that an adequate derivation is similar to a derivation in a lazy functional language, like for instance Haskell, where an expression is evaluated to a head-normal form. Adequacy is defined in terms of two new features that we add to logic programming, namely requests and strictness annotations.

Requests allow the user to specify the information (partial result) he is interested in. The definition is inspired by the definition of head normal form in functional programming: it is possible to specify that one wishes to compute the head-symbol of the value of a certain variable, or the complete value of a specific variable. Consider for instance the program

$$from(x, y) \leftarrow x' \text{ is } x + 1, from(x', z), y = [x|z].$$

that can be used to generate the list of natural numbers as follows:

$$from(0, y_0) \Rightarrow from(1, y_1), y_0 = [0|y_1]$$

$$\begin{aligned}
&\Rightarrow \text{from}(2, y_2), y_1 = [1|y_2], y_0 = [0|y_1] \\
&\Rightarrow \text{from}(2, y_2), y_0 = [0, 1|y_2] \\
&\Rightarrow \dots
\end{aligned}$$

If the partial result is specified to be the value of the second natural number,

$$\begin{aligned}
\text{from}(0, [x_0, x_1|z]) &\Rightarrow \text{from}(1, y_1), [x_0, x_1|z] = [0|y_1] \\
&\Rightarrow \text{from}(2, y_2), y_1 = [1|y_2], [x_0, x_1|z] = [0|y_1] \\
&\Rightarrow \text{from}(2, y_2), [x_0, x_1|z] = [0, 1|y_2]
\end{aligned}$$

is a computation that is sufficient. We say that the last query satisfies the request expressing that we wish to compute the value of the variable x_1 .

A *strictness annotation*, the second feature we add to logic programming, can be used by the programmer to specify which atoms can be left unsolved. As illustrated in the previous example, a query that satisfies the request specified by the user is not necessarily the empty one. Atoms that are annotated as *strict* need to be resolved. Intuitively, these atoms stand for tests that need to be performed in order to select the right branch of an SLD-tree. Atoms that are annotated as *lazy* may remain unresolved. Intuitively, these atoms stand for a calculation that may or may not be performed.

So, we take here a more concrete approach than in [7], where adequacy is a parameter with the intended use to extract a result from a partial computation. Another difference is that in [7] failure is replaced by a predicate for inadequacy, with the intended use to constrain the search space, whereas in the present paper failure is failure in the traditional sense.

Now the dynamics of our language is as follows. The computation proceeds until a point where the request specified by the user is satisfied, and where no strict atoms remain. If the computation fails, then the backtracking mechanism is used to explore alternatives, if any (in a Prolog-like implementation). In case deadlock is reached, the computation stops without providing any answer. No expressive power of the traditional approach is lost: successful derivations correspond to adequate derivations with the trivial request and all predicates declared to be strict.

A well-known result in the classical approach to logic programming is independence of the selection rule. This result doesn't hold anymore in the present setting, because not all atoms need to be selected. An obviously important question is then which selection rule should be applied. The idea is that only atoms that contribute to the desired result should be selected. We prove that under suitable conditions a class of selection rules can be identified that are correct in the sense that they yield adequate derivations whenever possible, and that are optimal in the sense that they yield adequate derivations with a minimal number of steps.

Further, we present a declarative semantics of logic programming with requests. Its definition makes use of a recombination of ideas of the s -semantics as defined in [6] and the Ω -semantics as defined in [5] (see also [6]). It is shown that under mild (necessary) restrictions the operational semantics is sound and complete with respect to the declarative one.

2 Programming with Requests

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 [17]. We roughly follow the approach described by Apt and Luitjes in [4]. A *delay condition* is a conjunction built from delay base conditions of the form:

- $nonvar(s)$, which holds if s is a non-variable term.
- $ground(s)$, which holds if s is a ground term,

A difference with [4] is that we do not consider delay conditions that are disjunctions; this restriction is necessary for the main result (Theorem 3.5) to hold. A *delay declaration for a predicate symbol p* is defined as

delay A until c

with A an atom with predicate symbol p and c a delay condition. We admit the possibility that there is more than one delay declaration for a relation symbol; this is a second difference with the approach of [4].

Derivations. A *program* is specified by a set of clauses and a set of delay declarations. An atom B is *selectable* in a program \mathcal{P} if for every delay declaration *delay A until c* in \mathcal{P} we have the following: if $B = A\sigma$ for some substitution σ , then $c\sigma$ holds. Note that an instance of a selectable atom is selectable. A *selection rule* is a mapping that given a state $\langle Q; \sigma \rangle$ returns a selectable atom in Q whenever Q contains a selectable atom, and returns a fresh symbol δ otherwise. A state not containing a selectable atom is said to be *in deadlock*. A difference with the standard approach is that instead of queries we consider *states*, which are pairs of the form $\langle Q; \sigma \rangle$ consisting of a query and a substitution. For a selection rule \mathcal{S} , we have an \mathcal{S} -derivation step $\langle Q; \sigma \rangle \xrightarrow[\mathcal{C}]{\tau} \langle Q'; \sigma\tau \rangle$ if the clause C can be applied to the atom selected by \mathcal{S} using most general unifier τ . We omit specifying the selection rule, the clause, the most general unifier whenever possible. Further, we usually specify only the relevant part of a substitution in a state. A (finite or infinite) sequence of \mathcal{S} -derivation steps is called a *\mathcal{S} -derivation*. We write \Rightarrow^+ for the transitive closure of \Rightarrow and \Rightarrow^* for its reflexive-transitive closure.

Requests. The key feature of the language we propose is that the user can specify which information he is interested in, and that a computation stops if it is observable that this information has been found. Expressing the desired information is done by means of so-called *requests*. We consider as an example the program *sieve* that implements the sieve of Eratosthenes:

$primes(x)$	\leftarrow	$from(2, y), sieve(y, x).$
$from(x, [x z])$	\leftarrow	$x' \text{ is } x + 1, from(x', z).$
$sieve([x y], [x z])$	\leftarrow	$filter(x, y, w), sieve(w, z).$
$filter(x, [y z], w)$	\leftarrow	$div(x, y), filter(x, z, w).$
$filter(x, [y z], [y w])$	\leftarrow	$nondiv(x, y), filter(x, z, w).$

We omit the clauses for the predicates *div* and *nondiv*. The following delay declarations are assumed:

$$\begin{aligned} & \text{delay filter}(x, y, w) \text{ until nonvar}(y). \\ & \text{delay filter}(x, [y|z], w) \text{ until ground}(x) \wedge \text{ground}(y). \\ & \text{delay sieve}(x, y) \text{ until nonvar}(x). \end{aligned}$$

We consider two examples where the information we are interested in is provided by a derivation that is not successful in the traditional sense of the word. First, suppose that we wish to find out whether the list of prime numbers is empty or not. The following derivation shows that the list of prime numbers is not empty, but of the form $[2|x']$ for some list x' :

$$\begin{aligned} \langle \text{primes}(x); \epsilon \rangle & \Rightarrow \langle \text{from}(2, y), \text{sieve}(y, x); \epsilon \rangle \\ & \Rightarrow \langle \text{from}(3, z), \text{sieve}([2|z], x); \epsilon \rangle \\ & \Rightarrow \langle \text{from}(3, z), \text{filter}(2, z, u), \text{sieve}(u, x'); x \mapsto [2|x'] \rangle. \end{aligned}$$

Second, suppose that our purpose is to compute the value of the second prime number. This is done in the following computation, which shows that value of the second prime number is 3:

$$\begin{aligned} \langle \text{primes}([x_1, x_2|l]); \epsilon \rangle & \Rightarrow \langle \text{from}(2, y), \text{sieve}(y, [x_1, x_2|l]); \epsilon \rangle \\ & \Rightarrow \langle \text{from}(3, z), y = [2|z], \text{sieve}(y, [x_1, x_2|l]); \epsilon \rangle \\ & \Rightarrow^* \langle \text{from}(3, z), \text{filter}(2, z, u), \text{sieve}(u, l); x_1 \mapsto 2 \rangle \\ & \Rightarrow^* \langle \text{from}(4, z'), \text{filter}(2, [3|z'], u), \text{sieve}(u, [x_2|l]); x_1 \mapsto 2 \rangle \\ & \Rightarrow^* \langle \dots, \text{filter}(2, z', u'), \text{sieve}([3|u'], [x_2|l]); \dots \rangle \\ & \Rightarrow \langle \dots, \text{filter}(3, u', w), \text{sieve}(w, l); \dots, x_2 \mapsto 3 \rangle. \end{aligned}$$

In the first example, the derivation reveals as information the constructor at the top of the data-structure that is computed. This is like a head-normal form in functional programming. The second example is concerned with a variation on this theme: here the aim of the computation is to calculate the value of one of the variables that occur in the initial state. Both forms of partial information can be expressed by means of requests. Requests are built from two basic expressions that capture the two typical cases of the examples above: *Root*(x), expressing that we aim at finding the constructor at the top of the value of x , and *Val*(x), expressing that we aim at finding the value of x . The formal definition of requests is as follows.

Definition 2.1 A *request condition* or shortly a *request* is a conjunction (denoted using \wedge) built from the following request base conditions:

- *Root*(x), which holds in a state $\langle Q; \sigma \rangle$ if $x\sigma \notin \text{Var}(Q)$,
- *Val*(x), which holds in a state $\langle Q; \sigma \rangle$ if $\text{Var}(x\sigma) \cap \text{Var}(Q) = \emptyset$.

The empty conjunction is written as *True* and holds in any state $\langle Q; \sigma \rangle$. \square

If a request r holds in a state $\langle Q; \sigma \rangle$, then we also say that the state $\langle Q; \sigma \rangle$ *satisfies* the request r . Going back to the two derivations in the program *sieve* above, we have that the request *Root*(x) holds in the state

$$\langle \text{from}(3, z), \text{filter}(2, z, u), \text{sieve}(u, x'); x \mapsto [2|x'] \rangle,$$

and we have that the request $Val(x_2)$ holds in the state

$$\langle \dots, filter(3, u', w), sieve(w, l); x_1 \mapsto 2, x_2 \mapsto 3 \rangle.$$

Requests seem similar to delay declarations. There are however important differences. First, it depends on the substitution of a state whether a request holds or not. Further, the difference between $Root(x)$ and $nonvar(x)$ is that $Root(x)$ not only holds in $\langle Q; \sigma \rangle$ if $x\sigma$ is a non-variable term, but also if $x\sigma$ is a variable that will never be instantiated further; then it serves as a constant. There is a similar difference between $Val(x)$ and $ground(x)$.

One can easily imagine variations and extensions of the definition of requests as given above. Our choice for requests built using $Val(x)$ and $Root(x)$ is motivated by the correspondence with head-normal forms in functional programming and is, we feel, a natural one.

Note that if Q is ground, then we have that $\langle Q; \sigma \rangle$ satisfies r , for every substitution σ and for every request r . In particular, we have that $\langle \square; \sigma \rangle$ satisfies r for every σ and r . So in a successful state any request is satisfied.

Strict and Lazy Predicates. The request conditions provide the user with a form of control in the sense that a computation can, as far as the user is concerned, stop once a state is reached where the specified request is satisfied. The language provides the programmer with an additional form of control: he can specify which atoms have to be evaluated and which ones may remain unselected. This is done by means of a partitioning of the predicates into *strict* and *lazy* ones. An atom is then strict (lazy) if its predicate is. The intuition is that strict atoms are meant to test and that a computation should not stop at a point where a test is not performed yet. Lazy atoms are meant to compute, and no harm is done if a derivation is stopped at a point where a(n unnecessary) part of the computation remains to be done.

Consider for example the program defined by the clause $p(x, a) \leftarrow x = 0..$ In one step $\langle p(1, y); \epsilon \rangle \Rightarrow \langle 1 = 0; y \mapsto a \rangle$ a state is reached in which the request $Val(y)$ holds. However, this state is intuitively speaking not safe since a next step would yield failure (as we will see below in Definition 2.3, our language inherits from logic programming the following notion of failure: a derivation fails if it ends in a state with a query that is not empty and no clause can be applied to the selected atom). Now the derivation does not stop in $\langle 1 = 0; y \mapsto a \rangle$ if the programmer has specified the predicate $=$ as strict; in that case every atom of the form $s = t$ needs to be selected eventually. The intuition is that $=$ is used as a test.

Dynamics. The operational semantics of our language makes use of the notion of request and of the partitioning of the predicates into strict and lazy ones. It is a refinement of the usual operational semantics of logic programming. The expressions that are transformed in a derivation are request conditions, consisting of a request and a state.

Definition 2.2

- A *request configuration* is an expression $r : \langle Q; \sigma \rangle$ with r a request and $\langle Q; \sigma \rangle$ a state.
- The definition of a derivation step is extended to request configurations as follows: if $\langle Q; \sigma \rangle \Rightarrow \langle Q'; \sigma' \rangle$, then $r : \langle Q; \sigma \rangle \Rightarrow r : \langle Q'; \sigma' \rangle$. \square

The following definition is crucial.

Definition 2.3 Let \mathcal{P} be a program. A request configuration $r : \langle Q; \sigma \rangle$ is

1. *adequate* if $\langle Q; \sigma \rangle$ satisfies r and Q doesn't contain strict atoms,
2. *failing* if $Q \neq \square$ and no clause from \mathcal{P} can be applied to the selected atom in $\langle Q; \sigma \rangle$,
3. *deadlocked* if $Q \neq \square$ and $\langle Q; \sigma \rangle$ contains no selectable atom. \square

In a derivation step, we assume that a suitable renaming of the clause is used, that is, one that has no variables in common with the request configuration. These notions carry over to derivations in the natural way: a finite derivation is adequate (failing, deadlocked) if its last request configuration is. Note that the notion of failing depends on the selection rule, which we let unspecified.

The implementation we imagine of our language is similar to the one of Prolog: if an adequate request configuration is reached, the computation stops and the desired information is communicated to the user. If a failing request configuration is reached, the computation stops and starts backtracking whenever this is possible.

We define the equivalence of two substitutions with respect to a request.

Definition 2.4 We define when two substitutions σ and τ are *equivalent with respect to a request* r , notation $\sigma \sim_r \tau$, by induction on the structure of r as follows:

1. $\sigma \sim_{True} \tau$ for every σ and τ ,
2. $\sigma \sim_{Val(x)} \tau$ if $x\sigma$ is a renaming of $x\tau$,
3. $\sigma \sim_{Root(x)} \tau$ if $x\sigma$ and $x\tau$ have the same outermost function symbol or if $x\sigma$ and $x\tau$ are both variables,
4. $\sigma \sim_{r \wedge r'} \tau$ if $\sigma \sim_r \tau$ and $\sigma \sim_{r'} \tau$. \square

For instance the substitutions $\{x \mapsto a, y \mapsto b\}$ and $\{x \mapsto a, y \mapsto c\}$ are equivalent with respect to the request $Val(x)$. It can be shown that if a computation has reached a state which satisfies a request r , then continuing the computation only yields states that satisfy r and that are equivalent to each other with respect to r .

Example. We consider the program for *quicksort* with accumulator:

$$\begin{array}{ll}
q(xs, ys) & \leftarrow qa(xs, ys, []). \\
qa([x|xs], ys, zs) & \leftarrow p(x, xs, ls, bs), qa(bs, ws, zs), qa(ls, ys, [x|ws]). \\
qa([], xs, xs) & \leftarrow . \\
p(x, [y|xs], [y|ls], bs) & \leftarrow x > y, p(x, xs, ls, bs). \\
p(x, [y|xs], ls, [y|bs]) & \leftarrow x \leq y, p(x, xs, ls, bs). \\
p(x, [], [], []) & \leftarrow .
\end{array}$$

Let the only strict predicates be $>$ and \leq . Suppose we are interested in knowing the smallest element of the list $[2, 1, 3]$. In order to compute it, we can use the request configuration $Val(y) : \langle qa([2, 1, 3], [y|ys], []); \epsilon \rangle$. The computation proceeds as follows. In each step, the selected atom is written in **boldface**. Further, we omit the request, and write only a part of the substitution. The underlining of variables is to illustrate the definition of demanded atom in Section 3.

$$\begin{array}{l}
\langle \mathbf{qa}([2, 1, 3], [y|ys], []); \epsilon \rangle \\
\Rightarrow \langle \mathbf{p}(2, [1, 3], \underline{ls}, \mathbf{bs}), qa(bs, ws, []), qa(ls, [y|ys], [2|ws]); \epsilon \rangle \\
\Rightarrow \langle p(2, [3], \underline{ls}', \mathbf{bs}), qa(bs, ws, []), \mathbf{qa}([1|\underline{ls}'], [y|ys], [2|ws]); ls \mapsto [1|ls'] \rangle \\
\Rightarrow \langle \mathbf{p}(2, [3], \underline{ls}', \mathbf{bs}), qa(bs, ws, []), p(1, \underline{ls}', \underline{ls}'', \mathbf{bs}''), \\
qa(\underline{bs}'', \underline{ws}'', [2|ws]), qa(\underline{ls}'', [y|ys], [1|\underline{ws}'']); \epsilon \rangle \\
\Rightarrow \langle \mathbf{p}(2, [], \underline{ls}', \mathbf{bs}'), qa([3|\underline{bs}'], \underline{ws}, []), p(1, \underline{ls}', \underline{ls}'', \mathbf{bs}''), \\
qa(\underline{bs}'', \underline{ws}'', [2|ws]), qa(\underline{ls}'', [y|ys], [1|\underline{ws}'']); bs \mapsto [3|\underline{bs}'] \rangle \\
\Rightarrow \langle qa([3], \underline{ws}, []), \mathbf{p}(1, [], \underline{ls}'', \mathbf{bs}''), \\
qa(\underline{bs}'', \underline{ws}'', [2|ws]), qa(\underline{ls}'', [y|ys], [1|\underline{ws}'']); ls' \mapsto [], bs' \mapsto [] \rangle \\
\Rightarrow \langle qa([3], \underline{ws}, []), qa([], \underline{ws}'', [2|ws]), \mathbf{qa}([], [y|ys], [1|\underline{ws}'']); ls'' \mapsto [], bs'' \mapsto [] \rangle \\
\Rightarrow \langle qa([3], \underline{ws}, []), qa([], \underline{ws}'', [2|ws]); y \mapsto 1, ys \mapsto \underline{ws}'' \rangle
\end{array}$$

This derivation is adequate. It satisfies the initial request and has no strict predicates which still need to be resolved. Notice that in order to obtain the desired answer it was not necessary to order the whole list, as it would be the case with a classical SLD-derivation ending in success.

3 An Optimal Strategy

In this section we first explain why in the present framework independence of the selection rule does not hold. Then we show that under reasonable conditions it is possible to define selection rules that are *correct* in the sense that they yield an adequate state whenever possible, and that are *optimal* in the sense that they do so in the least possible number of derivation steps.

Independence. A well-known result for the classical approach to logic programming is the independence of the selection rule ([2, Theorem 3.33]). In the present setting this result does not hold anymore. Consider for instance the program consisting of the rules $p \leftarrow .$ and $q \leftarrow q..$ with p strict and q lazy. Starting in $True : \langle p, q; \epsilon \rangle$ we find, using the leftmost selection rule, the adequate derivation $True : \langle p, q; \epsilon \rangle \Rightarrow True : \langle q; \epsilon \rangle$. There is however

no adequate derivation using the rightmost selection rule. This is not really surprising: Whereas in order to reach the empty query all atoms have to be selected eventually, it is not necessarily the case that all atoms have to be selected in order to reach an adequate request configuration.

Our Approach. The question arises which selection rule should be used in order to find an adequate derivation, or more precisely, an adequate SLD-tree, whenever possible. Possible answers are to use a fair selection rule, then every atom has to be selected eventually, or to restrict attention to programs and queries for which it doesn't matter which selection rule is used. The first approach is a quite inefficient one, and the second one is completely unsuitable to deal with potentially infinite data-structures and computations, since it means in particular that non-termination is excluded.

What we do instead is the following. We define for each request configuration the set of *demanded atoms*. If the programs and queries we consider are restricted to the ones that have, intuitively speaking, a functional and sequential character, then it is possible to show that in order to reach an adequate request configuration (if there is one), it is necessary and sufficient to select all demanded atoms. As a consequence, it can be shown that for the class of *demand-driven selection rules*, that are defined as the selection rules selecting demanded atoms only the following form of independence holds: if a request configuration admits an adequate derivation then it admits an adequate derivation via any demand-driven selection rule, yielding a computed answer substitution that is equivalent as far as the request under consideration is concerned.

We first define demanded atoms and the demand-driven selection rules. Then we formulate restrictions on the programs and queries that we consider.

Demanded Atoms. The aim is now to identify a set of atoms, called the demanded atoms, such that every atom in the set has to be selected eventually in order to reach an adequate request configuration whenever possible. For the definition we need to consider *moded* programs and queries, where modes are used to indicate whether an argument of a predicate is used as input or as output. A mode of a predicate symbol p is defined as a function that assigns to every argument of p either *in* or *out*. In the first case the argument is said to be *input* and in the second case it is said to be *output*. Reasonable modes for the predicates used in the programs *sieve* and *quicksort* are as follows:

$primes$:	out	q :	$in \times out$
$from$:	$in \times out$	qa :	$in \times out \times in$
$sieve$:	$in \times out$	p :	$in \times in \times out \times out$
$filter$:	$in \times in \times out$	$>$:	$in \times in$
div :	$in \times in$	\leq :	$in \times in$
$nondiv$:	$in \times in$		

Henceforth it is assumed that every predicate has a unique mode; in case of multiple modes the predicates are renamed.

Now demanded atoms come in three sorts. First, it is clear that every strict atom needs to be selected eventually, since a request configuration in which strict atoms are present cannot be adequate. Second, some atoms need to be selected in order to satisfy the request under consideration. For instance, in order to satisfy a request of the form $Val(x)$, an atom that produces a value of x , that is, an atom having an occurrence of x in its output position, needs to be selected. Third, it might be the case that an atom that needs to be selected in order to reach an adequate request configuration cannot be selected yet because it is not sufficiently instantiated, and as a consequence does not satisfy its delay declarations. In that case there is at least one variable that needs to be bound to a value. Then an atom having this variable in an output position needs to be selected. This intuition is now formalized in the following definition.

Definition 3.1

- Let $r : \langle Q; \sigma \rangle$ be a request configuration. We define when r *demands* a variable of Q by induction on the definition of request as follows:
 1. *True* does not demand any variable,
 2. $Val(x)$ demands all variables in $x\sigma \cap Var(Q)$,
 3. $Root(x)$ demands $x\sigma$ if it is a variable of Q , and does not demand any variable otherwise,
 4. $(r \wedge r')$ demands all variables demanded by r and all variables demanded by r' .
- An atom A in Q *demands* a variable x if for every substitution σ such that $A\sigma$ is selectable, $x\sigma$ is a non-variable term.
- The set of *demanded atoms of level n* of a request configuration $r : \langle Q; \sigma \rangle$, denoted by $\mathcal{D}_n(r : \langle Q; \sigma \rangle)$, is defined as follows:
 1. if A is strict, then $A \in \mathcal{D}_0(r : \langle Q; \sigma \rangle)$,
 2. if there is a variable demanded by r that occurs in an output position of A , then $A \in \mathcal{D}_0(r : \langle Q; \sigma \rangle)$,
 3. if there is a variable demanded by an atom in $\mathcal{D}_{n-1}(r : \langle Q; \sigma \rangle)$ that occurs in an output position of A , then $A \in \mathcal{D}_n(r : \langle Q; \sigma \rangle)$.
- The set of *demanded atoms* of a request configuration $r : \langle Q; \sigma \rangle$, denoted by $\mathcal{D}(r : \langle Q; \sigma \rangle)$, is defined as $\bigcup_0^\infty \mathcal{D}_n(r : \langle Q; \sigma \rangle)$. □

We have that the variable x is demanded in the request configuration $Val(x) : \langle primes([x|l]); \epsilon \rangle$. Hence the atom $primes([x|l])$ is demanded. Performing a derivation step yields $Val(x) : \langle from(2, y), sieve(y, [x|l]); \epsilon \rangle$. Here x is still

demanded by the request $Val(x)$, so the atom $sieve(y, [x|l])$ is demanded. Moreover, the atom $sieve(y, [x|l])$ demands the variable y , and therefore the atom $from(2, y)$ is demanded as well. In the example *quicksort*, the demanded variables in the derivation are underlined.

The notion of demanded atom is used to define the demand-driven selection rules as follows.

Definition 3.2 A *demand-driven selection rule* is a mapping that given a request configuration $r : \langle Q; \sigma \rangle$ that is not adequate returns an atom in Q that is selectable and demanded, and returns δ otherwise. \square

In fact the terminology is a bit sloppy here, since a demand-driven selection may return δ in a request configuration that contains selectable atoms (but no atoms that are both selectable and demanded), and as a consequence it is strictly speaking not a selection rule.

Main Result. The aim is now to show that independence of the selection rule holds for the class of demand-driven selection rules. To that end we need to impose some restriction on the programs and queries that are considered.

The first restriction, *ws-modeness*, concerns the data-flow in a program. It is equivalent to the combination of well-modeness and simply modeness as defined in [3].

Definition 3.3 A clause $p_0(\vec{t}_0, \vec{s}_{n+1}) \leftarrow p_1(\vec{s}_1, \vec{t}_1), \dots, p_n(\vec{s}_n, \vec{t}_n)$ is said to be *ws-moded* if

- $Var(\vec{s}_i) \subseteq \bigcup_{j=1}^{i-1} Var(\vec{t}_j)$,
- $\vec{t}_1, \dots, \vec{t}_n$ are distinct variables not occurring in t_0 . \square

A query Q is said to be *ws-moded* if the clause $q \leftarrow Q$ with q a dummy symbol is *ws-moded*. These definitions carry over to programs, states and request configurations in the usual way. The set of *ws-moded* queries is closed under computation with *ws-moded* clauses. Further, well-moded programs (queries) can be transformed into *ws-moded* ones; this transformation does not necessarily preserve termination.

The second restriction, *mode-drivenness*, concerns the interaction between data-flow and delay-declarations. It states that the data-flow taking place in the input position of an atom must proceed from the selected atom to the resolving clause, and not vice-versa, and that the selectability of an atom should depend uniquely on how his input arguments are instantiated.

Definition 3.4 A program \mathcal{P} is said to be *mode-driven* if the following two requirements hold:

1. for every delay declaration *delay A until c* of \mathcal{P} we have that all variables of c occur in input positions of A ,

2. if B is a selectable atom and H is the head of a clause of \mathcal{P} that can be applied to B , then B restricted to its input positions is an instance of H restricted to its input positions. \square

Observe that the program *sieve* is mode-driven. In the language moded GHC [21], using guarded Horn clauses, an atom is selectable if requirements similar to the ones of mode-drivenness hold.

These restrictions imply some properties that are used to prove the main result. If in a request configuration the request is not satisfied, then there is a demanded atom. Moreover, if a request configuration is not adequate but admits an adequate derivation, then there is an atom that is demanded and selectable. This means that a demand-driven selection rule doesn't deadlock prematurely. Further, every demanded atom has to be selected eventually in order to obtain an adequate derivation.

Theorem 3.5 *Let \mathcal{P} be a ws-moded and mode-driven program and let \mathcal{S} be a demand-driven selection rule. Let $r : \langle Q; \sigma \rangle$ be a ws-moded request configuration. If there is an adequate derivation*

$$r : \langle Q; \sigma \rangle \Rightarrow^* r : \langle Q'; \sigma' \rangle$$

consisting of n derivation steps, then there is an adequate \mathcal{S} -derivation

$$r : \langle Q; \sigma \rangle \Rightarrow^* r : \langle Q''; \sigma'' \rangle$$

consisting of not more than n derivation steps. Moreover, $\sigma' \sim_r \sigma''$. \square

4 Declarative Semantics

A declarative semantics for logic programming with requests is defined. The definition is a recombination of ideas present in the s-semantics defined in [6] and the Ω -semantics defined in [5], and also reported in [6]. The aim of the s-semantics approach is to model the observable behaviour of logic programming. The Ω -semantics is defined to provide a compositional semantics for definite logic programs. We will use notation and terminology as in [6].

The Immediate Consequence Operator. Let \mathcal{P} be a program. The set of atoms of \mathcal{P} modulo variable renaming is denoted by \mathcal{B} . A subset of \mathcal{B} is called an *interpretation* (in [6] this is called a π -interpretation). The application of the immediate consequence operator, or bottom-up operator, denoted by $T_{\mathcal{P}}^l$, to an interpretation I is defined as follows:

$$T_{\mathcal{P}}^l(I) = I \cup \left\{ A \in \mathcal{B} \mid \begin{array}{l} \exists A' \leftarrow B_1, \dots, B_n \in \mathcal{P}, \\ \exists B'_1, \dots, B'_n \text{ variants of atoms in } I, \\ \exists \theta = mgu((B_1, \dots, B_n), (B'_1, \dots, B'_n)), \\ A = A'\theta \end{array} \right\}$$

The variants B'_1, \dots, B'_n are supposed to be renamed apart. This definition differs from the usual one in two respects. First, it deals with non-ground

interpretations as is also the case in the s-semantics. Second, it adds I to its output, which is needed to properly deal with lazy atoms. We define

$$\begin{aligned} T_{\mathcal{P}}^l \uparrow 0(I) &= I, \\ T_{\mathcal{P}}^l \uparrow i + 1(I) &= T_{\mathcal{P}}^l(T_{\mathcal{P}}^l \uparrow i(I)), \\ T_{\mathcal{P}}^l \uparrow \omega(I) &= \bigcup_{i \geq 0} T_{\mathcal{P}}^l \uparrow i(I). \end{aligned}$$

By well-known results, $T_{\mathcal{P}}^l \uparrow \omega(I)$ is the least fixpoint of $T_{\mathcal{P}}^l$ containing I .

The Model. The *lazy base* of a program \mathcal{P} , denoted by $LB_{\mathcal{P}}$, is defined as the set that contains for every lazy predicate p an atom of the form $p(x_1, \dots, x_n)$ with x_1, \dots, x_n different variables. So $LB_{\mathcal{P}}$ contains exactly one representative for each lazy predicate in the language of \mathcal{P} . We write LB instead of $LB_{\mathcal{P}}$ if \mathcal{P} is clear from the context.

Definition 4.1 The *lazy model* of a program \mathcal{P} , denoted by $\mathcal{L}(\mathcal{P})$, is defined as $T_{\mathcal{P}}^l \uparrow \omega(LB)$. \square

The lazy model of a program is related to the π -model as defined in [6] and the s -model as defined in [8] as follows. First, $\mathcal{L}(\mathcal{P})$ is a π -model of \mathcal{P} . Second, $Ground(\mathcal{L}(\mathcal{P}))$ is the least Herbrand model containing $Ground(LB_{\mathcal{P}})$, and finally, $\mathcal{L}(\mathcal{P})$ is the least s -model (with respect to \subseteq) of \mathcal{P} containing $LB_{\mathcal{P}}$.

Soundness and Completeness. In order to show soundness and completeness of the operational semantics with respect to the declarative one we need to impose two restrictions on the programs. The first is that for every clause of the program we have that all variables occurring in the head occur in the body as well. These programs are said to be *allowed* (see [14]). The second restriction is that there are no delay declarations. Under these restrictions we have the following result.

Theorem 4.2 *Let \mathcal{P} be an allowed program without delay declarations. Let θ be a substitution and let $r : \langle A; \epsilon \rangle$ be a request configuration. Suppose that $Var(r) \subseteq Var(A)$. Then the following two statements are equivalent:*

1. *There exist a derivation $r : \langle A; \epsilon \rangle \Rightarrow^* r : \langle B_1, \dots, B_m; \sigma \rangle$ such that*

- *$r : \langle B_1, \dots, B_m; \sigma \rangle$ is adequate,*
- *$A\sigma = A\theta$ (modulo variable renaming).*

2. *There exist $A' \in \mathcal{L}(\mathcal{P})$, and a substitution $\sigma' = mgu(A, A')$ such that*

- *$\langle A\sigma'; \sigma' \rangle$ satisfies r ,*
- *$A\sigma' = A\theta$ (modulo variable renaming).* \square

The restriction to allowed programs is necessary for the implication 1 implies 2 to hold. Consider for instance the program defined by the clause $p(x) \leftarrow q..$ Its lazy base is the empty set and its lazy model is $\{p(x)\}$. We have an adequate derivation $Val(x) : \langle p(x); \epsilon \rangle \Rightarrow Val(x) : \langle q; \epsilon \rangle$, but the request $Val(x)$ is not satisfied in the state $\langle p(x); \epsilon \rangle$. With the definition of the immediate consequence operator as considered here, the restriction to programs without delay declarations is for the implication 2 implies 1 to hold. Consider the same program as above, now extended with the delay declaration *delay* $p(x)$ *until* $ground(x)$. In the state $\langle p(x); \epsilon \rangle$ the empty request is satisfied, but there is no adequate derivation starting in the request configuration $\langle p(x); \epsilon \rangle$.

5 Concluding Remarks

In [7], we studied a generalization of logic programming where success and failure are replaced by predicates for adequacy and inadequacy. In that general setting we defined, inspired by [12], needed atoms as atoms that have to be selected eventually in order to reach an adequate state, and we showed that under certain conditions adequate SLD-trees are obtained by repeatedly selecting needed atoms. The setting of the present paper is much more concrete. In particular, whereas needed atoms cannot be computed, demanded atoms as considered here can be effectively found. We remark that demanded atoms are needed in the sense of [7].

An important underlying idea in the framework we propose is that the more specific an answer substitution is, the more is the information it carries. This is similar to what happens in concurrent constraint languages such as in Oz [20] and ccp [19], where information can be added by further instantiating logical variables. Yet, this is in contrast with the classical logic programming approach. In particular it is in contrast with the S-semantics [6] of a program, where the empty answer substitution is considered the most general one, hence the one holding the most information.

A difference with the approach as in ccp and Oz is that there a thread (agent) carries its computation out as long as it has a sufficient input for doing so, even if the output is not needed by any other agent. So the computation mechanism of ccp and Oz can be called *input driven*. The computational mechanism of the language proposed in the present paper is *output driven*, and as such more related to the call-by-need evaluation mechanism of lazy functional programming languages as Haskell. The analogy with such functional programming languages stops here; our proposal is a strict extension of logic programming, enjoying the presence of logical variables, unification and nondeterminism (implemented via backtracking).

The papers [17] and [15] have in common that they propose a system in which the selection rule is dynamic and guided by a mechanism similar to the delay declarations we use here as well (those constructs were actually

introduced by Naish in [17]). In both cases the term ‘lazy evaluation’ is used to indicate that the selection rule is dynamic; the computational mechanism is the standard one of logic programming, so there is no generalization of the notion of successful derivation as in the present paper.

A different yet related area of research is that of the integration of functional and logic programming. Several functional-logic languages such as Kernel-LEAF [9], Babel [16], Curry [11], Escher [13] have been the subject of extensive research. For an overview of the subject see [10]. The present work is not meant to integrate the two programming paradigms, but only to enrich the expressive power of the logic programming.

Finally, the papers [1] and [18] concern methods to translate lazy functional programs into Prolog; the logical computational mechanism they refer to is also the standard one of Prolog. So although the titles seem to indicate otherwise, they are not related to the work presented here.

Acknowledgments. We are grateful to Krzysztof Apt and Vincent van Oostrom for helpful and inspiring discussions. The anonymous referees provided comments that were very useful in improving the presentation of the paper. This research was supported by NWO/SION project number 612-33-003, entitled ‘Parallel declarative programming: transforming logic programs to lazy functional programs’.

References

- [1] S. Antoy. Lazy evaluation in logic. In Maluszynski and M. Wirsing, editors, *Proceedings of PLILP '91*, volume 528 of LNCS, pages 371–382, Passau, Germany, 1991.
- [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)*, volume 711 of LNCS, pages 1–19, 1993.
- [4] 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)*, volume 936 of LNCS, 1995.
- [5] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A compositional semantics for logic programs. *Theoretical Computer Science*, 122(1-2):3–47, 1994.
- [6] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The S-semantics approach: Theory and applications. *Journal of Logic Programming*, 19 & 20:149–198, May 1994.
- [7] S. Etalle and F. van Raamsdonk. Beyond success and failure. In J. Jaffar, editor, *Joint International Conference and Symposium on Logic Programming*, pages 190–204, MIT Press, 1998.

- [8] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.
- [9] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel-leaf: A logic plus functional language. *Journal of Computer and System Sciences*, 42(2):139–185, April 1991.
- [10] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
- [11] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.
- [12] 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, 1991.
- [13] J. Lloyd. Combining functional and logic programming languages. In M. Bruynooghe, editor, *Proceedings ILPS'94*, MIT Press, 1994.
- [14] J. Lloyd. *Foundations of Logic Programming*. Symbolic Computation – Artificial Intelligence. Springer-Verlag, Berlin, 1987. Second edition.
- [15] S. Lüttringhaus. An interpreter with lazy evaluation for Prolog with functions. In E. Börger, H. Kleine Büning, and M. M. Richter, editors, *Proceedings of the 2nd Workshop on Computer Science Logic*, volume 385 of *LNCS*, pages 199–225, Berlin, October 1989. Springer.
- [16] J. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: The language babel. *Journal of Logic Programming*, 12(3&4):191–223, 1992.
- [17] L. Naish. An introduction to mu-prolog. Technical Report 82/2, The University of Melbourne, 1982.
- [18] S. Narain. A technique for doing lazy evaluation in Prolog. *Journal of Logic Programming*, 3(3):259–276, 1986.
- [19] V.A. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of concurrent constraint programming. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages*. ACM Press, 1991.
- [20] G. Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *LNCS*. Springer-Verlag, 1995.
- [21] K. Ueda and M. Morita. Moded flat GHC and its message-oriented implementation technique. *New Generation Computing*, 13(1):3–43, 1994.