Strong Termination of Logic Programs

Marc Bezem
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

We study a powerful class of logic programs which terminate for a large class of goals. Both classes are characterized in a natural way in terms of mappings from variable-free atoms to natural numbers. Based on this idea we present a technique which improves the termination behaviour and allows a more multidirectional use of Prolog programs. The class of logic programs is shown to be strong enough to compute every total recursive function. The class of goals considerably extends the variable-free ones.

1980 Mathematics Subject Classification (1985): 68Q40, 68T15.
1987 CR Categories: F.1.1, F.4.1, I.2.3.
Key Words & Phrases: logic programming, termination, recursion theory.
Note: a preliminary version of this paper appeared as [B].

1. INTRODUCTION

Termination of logic programs is of course of utmost importance. The question whether the top-down evaluation of a goal G terminates with respect to a logic program P is actually underspecified, given the fact that this evaluation may depend on the selection of atoms from goals and on the choice of the program clauses. In this paper termination is considered in the strong sense, i.e. irrespective of the selection of atoms in the goal and of the choice of the program clauses. This is the most demanding notion of termination. Less demanding approaches are: (1) termination for a fixed selection rule and for any choice of program clauses; (2) termination for some selection rule, depending on P, G and annotations on G, and for any choice of program clauses. All approaches can be weakened by requiring termination not for *any* but only for *some* choice of program clauses, but we shall not consider that weakening here. The approach under (2) is taken by Ullman and van Gelder in [UvG] and applies to a system like NAIL!. The approach under (1) is taken by Plümer in [P] and applies to Prolog in case the leftmost selection rule is adopted. Our approach applies to both Prolog and NAIL!.

Each approach has its own merits. It will be clear that fixing the selection rule in advance would enable us to prove more goals terminating. However, fixing the selection rule in advance has as disadvantage that the ordering of atoms in goals and bodies of program clauses becomes of crucial importance. On the other hand, our approach sacrifices some expressivity and may introduce some run-time overhead. The approach under (2) preserves independence of the ordering of atoms in goals and bodies of program clauses, but involves compile-time or even run-time overhead and also sacrifices some expressivity.

In general the least Herbrand model M_P of a logic program P, consisting of all variable-free atoms which logically follow from P, is recursively enumerable. Consequently, the above mentioned halting problem is in general unsolvable. Restriction to logic programs P with recursive least Herbrand model M_P does not help very much, not even for variable-free goals, for in general the mechanism of SLD-resolution does not provide a membership test for M_P. Certainly, if a variable-free atom A occurs in M_P, then a breadth-first search

procedure in an SLD-tree of $\leftarrow A$ always yields a successful refutation, but for A not in M_P it can happen that this tree is infinite, so that the search procedure does not terminate.

Let us consider the restriction to the class of determinate programs introduced by Blair in [Bl]. Determinate programs are logic programs with complementary success and finite failure set. Consequently, if P is a determinate program, then a breadth-first search procedure in any fair SLD-tree of a variable-free goal will always terminate. Although this constitutes an improvement, it still leaves a lot to be desired. First, breadth-first search is generally considered inefficient and it is not possible to sharpen the above property of determinate programs to depth-first search procedures. Second, it is desirable to be able to ensure termination for a larger class of goals than just the variable-free ones. Then the program can not only test, but also compute. Third, and less important than the previous two desiderata, we would like to do away with the fairness condition and obtain termination for arbitrary selection rules. Note that these three desiderata ensure the termination of a Prolog-like evaluation of goals from the larger class of goals mentioned above.

The technical tool we shall use is called *level mapping* by Cavedon [C], who studied various classes of logic programs with negation. A level mapping is a function assigning natural numbers to variable-free atoms. Level mappings are a natural refinement of Clark's finite partition of the set of predicate symbols in [Cl]. We show that if a logic program is *recurrent*, i.e. satisfies the condition that heads of variable-free instances of program clauses have higher levels than the atoms occurring in the body of the same instance, then it is terminating with respect to *bounded* goals, i.e. goals whose instances are below some fixed level. This result improves on [C], where only termination with respect to variable-free goals is obtained. We also prove the converse, namely that a logic program is recurrent if it terminates with respect to variable-free goals. In Section 3 we elaborate some examples and show how certain programs can be transformed into recurrent programs, thus improving their termination behaviour. In Section 4 we undertake an extensive study of the recursion theoretic aspects of the class of recurrent programs. Among other results, we show that every total recursive function can be computed by a recurrent program. As the class of recurrent programs can easily be shown to be a strict subclass of the determinate programs, this result improves on Blair [Bl], who proved that every total recursive function can be computed by a determinate program. Section 5 discusses closely related research.

2. RECURRENT PROGRAMS

For definitions, terminology and notation concerning logic programming we refer the reader to [A] or [L]. More specifically, for a logic program P we use U_P, B_P, T_P, $T_P{\uparrow}\alpha$, and $T_P{\downarrow}\alpha$ as abbreviations of, respectively, the Herbrand Universe of P, the Herbrand Base of P, the immediate consequence operator of P, the upward ordinal powers of T_P, and the downward ordinal powers of T_P. Furthermore we (ab)use $T_P{\uparrow}\alpha$, $T_P{\downarrow}\alpha$ as abbreviations of $T_P{\uparrow}\alpha(\emptyset)$, $T_P{\downarrow}\alpha(B_P)$, so that $M_P{=}T_P{\uparrow}\omega$ denotes the least Herbrand model of the logic program P.

DEFINITION 2.1. Let P be a logic program. A *level mapping* for P is a function $|\ |: B_P \rightarrow \mathbf{N}$ of variable-free atoms to natural numbers. For $A \in B_P$ we call $|A|$ the *level* of A.

DEFINITION 2.2. Let P be a logic program and $|\ |$ a level mapping for P. We call P *recurrent with respect to* $|\ |$ if for every variable-free instance $B \leftarrow A_1, \ldots, A_n$ ($n \geq 0$) of a clause from P the level of B is higher than the level of every A_i ($1 \leq i \leq n$). Moreover P is called *recurrent* if P is recurrent with respect to some level mapping for P.

In the sequel we shall show that recurrent programs have a good termination behaviour: for a large class of goals, including the variable-free ones, every SLD-derivation from a recurrent program terminates. This class of goals is characterized by Definition 2.3 below. The underlying idea is to assign elements of a well-founded ordering to these goals in such a way that SLD-derivations correspond to strictly decreasing sequences. Then termination will be ensured since the ordering is well-founded. This idea is quite old and originates from mathematical logic. It has recently been applied to term rewriting systems, see for example

[D]. The well-founded ordering we shall use is called the *multiset ordering*. A *multiset*, sometimes called *bag*, is an unordered sequence. Multisets are like sets, but allow multiple occurrences of identical elements. The multiset ordering over **N** is an ordering of finite multisets of natural numbers such that X is smaller than Y if X can be obtained from Y by replacing one or more elements in Y by any (finite) number of natural numbers, each of which is smaller than one of the replaced elements. See [D] for more information on the multiset ordering and its use in term rewriting systems.

DEFINITION 2.3. An atom A is called *bounded* with respect to a level mapping $|\ |$ if $|\ |$ is bounded on the set [A] of variable-free instances of A. If A is bounded, then $|A|$ denotes the maximum that $|\ |$ takes on [A]. A goal $G = \leftarrow A_1, \ldots, A_n$ is called *bounded* if every A_i $(1 \leq i \leq n)$ is bounded. If G is bounded then $|G|$ denotes the (finite) multiset consisting of the natural numbers $|A_1|, \ldots, |A_n|$.

EXAMPLE 2.4. Consider the following familiar program.

append ([], z,z)←

append ([w|x], y, [w|z]) ← *append* (x, y, z)

Here and below we use the representation of lists such as in Prolog; x,y,z,w are variables. This program is recurrent with respect to the level mapping $|\ |$ defined by $|\mathit{append}(t_1, t_2, t_3)| = \mathit{min}(l(t_1), l(t_3))$, where $l(t)$ denotes the length of the variable-free term t as a list. More precisely: $l([]) = 0$ and $l([h\,|\,t]) = 1 + l(t)$. In general there will be more terms in the Herbrand Universe than just lists of lists. In that case we extend $|\ |$ by putting $l(f(t_1, \ldots, t_n)) = 0$ whenever f is different from the list constructing function symbol $(n \geq 0)$. Note that a simpler definition of $|\ |$ $(= l(t_1)$ or $l(t_3))$ would also make the *append* program into a recurrent program, but would result in a smaller class of bounded goals.

LEMMA 2.5. *Let P be a logic program which is recurrent with respect to a level mapping $|\ |$. Let G be a bounded goal and G' an SLD-resolvent of G from P. Then we have: (i) The goal G' is bounded; (ii) The multiset $|G'|$ is smaller than $|G|$ in the multiset ordering.*

PROOF. Let conditions be as above. As to (i), assume A_i is the selected literal in $G = \leftarrow A_1, \ldots, A_n$, and $A \leftarrow B_1, \ldots, B_k$ $(k \geq 0)$ the program clause used. Then we have $G' = \leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, B_k, A_{i+1}, \ldots, A_n)\theta$, with θ the mgu of A and A_i. We have $[A_j\theta] \subseteq [A_j]$, so $A_j\theta$ is bounded for every $1 \leq j \leq n$. Furthermore, every $C \in [B_j\theta]$ $(1 \leq j \leq k)$ occurs at the right hand side of a variable-free instance of $(A \leftarrow B_1, \ldots, B_k)\theta$, so $|C| < |A\theta|$ since P is recurrent. It follows that $B_j\theta$ is bounded $(1 \leq j \leq k)$, and hence G' is bounded. Moreover we have $|B_j\theta| < |A\theta| = |A_i\theta| \leq |A_i|$ for all $1 \leq j \leq k$. It follows that $|G'|$ is smaller than $|G|$ in the multiset ordering, which proves (ii). □

COROLLARY 2.6. *Every SLD-derivation from a recurrent program starting with a bounded goal terminates.*

PROOF. Immediate, since the multiset ordering over **N** is well-founded. □

Bounded goals may contain variables. The importance of Corollary 2.6 lies in the fact that one doesn't have to be satisfied with only *testing* variable-free goals, but can also *compute* bindings for variables in bounded goals.

DEFINITION 2.7. A goal G is called *terminating* with respect to a logic program P if every SLD-derivation from P starting with G is finite. A logic program P is called *terminating* if every variable-free goal is terminating with respect to P.

Terminating programs have the property that SLD-trees of variable-free goals are finite. Consequently any depth-first search procedure in such an SLD-tree will always terminate. We now proceed by showing that a

logic program is terminating if and only if it is recurrent.

THEOREM 2.8. *Every recurrent program is terminating.*

PROOF. By Corollary 2.6, since variable-free goals are bounded. □

The above theorem was obtained independently by Cavedon in [C], in the more general setting of logic programs with negation. Cavedon's proof is different and the details do not suggest the stronger Corollary 2.6. For the converse of Theorem 2.8 we need a version of the so-called Lifting Lemma. We omit the standard proof; the lemma below also follows from [LS, Lemma 4.1].

LEMMA 2.9. *Let G be a goal, C a program clause and θ a substitution. If $G\theta$ and C have an SLD-resolvent G', then G and C have an SLD-resolvent G'' such that $G' = G''\theta'$ for some substitution θ'.*

THEOREM 2.10. *Every logic program is terminating if and only if it is recurrent.*

PROOF. The if-part is Theorem 2.8. For the converse, assume P is terminating. Consider, for an arbitrary goal G, the set of *all* SLD-derivations starting from G, allowing arbitrary selection of atoms in goals. This set can be structured as a tree, which we call the *LD-tree* of G (the S is dropped since no selection rule is fixed in advance). LD-trees are finitely branching since logic programs are finite and since a goal and a program clause can have only finitely many resolvents (modulo renaming of variables). Since P is terminating it follows by König's Lemma that for every variable-free atom A the LD-tree of $\leftarrow A$ is finite. Hence we can define a level mapping $|\ |: B_P \rightarrow \mathbf{N}$ by taking for $|A|$ the number of nodes in the LD-tree of $\leftarrow A$. It remains to show that P is recurrent with respect to $|\ |$. Let $A\theta \leftarrow B_1\theta, \ldots, B_n\theta$ be a variable-free instance of a program clause in P. We have to show that $|A\theta| > |B_i\theta|$ for all $1 \leq i \leq n$. Consider the LD-tree of $\leftarrow A\theta$. We have that $A\theta\theta = A\theta$, so θ is a unifier of $A\theta$ and A. So for some mgu μ of $A\theta$ and A the LD-tree of $\leftarrow B_1\mu, \ldots, B_n\mu$ is a subtree of the LD-tree of $\leftarrow A\theta$. Now consider an SLD-derivation starting with $\leftarrow B_i\theta$ $(1 \leq i \leq n)$. We have $\theta = \mu\theta'$ for some substitution θ'. Using Lemma 2.9 we can lift this derivation into a derivation starting with $\leftarrow B_i\mu$. This latter derivation can be embedded in a derivation starting with $\leftarrow B_1\mu, \ldots, B_n\mu$. It follows that the LD-tree of $\leftarrow B_i\theta$ has a smaller number of nodes than the LD-tree of $\leftarrow A\theta$, i.e. $|A\theta| > |B_i\theta|$ for all $1 \leq i \leq n$. □

COROLLARY 2.11. *For the level mapping $|\ |$ constructed in the proof above we have for every goal G: G is terminating with respect to P if and only if G is bounded with respect to $|\ |$.*

PROOF. The if-part follows by Corollary 2.6 since P is recurrent with respect to $|\ |$. For the converse, assume that $G = \leftarrow A_1, \ldots, A_n$ $(n \geq 0)$ is terminating with respect to P. We have to prove that every A_i $(1 \leq i \leq n)$ is bounded. Let $1 \leq i \leq n$, then A_i is terminating with respect to P since G is. It follows that the LD-tree of $\leftarrow A_i$ is finite. Let A_i' be a variable-free instance of A_i. Similarly as in the proof above, the LD-tree of $\leftarrow A_i'$ can be embedded in the LD-tree of $\leftarrow A_i$. So $|A_i'|$ is at most the number of nodes in the LD-tree of $\leftarrow A_i$. It follows that $\leftarrow A_i$ is bounded and that $|A_i|$ is at most the number of nodes in the LD-tree of $\leftarrow A_i$. □

The following counterexample shows that Corollary 2.11 can not be strengthened to arbitrary level mappings. Consider the *append* program from Example 2.4. The *append* program is recurrent with respect to the level mapping defined by $append(t_1, t_2, t_3) = l(t_1)$. Now a goal like $\leftarrow append(x, y, [])$ is terminating but not bounded.

We finish this section by characterizing the class of determinate programs in terms of level mappings. Determinate programs are introduced by Blair in [Bl] with the following definition.

DEFINITION 2.12. A logic program P is called *determinate* if $T_P{\downarrow}\omega = T_P{\uparrow}\omega$.

Due to the well-known characterization results [A, Theorem 3.13 and Theorem 5.6] we have that a logic program P is determinate if and only if P has complementary success set and finite failure set. Consequently a breadth-first search procedure in any fair SLD-tree of a variable-free goal will always terminate. The intuition behind the relation between a determinate program P and a level mapping can be explained as follows. If P is determinate and $B \in B_P$ with $B \notin M_P$, then we have $B \notin T{\downarrow}(n+1)$ for some n. If $B \leftarrow A_1, \ldots, A_k$ is a variable-free instance of a clause form P, then $A_i \notin T{\downarrow}n$ for at least one of the A_i's. Hence the maximal level on which a failing head occurs is higher than the maximal level on which all the failing atoms of the body occur. This observation motivates the following definition.

DEFINITION 2.13. Let P be a logic program and $|\ |$ a level mapping for P. We call P *weakly recurrent with respect to* $|\ |$ if for every variable-free instance $B \leftarrow A_1, \ldots, A_n$ of a clause from P the following holds: if $B \notin M_P$, then there exists $1 \leq i \leq n$ such that $A_i \notin M_P$ and B has a higher level than A_i. P is called *weakly recurrent* if P is weakly recurrent with respect to some level mapping for P.

We proceed by showing that a logic program is determinate if and only if it is weakly recurrent. Since recurrent trivially implies weakly recurrent, it follows that every terminating program is determinate (of course this can also be seen directly). The converse does not hold: $p \leftarrow, p \leftarrow p$ form a logic program which is determinate but not terminating.

LEMMA 2.14. *Let P be weakly recurrent with respect to a level mapping $|\ | : B_P \rightarrow \mathbf{N}$. Then, for all natural numbers n, $T_P{\downarrow}n - T_P{\uparrow}\omega$ contains only atoms of level $\geq n$.*

PROOF. By induction on n, recalling that the immediate consequence operator T_P of P satisfies $T_P{\uparrow}\alpha \subseteq T_P{\downarrow}\beta$ for all ordinals α, β, so in particular for $\alpha = \omega$, $\beta = n$. Let P be weakly recurrent with respect to $|\ |$. We trivially have that $T_P{\downarrow}0 - T_P{\uparrow}\omega$ contains only atoms of level ≥ 0. Assume $T_P{\downarrow}n - T_P{\uparrow}\omega$ contains only atoms of level $\geq n$ and consider the case $n+1$. By definition we have $T_P{\downarrow}(n+1) = T_P(T_P{\downarrow}n)$. Using the induction hypothesis the situation can now be depicted as in Figure 1.

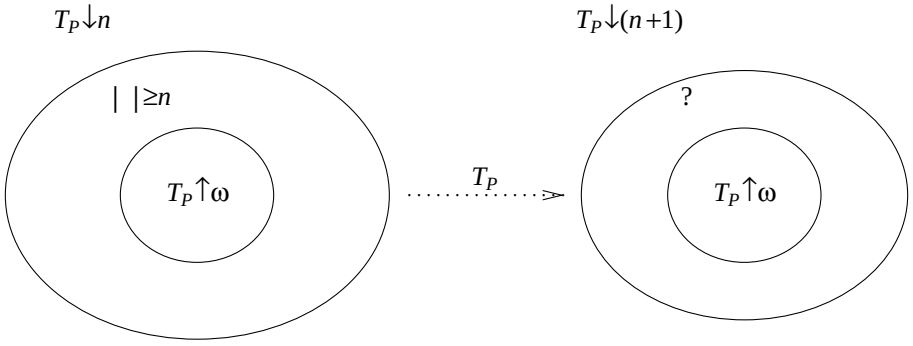

Figure 1

Let $B \in T_P{\downarrow}(n+1)$; then there exists (by the definition of T_P) a variable-free instance $B \leftarrow A_1, \ldots, A_k$ ($k \geq 0$) of a clause from P such that $A_i \in T_P{\downarrow}n$ for all $1 \leq i \leq k$. If $B \notin T_P{\uparrow}\omega$, then there exists $1 \leq i \leq k$ such that $A_i \notin T_P{\uparrow}\omega$ and $|B| > |A_i|$, since P is weakly recurrent. By the induction hypothesis we have $|A_i| \geq n$, so $|B| \geq n+1$. This completes the induction step. □

THEOREM 2.15. *If a logic program P is weakly recurrent, then P is determinate.*

PROOF. Observe that $T_P{\downarrow}\omega - T_P{\uparrow}\omega = \bigcap_{n<\omega} (T_P{\downarrow}n - T_P{\uparrow}\omega)$. By Lemma 2.14 this latter intersection is empty. □

THEOREM 2.16. *Every logic program is determinate if and only if it is weakly recurrent.*

PROOF. The if-part is Theorem 2.15. For the converse, assume P is determinate. Define a level mapping $|\ |: B_P \to \mathbf{N}$ by taking for $|A|$ the greatest k such that $A \in T_P{\downarrow}k$ if $A \notin T_P{\uparrow}\omega$, and 0 otherwise. Since $T_P{\downarrow}\omega = T_P{\uparrow}\omega$ this level mapping is well-defined. Moreover P is weakly recurrent with respect to $|\ |$. For, let $B \leftarrow A_1, \ldots, A_n$ be a variable-free instance of a clause from P. If $B \notin T_P{\uparrow}\omega$, then there exists $1 \leq i \leq n$ such that $A_i \notin T_P{\uparrow}\omega$ and $|A_i|$ is minimal. It follows that $B \in T_P{\downarrow}(|A_i|+1)$, so $|B| > |A_i|$. □

3. SOME ELABORATED EXAMPLES

To illustrate the use of recurrent programs in improving the termination behaviour of logic programs we elaborate in this section three examples. By 'Prolog program' we mean a logic program that is executed by a depth-first search in the SLD-tree of the given goal, using the leftmost selection rule.

An often mentioned advantage of logic (relational) programming over functional programming is *multidirectionality*. This means that for example goals like $\leftarrow p(t,x)$ and $\leftarrow p(x,t)$ can be evaluated with respect to the same program defining the relation p. If the relation p is the graph of a function, then the same program computes the function value of t by the first goal and the inverse images of t by the last. Although this may be true in theory, the Prolog practice shows few examples of multidirectional programs. The NAIL! practice is better in this respect, since in NAIL! the selection of atoms in a goal is depending on the binding pattern of that goal. Our examples also show how recurrent programs can be used in a multidirectional way when executed as Prolog programs.

The first example, merging two lists of natural numbers, is a variation of an example from [N], also discussed in [UvG]. In the second example we show how the familiar Ackermann function can be computed by a recurrent program. As a Prolog program, the recurrent implementation of the Ackermann function can also be used to compute inverse images of the Ackermann function. This is a definite improvement over the naive, straightforward Prolog implementation of the Ackermann function. Our third example, although still small-scale, is of a more practical nature: the transformation of a well-known mergesort program, which can also be found in [UvG], into a recurrent program. The benefits here are again a better termination behaviour, allowing different uses of the mergesort program, for example as a permutation generator.

In all cases the underlying idea is that of using the arguments of a predicate as a depth-bound. This technique may be applicable in many practical programs. It should be stressed, however, that application is restricted to cases in which the depth-bound can indeed be expressed in terms of one of the arguments. For example, when sorting a list the depth-bound can be expressed in terms of the length of the list. Multidirectionality can be achieved in this case since the sorted list is of the same length: see Example 3.4 below.

EXAMPLE 3.1. Consider the following program (x,y,z,v,w are variables).

$$lessthan(0, s(y)) \leftarrow$$
$$lessthan(s(x), s(y)) \leftarrow lessthan(x,y)$$

$$merge(x, [\,], x) \leftarrow$$
$$merge([\,], x, x) \leftarrow$$
$$merge([v\,|\,x], [w\,|\,y], [w\,|\,z]) \leftarrow lessthan(w,v), merge([v\,|\,x], y, z)$$
$$merge([v\,|\,x], [w\,|\,y], [v\,|\,z]) \leftarrow lessthan(v,w), merge(x, [w\,|\,y], z)$$
$$merge([v\,|\,x], [v\,|\,y], [v\,|\,z]) \leftarrow merge(x, [v\,|\,y], z)$$

The intent is that $merge(l_1, l_2, l_3)$ holds when l_3 is the merge of the sorted lists l_1 and l_2 of natural numbers.

The program above is recurrent, which can be seen as follows. Define $\|t\|=0$ for any term t which is not of the form $s(t')$ for any term t', and $\|s(t')\|=1+\|t'\|$ otherwise. Furthermore, define $\Sigma t=0$ if the term t is not of the form $[h\,|\,l]$ for any terms h,l, and $\Sigma[h\,|\,l]=1+\|h\|+\Sigma l$ else. Define a level mapping by putting $|lessthan(t_1,t_2)|=min(\|t_1\|,\|t_2\|)$ and $|merge(l_1,l_2,l_3)|=min(\Sigma l_1+\Sigma l_2,\Sigma l_3)$. It is not difficult to see that this level mapping makes the above program into a recurrent one. Among the bounded goals are goals of the form $\leftarrow merge(l_1,l_2,x)$, with l_1 and l_2 lists of natural numbers, as well as goals of the form $\leftarrow merge(x,y,l_3)$, with l_3 a list of natural numbers.

EXAMPLE 3.2. The Ackermann function $A:\mathbf{N}\times\mathbf{N}\rightarrow\mathbf{N}$ is a recursive function which is not primitive recursive, and is defined as follows:

$$A(0,n)=n+1;$$

$$A(i+1,0)=A(i,1);$$

$$A(i+1,n+1)=A(i,A(i+1,n)).$$

For every $n\in\mathbf{N}$, let $\bar{n}$ denote the term $s^n(0)$. A straightforward translation of the definition of the Ackermann function suggests the following logic program.

$$p(0,y,s(y))\leftarrow$$

$$p(s(x),0,z)\leftarrow p(x,\bar{1},z)$$

$$p(s(x),s(y),z)\leftarrow p(s(x),y,z'),p(x,z',z)$$

However, this program is not recurrent since it is not terminating. For example, the goal $\leftarrow p(\bar{2},\bar{2},0)$ has as SLD-resolvent $\leftarrow p(\bar{2},\bar{1},x),p(\bar{1},x,0)$, and $\leftarrow p(\bar{1},x,0)$ heads an infinite left-most SLD-derivation by applying the third program clause again and again. Moreover, goals of the form $\leftarrow p(x,y,\bar{n})$ are not correctly evaluated by Prolog. For example the goal $\leftarrow p(x,y,\bar{13})$ is not correctly evaluated by Prolog: after two solutions the Prolog interpreter hits an infinite branch in the SLD-tree. By using the (ad hoc) observations that the Ackermann function is monotonic (so $A(i+1,n)<A(i+1,n+1)$) and that the function value constitutes (in some sense) a logarithmic upper bound on the length of the computation, we devised the following logic program P.

$$p(0,y,s(y),w)\leftarrow$$

$$p(s(x),0,z,w)\leftarrow p(x,\bar{1},z,w)$$

$$p(s(x),s(y),z,s(w))\leftarrow p(s(x),y,z',w),p(x,z',z,s(w))$$

This program is recurrent with respect to the level mapping defined by $|p(t_1,t_2,t_3,t_4)|=\|t_1\|+\|t_4\|$, where $\|\bar{n}\|=n$ for all $n\in\mathbf{N}$. With some technical effort (notably transfinite induction upto ω^2) one proves that for all $n,i,m\in\mathbf{N}$

$$m=A(n,i) \text{ if and only if } p(\bar{n},\bar{i},\bar{m},\bar{m})\in M_P.$$

Hence the Ackermann function is computed by the recurrent program obtained by augmenting P with the clause

$$p_A(x,y,z)\leftarrow p(x,y,z,z),$$

where the level mapping is extended by putting $|p_A(t_1,t_2,t_3)|=1+|p(t_1,t_2,t_3,t_3)|$.

Since P is recurrent, bounded goals, such as $\leftarrow p_A(\bar{1},z,0)$, are correctly evaluated by a depth-first search procedure in the SLD-tree, no matter the ordering of the program clauses and the selection of atoms in goals. However, even the Prolog evaluation of goals of the form $\leftarrow p_A(x,y,\bar{n})$ ($n\in\mathbf{N}$), which are not bounded, has improved, as follows from Claim 3.3 below. Due to the left-most selection rule of Prolog all solutions to such goals are successively found, after which the evaluation correctly terminates.

CLAIM 3.3. *Let P be the recurrent implementation of the Ackermann function from Example 3.2 above. Then, for every $n \in \mathbf{N}$, the Prolog evaluation of the goal $\leftarrow p_A(x,y,\overline{n})$ terminates.*

PROOF. By induction on n we prove: for every term t, any leftmost SLD-derivation starting with $\leftarrow p(t,y,z,\overline{n})$ terminates. Moreover, if such an SLD-derivation is successful, then the computed answer substitution maps t on a numeral. The base case $n=0$ follows immediately, since then only the first two program clauses of P can be applied. Regarding the induction step, assume the case n has been settled and consider an SLD-derivation starting with a goal of the form $\leftarrow p(t,y,z,\overline{n+1})$. If the first program clause is applied to this goal then we are done. If the second program clause is applied, then it must be folowed by either an application of the first program clause, in which case we are done again, or by an application of the third program clause. This latter case yields a goal of the form $\leftarrow p(s(t'),0,z',\overline{n}),p(t',z',z,\overline{n+1})$, where $t \equiv s(s(t'))$. Now we are done by, first, applying the induction hypothesis to the first (left) atom of this goal and, second, if the first atom is successfully refuted with an answer substitution that maps t' on a numeral, by noting that the instantiated second atom is bounded. If the third program clause is applied to the goal $\leftarrow p(t,y,z,\overline{n+1})$, then we obtain an SLD-resolvent $p(s(t'),y',z',\overline{n}),p(t',z',z,\overline{n+1})$ with $t \equiv s(t')$ and $y \equiv s(y')$, and proceed in a similar way as above. This settles the case $n+1$ and hence the induction step. □

EXAMPLE 3.4. The next example is of a more practical nature, although still small-scale. Consider the following mergesort program, also considered in [UvG].

$$
\begin{aligned}
&\mathit{merge}(x, [], x) \leftarrow \\
&\mathit{merge}([], x, x) \leftarrow \\
&\mathit{merge}([v \mid x], [w \mid y], [w \mid z]) \leftarrow v > w, \mathit{merge}([v \mid x], y, z) \\
&\mathit{merge}([v \mid x], [w \mid y], [v \mid z]) \leftarrow v \leq w, \mathit{merge}(x, [w \mid y], z) \\
&\\
&\mathit{split}([], [], []) \leftarrow \\
&\mathit{split}([v \mid x], [v \mid y], z) \leftarrow \mathit{split}(x, z, y) \\
&\\
&\mathit{mergesort}([], []) \leftarrow \\
&\mathit{mergesort}([v], [v]) \leftarrow \\
&\mathit{mergesort}([v, w \mid x], y) \leftarrow \mathit{split}([v, w \mid x], z_1, z_2), \\
&\qquad\qquad\qquad\qquad \mathit{mergesort}(z_1, z_3), \\
&\qquad\qquad\qquad\qquad \mathit{mergesort}(z_2, z_4), \\
&\qquad\qquad\qquad\qquad \mathit{merge}(z_3, z_4, y)
\end{aligned}
$$

In this program we use an extension of logic programming with the arithmetical predicates $>$ and $\leq$. The Herbrand Universe is assumed to contain any floating point number as a constant and is closed under list construction. The arithmetical predicates have their usual meaning, but arithmetical atoms can only be evaluated when they are variable-free. For example $2<1$ fails whereas the evaluation of $2<x$ leads to an error. This enforces us to require that the selection rule is *safe* with respect to the arithmetic, i.e. only selects arithmetical atoms when they are variable-free. This requirement introduces problems with completeness: if a goal consists entirely of arithmetical atoms containing variables (for example $\leftarrow 2<x$), then no atom can be selected and there is no resolvent, although there may be correct answer substitutions (e.g. $\{x/3\}$ in the

example). These completeness problems do not interest us here and we choose our examples such that they are avoided.

The mergesort program above is not recurrent, due to the introduction of new variables in bodies of clauses (z_1-z_4 in the last program clause). However, while sorting a list of length k, it can easily be seen that all these variables become instantiated by lists of length at most k. This supports the intuition that all (implicit) existential quantifications are bounded quantifications. Based on this observation we devised the following program P:

$$merge(x, [], x, l) \leftarrow$$
$$merge([], x, x, l) \leftarrow$$
$$merge([v \mid x], [w \mid y], [w \mid z], [h \mid l]) \leftarrow v > w, merge([v \mid x], y, z, l)$$
$$merge([v \mid x], [w \mid y], [v \mid z], [h \mid l]) \leftarrow v \leq w, merge(x, [w \mid y], z, l)$$

$$split([], [], [], l) \leftarrow$$
$$split([v \mid x], [v \mid y], z, [h \mid l]) \leftarrow split(x, z, y, l)$$

$$mergesort([], [], l) \leftarrow$$
$$mergesort([v], [v], l) \leftarrow$$
$$mergesort([v, w \mid x], y, [h \mid l]) \leftarrow split([v, w \mid x], z_1, z_2, [h \mid l]),$$
$$mergesort(z_1, z_3, l),$$
$$mergesort(z_2, z_4, l),$$
$$merge(z_3, z_4, y, [h \mid l])$$

This program is recurrent with respect to a level mapping assigning to arithmetical atoms level 0, to *merge* and *split* atoms a level which equals the length of the list in the fourth argument, and to *mergesort* atoms a level which equals one plus the lenght of the list in the third argument. Goals of the form $\leftarrow mergesort(l, y, l)$, where l is a list of floating point numbers, are bounded and hence terminate with respect to P. When a safe selection rule is used the goal $\leftarrow mergesort(l, y, l)$ is successfully refuted, binding the variable y to the sorted permutation of l. However, also goals of the form $\leftarrow mergesort(x, l, l)$ are bounded and hence terminate with respect to P. This opens up a possibility of using P to compute the 'inverse' of the mergesort relation, so that the same program could also be used as a permutation generator. Indeed, when using a safe selection rule and provided that l is a *sorted* list of floating point numbers, the goal $\leftarrow mergesort(x, l, l)$ is successfully refuted, binding the variable x to a permutation of l. Backtracking yields all permutations of l. If l is not sorted, then the goal $\leftarrow mergesort(x, l, l)$ finitely fails. With the naive implementation of the mergesort algorithm one can only achieve this using a special purpose selection rule.

4. RECURSION THEORETIC CONSIDERATIONS

In this section we study the recursion theoretic properties of the class of recurrent programs. We assume the reader is familiar with the basic facts of recursion theory. We use the denotations PR, $PR(f)$, R, RE for, respectively, the classes of primitive recursive functions, functions which are primitive recursive in f, recursive functions and recursively enumerable sets. We start with establishing upper bounds on the recursion theoretic complexity of the least Herbrand model of a (weakly) recurrent program, preparing by the

following lemma.

LEMMA 4.1. *Let P be recurrent with respect to a level mapping $|\ |: B_P \to \mathbf{N}$. Then, for all natural numbers n, $T_P\downarrow n - T_P\uparrow n$ contains only atoms of level $\geq n$.*

PROOF. By induction on n, similarly to the proof of Lemma 2.14 (see also [C]). □

LEMMA 4.2. *Let P be a logic program which is recurrent with respect to a level mapping $|\ |$. Then M_P is primitive recursive in $|\ |$.*

PROOF. Observe that $T_P\uparrow n \subseteq M_P \subseteq T_P\downarrow n$ for all n. Using Lemma 4.1 it follows that $M_P - T_P\uparrow n$ contains only atoms of level at least n. So we have:

$$A \in M_P \text{ iff } \exists k \leq |A|+1 \ \ A \in T_P\uparrow k\,.$$

After appropriate coding the sets $T_P\uparrow k$ can easily be seen to be primitive recursive. Moreover $|\ |$ may be seen as a function of natural numbers. Since the quantification in the right hand side above is bounded by $|A|+1$ it follows from well-known results in recursion theory that M_P is primitive recursive in $|\ |$. □

THEOREM 4.3. *Let P be a weakly recurrent program. Then M_P is recursive.*

PROOF. If P is weakly recurrent, then by Theorem 2.15 P is determinate, i.e. $T_P\downarrow\omega = T_P\uparrow\omega$. Generally, $T_P\uparrow\omega$ and $B_P - T_P\downarrow\omega$ are *RE*. So both M_P and its complement are *RE*. Now it follows from a well-known result in recursion theory that M_P is recursive. □

THEOREM 4.4. *If P is recurrent, then M_P is in $PR(|\ |) \cap R$.*

PROOF. By Lemma 4.2 and Theorem 4.3, since recurrent programs are weakly recurrent. □

This theorem shows that the computational power of a recurrent program largely depends on the level mapping $|\ |$. Clearly the most simple level mappings are the most appealing. For example, the level mappings used in the examples in Section 3 above are, after appropriate coding, all primitive recursive. Consequently, by Lemma 4.2, the least Herbrand models of the programs involved are primitive recursive. (Of course this latter fact can also be seen directly. It may seem paradoxical that the graph of the Ackermann function is primitive recursive. However, it so happens that some non primitive recursive functions do have primitive recursive graphs.) Although programs with primitive recursive semantics are computationally rather weak, they are not to be depreciated. Among them are many programs met in practice (such as list processing programs). Moreover we can prove the following version of Kleene's normal form theorem from recursion theory.

THEOREM 4.5. (Normal form theorem for logic programs.) *For every logic program P there exists a program P' (in which the same predicate symbols occur with incremented arities) which is recurrent with respect to a primitive recursive level mapping such that for all $p(t_1, \ldots, t_n) \in B_P$*

$$p(t_1, \ldots, t_n) \in M_P \text{ iff } \exists t'\, p(t_1, \ldots, t_n, t') \in M_{P'}\,.$$

PROOF. We tacitly assume that no predicate symbol occurs in P with more than one arity. First we fix the alphabet of P'. Every predicate symbol occurring in P with arity $n \geq 0$ occurs in P' with arity $n+1$. Furthermore, if the alphabet of P does not contain a unary function symbol s, then we add s to obtain the alphabet of P'. Now the clauses of P' are obtained by applying a simple transformation to the clauses of P, of which we shall give two typical examples:

$$p(t_1, t_2) \leftarrow$$

is transformed into

$$p(t_1, t_2, x) \leftarrow,$$

and

$$p(t_1, t_2) \leftarrow q(t_3),\ p(t_4, t_5)$$

is transformed into

$$p(t_1, t_2, s(x)) \leftarrow q(t_3, x),\ p(t_4, t_5, x),$$

where x is a variable not occurring in $t_1, \ldots, t_5$. It is immediately clear that P' thus obtained is recurrent with respect to the level mapping $|\ |: B_{P'} \rightarrow \mathbf{N}$ defined by

$$|p(t_1, \ldots, t_n, t')| = \|t'\|,$$

where $\|\|$ is defined by $\|s(t)\| = 1 + \|t\|$ and $\|f(t_1, \ldots, t_k)\| = 0$ whenever f differs from s ($k \geq 0$). Moreover, we obviously have for all $p(t_1, \ldots, t_n) \in B_P$:

$$p(t_1, \ldots, t_n) \in M_P \text{ iff } \exists t'\, p(t_1, \ldots, t_n, t') \in M_{P'}. \ \square$$

COROLLARY 4.6. *Let P'' be the program P' augmented with clauses $p(x_1, \ldots, x_n) \leftarrow p(x_1, \ldots, x_n, x')$ for all relations p occurring in P. Then we have $M_P = M_{P''} \cap B_P$. The program P'' may be viewed as a normal form of P. Note that P'' satisfies $T_{P''} \uparrow \omega = T_{P''} \downarrow (\omega+1)$.*

From Theorem 4.4 we know that both $PR(|\ |)$ and R are upper bounds of the recursion theoretic complexity of the least Herbrand model of a recurrent program. We experienced considerable difficulty in establishing the exact computational power of the class of recurrent programs. These difficulties can be traced back to the fact that the class of recurrent programs (as well as the class of weakly recurrent programs) does not enjoy a natural closure property such as composition. Let us first define how a logic program computes a function.

DEFINITION 4.7. For every $n \in \mathbf{N}$, let $\overline{n}$ denote the term $s^n(0)$. A logic program P with Herbrand Universe $\{\overline{n} \mid n \in \mathbf{N}\}$ is said to compute a partial function $f: \mathbf{N}^k \rightarrow \mathbf{N}$ if for some predicate symbol p we have

$$f(n_1, \ldots, n_k) = n \text{ iff } p(\overline{n}_1, \ldots, \overline{n}_k, \overline{n}) \in M_P$$

for all $n_1, \ldots, n_k, n \in \mathbf{N}$.

DISCUSSION.

The technical obstacle mentioned above becomes apparent when one tries to prove that the class of functions which can be computed by recurrent programs is closed under composition. If $f(x) = g(h(x))$, then it seems natural to add the clause

$$p_f(x,y) \leftarrow p_h(x,z),\ p_g(z,y)$$

to (disjoint) recurrent programs computing g and h in order to obtain a program that computes f. However, this latter program will in general not be recurrent, due to the presence of the variable z in the body of the above clause. Consider, for example, $p_h(x, 0) \leftarrow$ and $p_g(0,0) \leftarrow$, $p_g(s(x),y) \leftarrow p_g(x,y)$, with p_f as defined above. Then $\leftarrow p_f(0,0)$ heads an infinite (right-most) SLD-derivation.

A posteriori, using the knowledge that the computational power of the class of recurrent programs is R, it can easily be understood why the class of recurrent programs is not closed under composition. Namely, whereas the class of functions from R is closed under composition, the class of relations from R (i.e. relations whose characteristic function is in R) is not. For example in the clause above, the relation p_f is defined as the composition of the relations p_g and p_h, so that

$$p_f(x,y) \leftrightarrow \exists z\,(p_h(x,z) \wedge p_g(z,y))$$

holds in the minimal Herbrand model of the program. Due to the existential quantification, the relation p_f will in general be *RE* when the relations p_g and p_h are recursive. A notable exception is of course the special case of bounded existential quantification. This case applies to graphs of total recursive functions, where $z=h(x)$, but this is obscured by the underspecification resulting from the representation of functions by their graphs. This observation touches the heart of a disadvantage of logic (relational) programming as compared to functional programming. In our examples from the previous section the bounded existential quantification is made explicit, resulting in logic programs with a better termination behaviour. □

An elegant solution to the problem of the computational power of recurrent programs becomes possible after the choice of a convenient machine model, namely the register machine (see [S1]).

DEFINITION 4.8. A *register machine program* is a finite sequence $I_1, \ldots, I_n$ of instructions which operate on registers $x_1, \ldots, x_m$, where each instruction is of one of the following two forms (with $1 \le i \le m$, $1 \le j \le n+1$).

$x_i := x_i + 1$

IF $x_i \neq 0$ *THEN* $x_i := x_i - 1$ *AND GOTO* j

The program is completed with a halt instruction I_{n+1}. Execution of a register machine program with respect to given contents $n_1, \ldots, n_m \in \mathbf{N}$ of the registers $x_1, \ldots, x_m$ starts from I_1, executing the instructions in the obvious sequential way, and terminates when the halt instruction I_{n+1} is reached. A register machine program is said to compute a (partial) function $f : \mathbf{N}^k \to \mathbf{N}$ $(k \le m)$ if, for all $n_1, \ldots, n_k \in \mathbf{N}$, the execution of the register machine program with respect to contents $n_1, \ldots, n_k, 0, \ldots, 0$ of the registers terminates with value $f(n_1, \ldots, n_k)$ in register x_1 (if this function value is not defined, then the execution is not allowed to terminate).

A classical result from [S1] states that every partial recursive function can be computed by a register machine program. This same result can be obtained for logic programs by transforming register machine programs into logic programs. This can be done in a very natural way, as shown in [S]. Every instruction corresponds to the definition of a predicate symbol, every register corresponds to an argument of these predicate symbols. All predicate symbols have one more argument through which the function value is passed. The predicate symbol p_k corresponding to the instruction I_k has one of the following two definitions, corresponding to the possible forms of the instruction as displayed above (the second definition consists of two clauses).

$$p_k(x_1, \ldots, x_i, \ldots, x_m, y) \leftarrow p_{k+1}(x_1, \ldots, s(x_i), \ldots, x_m, y)$$

$$p_k(x_1, \ldots, s(x_i), \ldots, x_m, y) \leftarrow p_j(x_1, \ldots, x_i, \ldots, x_m, y)$$
$$p_k(x_1, \ldots, 0, \ldots, x_m, y) \leftarrow p_{k+1}(x_1, \ldots, 0, \ldots, x_m, y)$$

The halt instruction corresponds to the following program clause, by which the function value is transported from register x_1 to the last argument of the predicate symbols.

$$p_{n+1}(x_1, \ldots, x_m, x_1) \leftarrow$$

If we finally add the clause

$$p(x_1, \ldots, x_l, y) \leftarrow p_1(x_1, \ldots, x_l, 0, \ldots, 0, y)$$

to the logic program corresponding to a register machine program, then it is easy to see that both compute the same partial function. It is also easy to see that the SLD-trees corresponding to register machine computations consist of one single path, and that this path is finite if and only if the computation terminates. However, this does not imply that the logic program corresponding to a register machine program which computes a total recursive function is terminating. For example, it is not guaranteed that the goal

$\leftarrow p_7(0,\ldots,0,0)$ terminates (the malicious programmer could have taken *GOTO* 7 for I_7, taking care that any normal execution of his program never falls into this trap). Evidently we need a stronger result than can be extracted from [S1]. Goals of the form $\leftarrow p_k(\overline{n}_1,\ldots,\overline{n}_m,\overline{n})$ correspond to starting the execution of the register machine program at the k-th instruction, with the registers initialized on $n_1,\ldots,n_m$. Hence the result we actually need is that every total recursive function can be computed by a register machine program which always halts, irrespective of the initialization of the registers and of the instruction at which the execution starts. This result was proved by Shepherdson in [S2]. An analogous result for Turing machines has been proved by Davis [Da, Theorem V.3]. We are now in a position to state and prove the main result of this paper.

THEOREM 4.9. *Every total recursive function can be computed by a recurrent program.*

PROOF. By Theorem 2.10 and [S2, Theorem 4], using the transformation of register machine programs to logic programs as described above. Note that the level of a variable-free atom is independent of its last argument. □

At first sight the statement 'logic program P is recurrent' seems to be analytical, since the definition involves the existence of a level mapping. However, by using the connection with the termination behaviour of P as stated in Theorem 2.10, the complexity of the above statement can be shown to be considerably lower than analytical. In fact the recursion theoretic complexity of the statement 'logic program P is recurrent' is Π^0_2-complete, i.e. the complexity of the uniform halting problem. This will be proved in the next paragraphs.

LEMMA 4.10. *The statement 'logic program P is terminating' has recursion theoretic complexity* Π^0_2.

PROOF. First we observe that P is terminating if and only if for every $A\in B_P$ the goal $\leftarrow A$ is terminating with respect to P. We recall from the proof of Theorem 2.10 that the *LD-tree* of $\leftarrow A$ is the tree of *all* SLD-derivations (allowing arbitrary selection of atoms in goals) starting with a goal $\leftarrow A$, and that this tree is finitely branching. Hence by König's Lemma we have that P is terminating if and only if for every $A\in B_P$ the LD-tree of $\leftarrow A$ is finite. After suitable encoding this latter statement is of the form $\forall x\,\exists y R(x,y)$, where $R(x,y)$ is a recursive relation expressing 'x is the code of a variable-free atom, say A, and y is the code of the LD-tree of the corresponding goal $\leftarrow A$'. It follows that the statement 'P is terminating' is Π^0_2. □

The argument for Π^0_2-completeness is rather more involved. We shall show that the statement 'i is an index of a total recursive function', which is known to be Π^0_2-complete, can be reduced to the statement 'P is a terminating program', thus settling the claim. The statement 'i is the index of a total recursive function', i.e. the uniform halting problem, can be formalized by $\forall j\,\exists k\,T(i,j,k)$, where T is Kleene's primitive recursive T-predicate. For reasons that will become clear below we assume that T is monotonic in the third argument, i.e. $(T(i,j,k)\wedge l\geq k)\rightarrow T(i,j,l)$ for all $i,j,k,l\in\mathbf{N}$. This is a natural assumption about a T-predicate: if it is not satisfied, then take the primitive recursive predicate T' defined by $T'(i,j,k)\leftrightarrow\exists l\leq k\,T(i,j,l)$.

We observe that the characteristic function X_T of the T-predicate is primitive recursive. Consequently, by Theorem 2.10 and Theorem 4.9, there exists a terminating program P_T and a predicate symbol p_T such that for all $i,j,k\in\mathbf{N}$ and $l\in\{0,1\}$ we have

$$\mathrm{X}_T(i,j,k)=l \text{ iff } p_T(\overline{i},\overline{j},\overline{k},\overline{l})\in M_{P_T}.$$

By inspection of the proof of Theorem 4.9 it follows that P_T consists of a clause

$$p_T(x_1,x_2,x_3,x)\leftarrow p_1(x_1,x_2,x_3,0,\ldots,0,x)$$

and a clause

$$p_{n+1}(x_1,\ldots,x_m,x_1)\leftarrow$$

besides definitions of the predicates $p_1, \ldots, p_n$ corresponding to the instructions of the register machine program computing X_T. We shall transform P_T into programs P_i which, for fixed i and given j, computes successively $X_T(i,j,0)$, $X_T(i,j,1)$, ... until a zero is found. The transformation will be such that, for every $i \in \mathbf{N}$, i is an index of a total recursive function if and only if P_i is terminating. As the construction of P_i is uniformly recursive in i, this establishes the desired reduction of the uniform halting problem to the statement 'logic program P is terminating'.

The transformation of P_T into P_i requires a few technicalities. Since a register machine computation may affect the registers so that their content after the register machine computation is different from the content before, it will be necessary to use extra variables to store the second argument of X_T and the (increasing) third argument. Moreover, since the output of the computation of $X_T(i,j,k)$ is only used to decide to terminate or to go on with the computation of $X_T(i,j,k+1)$, the last argument of all the predicate symbols occurring in P_T is superfluous. Hence all these last arguments can be skipped. Thereafter we extend every predicate symbol with two new variables y and z which do not occur in P_T. So we have for example the following clauses in P_i:

$$p_T(x_1,x_2,x_3,y,z) \leftarrow p_1(x_1,x_2,x_3,0,\ldots,0,y,z)$$

and

$$p_{n+1}(x_1,\ldots,x_m,y,z) \leftarrow$$

The successive computation of X_T-values until a zero is found is achieved by replacing the last clause by the following two clauses:

$$p_{n+1}(0,\ldots,x_m,y,z) \leftarrow$$

$$p_{n+1}(s(x_1),\ldots,x_m,y,z) \leftarrow p_T(\bar{i},y,s(z),y,s(z))$$

This completes our program P_i. Note that i occurs in the last clause of the definition of p_{n+1}, which makes P_i depending on i. The successive computation of $X_T(i,j,0), X_T(i,j,1)$, ... until a zero is found is sparked off by the goal $\leftarrow p_T(\bar{i},\bar{j},0,\bar{j},0)$. Note that if $\forall j \exists k T(i,j,k)$, then for all $n_1,\ldots,n_5 \in \mathbf{N}$ the goal $\leftarrow p_T(\bar{n}_1,\ldots,\bar{n}_5)$ terminates with respect to P_i for the following two reasons. First, due to the crucial occurrence of i in P_i the following sequence of X_T values will be computed until a zero is found: $X_T(n_1,n_2,n_3), X_T(i,n_4,n_5+1), X_T(i,n_4,n_5+2), \ldots$. Second, due to the assumption that the T-predicate is monotonic in its third argument, a zero in the above sequence of X_T-values will be found, even if n_5+1 surpasses the least k such that $T(i,n_4,k)$. After some reflection it follows for every $i \in \mathbf{N}$ that i is an index of a total recursive function if and only if P_i is terminating. We are now in the position to obtain the following result.

THEOREM 4.11. *The statement 'logic program P is recurrent' has recursion theoretic complexity Π^0_2-complete.*

PROOF. By Theorem 2.10, Lemma 4.10 and the argument above. □

5. CLOSELY RELATED RESEARCH

Given the vast amount of research on termination of logic programs it is impossible to give a comprehensive account of the state-of-the-art in a section of a journal paper. A key reference for an approach complementary to ours is the monograph of Plümer [P]. As an alternative to our bounded goals, Bossi, Cocco and Fabris develop in [BCF] a concept of *rigidity* of terms (identifying invariance of the level mapping under instantiation), which allows to extend termination analysis techniques beyond beyond the variable-free case. An important objective of present and future research is the automation of proving termination of logic programs, already touched upon by [UvG], [P], [BCF] and [VS]. In the rest of this section we review closely

related research in some more detail.

Together with Apt we extend in [AB] the ideas of this paper to the case of logic programs with negation, with $|\neg A| = |A|$ as the obvious extension of the level mapping. The resulting class of general programs is called the class of *acyclic programs*. Among other results it is proved in [AB] that for acyclic programs all known approaches to the semantics of logic programs with negation coincide, and that SLDNF- and SLS-resolution coincide and are decidable rules of inference for such programs. It should be noted that some of the results of [AB] and of Section 2 of the present paper, are found independently by Cavedon in [C].

In the present paper, as well as in [AB] and [C], a rather 'holistic' approach to termination is taken: termination for *all* variable-free atoms and for *all* selection rules; the program is taken as *one whole* simultaneous definition of a set of predicates. Consequently, all this research is of a theoretical nature. In practice one usually has to deal only with one specific selection rule, and one is usually interested in termination for a specific set of (not nececesarily variable-free) atoms. Moreover it can be profitable to take the so-called *dependency graph* of the program into account. It is therefore very fortunate that a number of researchers have transformed and refined the technique exhibited in this paper into more practically applicable ones.

To begin with, Apt and Pedreschi have studied *left* termination, i.e. termination with respect to the leftmost (Prolog) selection rule. In [AP90] and [AP91], respectively, they generalize recurrent and acyclic programs to so-called *acceptable* programs and prove that left terminating is equivalent to acceptable (on the condition of non-floundering in the case of negation). The proof of this analogue of Theorem 2.10 is refined hightech, involving a double multiset ordering and subtle counting arguments. Among the programs that are acceptable but not acyclic are a quicksort program and a cycle-avoiding transitive closure program for *finite* graphs. General transitive closure is not acceptable, since it can lead to recursive enumerability (see the discussion in Section 4).

In a separate development, Verschaetse and de Schreye in [VS], and together with Bruynooghe in [SVB], study termination and left termination with respect to an arbitrary set S of (not necessarily variable-free) atoms. They introduce relativizations with respect to S of the notions (left) terminating program, level mapping, recurrent and acceptable program. The relativized notions are proper generalizations: e.g. a program P is recurrent if and only if P is recurrent with respect to B_P. For the relativized notions the main results from Section 2 go through, for example Theorem 2.10 generalizes into: every logic program is terminating with respect to S if and only if it is recurrent with respect to S. Taking the dependency graph of the program into account the authors come up with an interesting sufficient condition for recurrent/acceptable with respect to S. They convincingly show the following advantages of their approach: (i) more natural level mappings can be obtained; (ii) techniques such as abstract interpretation can be applied; (iii, very attractive) termination proofs may be automated.

ACKNOWLEDGEMENTS. I would like to thank Krzysztof Apt, Johan van Benthem, Roland Bol, Jan Heering, Jan Willem Klop, Hans Mulder, Dirk Roorda and Professor J.C. Shepherdson for helpful advice and stimulating discussions on this paper. The research for this paper has been partially supported by the ESPRIT Basic Research Action 3020.

REFERENCES

[A] K.R. APT, *Logic Programming*, in: J. VAN LEEUWEN (editor), *Handbook of theoretical computer science*, Vol. B, Ch. 10, pp. 493-574, North Holland, Amsterdam, 1990.

[AB] K.R. APT, M.A. BEZEM, *Acyclic programs*, New Generation Computing 9, pp.335-363, 1991.

[AP90] K.R. APT, D. PEDRESCHI, *Studies in pure Prolog: termination*, in: J.W. LLOYD (editor), Symposium on computational logic, pp. 150-176, Springer-Verlag, Berlin, 1990.

[AP91] K.R. APT, D. PEDRESCHI, *Proving termination of general Prolog programs*, in: T. ITO, A. MEYER, (editors), Proceedings of the international conference on theoretical aspects of computer software, LNCS 526, pp. 265-289, Springer-Verlag, Berlin, 1991.

[B] M. BEZEM, *Characterizing termination of logic programs with level mappings*, in: E. L. LUSK, R. A. OVERBEEK, (editors), Proceedings of the North American conference on logic

programming, pp. 69-80, MIT Press, 1989.

[BCF] A. BOSSI, N. COCCO, M. FABRIS, *Proving termination of logic programs by exploiting term properties*, in: S. ABRAMSKY, T.S.E. MAIBAUM, Proceedings CCPSD-TAPSOFT 91, LNCS 494, pp. 153-180, Springer-Verlag, Berlin, 1991.

[Bl] H. BLAIR, *Decidability in the Herbrand base*, manuscript (presented at the Workshop on foundations of deductive databases and logic programming, Washington D.C., August 1986).

[C] L. CAVEDON, *Acyclic logic programs and the completeness of SLDNF-resolution*, Theoretical Computer Science 86, pp. 81-92, 1991.

[Cl] K.L. CLARK, *Negation as failure*, in: H. GALLAIRE, J. MINKER (editors), *Logic and data bases*, pp. 293-322, Plenum Press, New York, 1978.

[D] N. DERSHOWITZ, *Termination of rewriting*, Journal of Symbolic Computation 3, pp. 69-116, 1987.

[Da] M. DAVIS, *A note on universal Turing machines*, in: J. MCCARTHY, C.J. SHANNON (editors), *Automata studies*, pp. 167-175, Princeton University Press, Princeton, 1956.

[L] J.W. LLOYD, *Foundations of Logic Programming*, Second Edition, Springer-Verlag, Berlin, 1987.

[LS] J.W. LLOYD, J.C. SHEPHERDSON, *Partial evaluation in logic programming*, Journal of Logic Programming 11, pp. 217-242, 1991.

[N] L. NAISH, *Automatic generation of control for logic programs*, Technical Report TR83/6, Department of Computer Science, University of Melbourne, 1983.

[P] L. PLUMER, *Termination proofs for logic programs*, LNAI 446, Springer-Verlag, Berlin, 1990.

[S] J.C. SHEPHERDSON, *Undecidability of Horn clause logic and pure Prolog*, unpublished manuscript, 1985.

[S1] J.C. SHEPHERDSON, H.E. STURGIS, *Computability of recursive functions*, Journal of the ACM 10, pp. 217-255, 1963.

[S2] J.C. SHEPHERDSON, *Machine configuration and word problems of given degree of unsolvability*, Zeitsch. f. math. Logik und Grundlagen d. Math. 11, pp. 149-175, 1965.

[Sh] J.R. SHOENFIELD, *Mathematical Logic*, Addison-Wesley, Reading, Mass., 1967.

[SVB] D. DE SCHREYE, K. VERSCHAETSE, M. BRUYNOOGHE, *A framework for analysing the termination of definite logic programs with respect to call patterns*, to appear in: Proceedings of the international conference on fifth generation computer systems, Tokyo, 1992.

[UvG] J.D. ULLMAN, A. VAN GELDER, *Efficient tests for top-down termination of logical rules*, Journal of the ACM 35, pp. 345-373, 1988.

[VS] K. VERSCHAETSE, D. DE SCHREYE, *Deriving termination proofs for logic programs using abstract procedures*, in: K. FURUKAWA, (editor), Proceedings of the international conference on logic programming, Paris, pp. 301-315, MIT Press, 1991.