A completeness result for SLDNF–resolution*

Karl Stroetmann†

January 4, 2003

Abstract

Because of the possibility of floundering and infinite derivations, SLDNF–resolution is in general not complete. The classical approach [LlTo86] to get a completeness result is to restrict the attention to normal programs P and normal goals G, such that $P \cup \{G\}$ is allowed and P is hierarchical. Unfortunately, the class of all normal programs and all normal goals meeting these requirements is not powerful enough to be of great practical importance. But after refining the concept of allowedness by taking *modes* [HeAv85] into account, we can broaden the notion of a hierarchical program and thereby define a subclass of the class of normal programs and normal goals which is powerful enough to compute all primitive recursive functions without losing the completeness of SLDNF–resolution.

1 Introduction

The two main obstacles which have to be faced in order to get a completeness result for SLDNF–resolution are:

- Floundering
- Infinite derivations

Today the problem of floundering is well understood. The simplest possibility to exclude floundering computations is to restrict the attention to *allowed* queries, which were introduced by Clark [Cla78]. However, an allowed program may contain only closed facts. Therefore these programs are in general not sufficient. The concept of allowed programs can be refined by taking *modes* [He84, HeAv85, DeMa85] into account. Modes are a way to divide the variables in a clause into input and output variables. Thereby the problem of floundering can be controlled. Moreover, modes permit a refinement of the concept of a hierarchical program, which no longer bans recursion completely, but still guarantees the finiteness of all SLDNF–derivations.

If P is a **normal** logic program and G is a goal, the set $P \cup \{G\}$ will be called a *queried program*.

All notions not defined in this paper can be looked up in the book of J. W. Lloyd [Llo87]. There is however one major difference from Lloyd: we fix the computation rule to select always the first literal in a goal. This selection rule is actually the one applied in Prolog and it therefore deserves our special attention. Because the "S" in SLDNF stands for "selection rule", we will drop it as suggested in [ApPe90]. Hence in future we will speak of LDNF–derivation, LDNF–refutation and LDNF–resolution instead of SLDNF–derivation, SLDNF–refutation

*This paper is part of the author's Ph.D. dissertation [Str91], which has been written under the supervision of Prof. Dr. J. Diller at the University of Münster.

†*Address correspondence to* Karl Stroetmann, Siemens AG, ZFE BT SE 11, Otto–Hahn–Ring 6, D–8000 München 83, Germany. E–Mail: karl@inf21.zfe.siemens.de.

and SLDNF–resolution. The results of this paper will show that, for the class of normal programs under consideration, the Prolog selection rule is safe. It will trivially turn out to be fair, because there will be no infinite LDNF–derivations.

Throughout this paper, P will stand for a normal program and G for a normal goal. Σ denotes the set of all predicate symbols occurring in P, while Λ denotes the set of all function symbols used in P. If q is an n–ary predicate symbol from P, then $arity(q) = n$. If E is an expression, $FV(E)$ denotes the multiset of free variables in E. So if x is a variable that occurs n times in E, then it will occur n times in the multiset $FV(E)$.

The paper is organized as follows. In Section 2 we introduce the concept of an I/O–specification and show that an LDNF–computation for $P \cup \{G\}$ will not flounder if there exists an I/O–specification σ such that $P \cup \{G\}$ is stable. In Section 3 we develop the notion of a program which is ordered for an I/O–specification σ and show that for a queried program $P \cup \{G\}$ and an I/O–specification σ such that $P \cup \{G\}$ is stable and furthermore P is ordered, no infinite LDNF–derivations exist. In Section 4 we collect our results and prove the completeness of LDNF–resolution for a stable queried program $P \cup \{G\}$ such that P is ordered. In Section 5 we show that this class of queried programs is sufficient for the computation of all primitive recursive functions. Finally, in Section 6 we will compare our results with some other methods proposed in the literature.

2 How to avoid floundering

In this section we define the notion of a stable program P and we give some usefull properties of stable programs. None of the concepts and lemmata given in this section is new, they can all be found either in [He84] or [DeMa85]. Our reason for repeating all the definitions and lemmata is to make this article independent of the aforementioned work.

Everything starts from the notion of an I/O–specification. An I/O–specification σ is a function assigning to every predicate symbol p a *mode* σ_p. A mode divides the occurrences of variables appearing in the head of a clause into input and output variables. Formally, a mode σ_p for an n–ary predicate symbol p is a function from the set $\{1, \cdots, n\}$ to the set $\{+, -\}$. We write σ_p in a more suggestive form as

$$p(\sigma_p(1), \cdots, \sigma_p(n)).$$

Modes indicate how the arguments of a predicate should be used. If $\sigma_p(i) = \text{“}+\text{”}$, we call i an *input position* of p and if $\sigma_p(i) = \text{“}-\text{”}$, we call i an *output position* of p (both w.r.t. σ_p). The intention is that the input positions should be replaced by ground terms.

The notion of an I/O–specification makes it possible to define input and output variables.

Definition 1 (Input variables and output variables) *Let $P \cup \{G\}$ be a queried program, σ an I/O–specification for $P \cup \{G\}$, and q an n–ary predicate symbol from P.*

We define the multiset $FV^+\big(q(t_1, \cdots, t_n)\big)$ of input variables *of the atom $q(t_1, \cdots, t_n)$ as*

$$FV^+\big(q(t_1, \cdots, t_n)\big) \quad := \quad \biguplus\{FV(t_i) \colon\ i \in \{1, \cdots, n\}\ \&\ \sigma_q(i) = \text{“}+\text{”}\}.$$

(Here $\biguplus$ stands for the union of multisets.) The multiset $FV^-\big(q(t_1, \cdots, t_n)\big)$ of output variables *of the atom $q(t_1, \cdots, t_n)$ is defined as*

$$FV^-\big(q(t_1, \cdots, t_n)\big) \quad := \quad \biguplus\{FV(t_i) \colon\ i \in \{1, \cdots, n\}\ \&\ \sigma_q(i) = \text{“}-\text{”}\}.$$

For a negative literal $\neg Q$ *we set*

$$FV^{+}(\neg Q) \ := \ FV(Q) \qquad \textit{and} \qquad FV^{-}(\neg Q) \ := \ \emptyset.$$

We are now able to define the notion of *stableness* for an I/O–specification σ. This notion is central for everything that follows.

Definition 2 (Stable)
Let $P \cup \{G\}$ *be a queried program and* σ *an I/O–specification for* $P \cup \{G\}$.
We call a clause $Q \leftarrow L_1, \cdots, L_n$ *from* P stable *(w.r.t.* σ*) iff*

1. $x \in FV^{-}(Q) \ \Rightarrow \ x \in FV^{+}(Q) \ \textit{ or } \ \exists j \in \{1, \cdots, n\}\big[x \in FV(L_j)\big] \quad \textit{and}$
2. $x \in FV^{+}(L_i) \ \Rightarrow \ x \in FV^{+}(Q) \ \textit{ or } \ \exists j \in \{1, \cdots, i-1\}\big[x \in FV(L_j)\big].$

A normal goal G *of the form* $\leftarrow L_1, \cdots, L_n$ *is* stable *(w.r.t.* σ*) iff*

$$x \in FV^{+}(L_i) \ \Rightarrow \ \exists j \in \{1, \cdots, i-1\}\big[x \in FV(L_j)\big].$$

Finally, the queried program $P \cup \{G\}$ *is called* stable *(w.r.t.* σ*) iff every clause from* $P \cup \{G\}$ *is stable.*

(In the following we will always assume that the I/O–specification σ is clear from the context and therefore we will drop the clause "w.r.t. σ".)
Observation: There are two important observations following immediately from the definition of stable goals and clauses.

1. If $G \equiv \ \leftarrow L_1, \cdots, L_m$ is a normal goal such that $P \cup \{G\}$ is stable, then $FV^{+}(L_1) = \emptyset$. If L_1 is a negative literal this implies that $FV(L_1) = \emptyset$.
2. If C is a normal clause, which is stable, then every instance of C is again stable. Of course, the same is true for a normal goal.

These two observations will be used repeatedly in the proof of the next proposition.

Before we are able to prove that no LDNF–computation of $P \cup \{G\}$ flounders if $P \cup \{G\}$ is stable, we have to formalize the notion of an LDNF–*computation*, which is essentially due to Clark [Cla78].

Definition 3 (LDNF–computation, LDNF–refutation, computed answer)

Let $P \cup \{G\}$ *be a queried program.*
An LDNF–computation of $P \cup \{G\}$ *consists of a finite sequence* $G_0, G_1, \cdots, G_n$ *of normal goals (where* $G_0 = G$*), a sequence* $C_1, C_2, \cdots, C_n$ *of variants*[1] *of clauses of* P *or ground negative literals, and a sequence* $\vartheta_1, \vartheta_2, \cdots, \vartheta_n$ *of substitutions such that we have for all* $i \in \{0, \cdots, n-1\}$ *either*

(a) G_{i+1} *is derived from* G_i *and an input clause* C_{i+1} *using* ϑ_{i+1} *or*

(b) G_i *is* $\leftarrow \neg Q, L_1, \cdots, L_m$*, the literal* $\neg Q$ *is ground, and there is a finitely failed LDNF–tree for* $P \cup \{\leftarrow Q\}$*. In this case* G_{i+1} *is* $\leftarrow L_1, \cdots, L_m$*, the substitution* ϑ_{i+1} *is the identity substitution, and* C_{i+1} *is* $\neg Q$.

The LDNF–computation is an LDNF–refutation *iff the last goal* G_n *is the empty goal* $\Box$*. In this case the substitution* $\vartheta_1 \cdots \vartheta_n|_{FV(G)}$ *is called a computed answer for* $P \cup \{G\}$.

[1] These variants are such that they satisfy the conditions $FV(G_i) \cap FV(C_{i+1}) = \emptyset$ and $FV(C_i) \cap FV(C_j) = \emptyset$ for $i \neq j$. This can all be achieved most easily by requiring that the variables in C_i are *new*, i.e. that they have not appeared anywhere else before.

Observe that the definition of an LDNF–computation given above is just the definition of an SLDNF–derivation of [Llo87] without the last clause referring to the case that the sequence $G_0, G_1, \cdots$ is finite.[2] Instead, an LDNF–computation is always finite.

Proposition 4
Let $P \cup \{G\}$ be a queried program and σ an I/O–specification for $P \cup \{G\}$.
Then if $P \cup \{G\}$ is stable, then we have the following:

1. *No computation of $P \cup \{G\}$ flounders.*
2. *If ϑ is a computed answer for $P \cup \{G\}$, then $\vartheta(x)$ is ground for all $x \in FV(G)$.*

Proof: Take an arbitrary LDNF–computation of $P \cup \{G\}$. It consists of a sequence $G_0, G_1, \cdots, G_n$ of normal goals (where $G_0 = G$), a sequence $C_1, C_2, \cdots, C_n$ of variants of program clauses of P, and a sequence $\vartheta_1, \vartheta_2, \cdots, \vartheta_n$ of substitutions. We will show by induction on n that for every natural number n the goal G_n is stable. According to our observation following the definition of stable goals the first result then follows. We proceed to prove the above stated claim by induction. The case $n = 0$ is trivial and so we look at G_{n+1}. By the induction hypothesis G_n is stable. The goal G_n will start either with a positive or with a negative literal. We distinguish these two cases.

Case 1: $G_n \equiv\ \leftarrow Q, L_1, \cdots, L_k$, so the first literal in G_n is positive.
Then C_{n+1} has the form $R \leftarrow K_1, \cdots, K_m$ and $\vartheta_{n+1} = \mathrm{mgu}(Q, R)$. This implies

$$G_{n+1} \equiv\ \leftarrow (K_1, \cdots, K_m,\ L_1, \cdots, L_k)\vartheta_{n+1}.$$

In order to establish that G_{n+1} is stable we have to prove two claims:

$$\begin{array}{lll} \text{Claim 1:} & x \in FV^+(K_i\vartheta_{n+1}) \;\Rightarrow & \exists j \in \{1, \cdots, i-1\}\big[x \in FV(K_j\vartheta_{n+1})\big]. \\ \text{Claim 2:} & x \in FV^+(L_i\vartheta_{n+1}) \;\Rightarrow & \exists j \in \{1, \cdots, m\}\big[x \in FV(K_j\vartheta_{n+1})\big] \quad \text{or} \\ & & \exists j \in \{1, \cdots, i-1\}\big[x \in FV(L_j\vartheta_{n+1})\big]. \end{array}$$

We start with the first claim. If $x \in FV^+(K_i\vartheta_{n+1})$ there has to be a variable $y \in FV^+(K_i)$ such that $x \in FV\big(\vartheta_{n+1}(y)\big)$. Because C_{n+1} is stable we have

$$(*) \qquad y \in FV^+(R) \quad \text{or} \quad \exists j \in \{1, \cdots, i-1\}\big[y \in FV(K_j)\big].$$

The possibility $y \in FV^+(R)$ can be excluded, because from

$$y \in FV^+(R) \quad \text{and} \quad x \in FV\big(\vartheta_{n+1}(y)\big)$$

it follows that $x \in FV^+(R\vartheta_{n+1})$, which implies $x \in FV^+(Q\vartheta_{n+1})$, since $Q\vartheta_{n+1} \equiv R\vartheta_{n+1}$. But the fact that G_n is stable shows that $FV^+(Q) = \emptyset$, which entails $FV^+(Q\vartheta_{n+1}) = \emptyset$. This contradicts $x \in FV^+(Q\vartheta_{n+1})$ and therefore $(*)$ yields

$$\exists j \in \{1, \cdots, i-1\}\big[y \in FV(K_j)\big].$$

From $y \in FV(K_j)$ we see that $x \in FV(K_j\vartheta_{n+1})$ so that we have shown the first claim.

To establish the second claim we suppose $x \in FV^+(L_i\vartheta_{n+1})$. This time there must be a variable $y \in FV^+(L_i)$ such that $x \in FV\big(\vartheta_{n+1}(y)\big)$. Because $G_n \equiv\ \leftarrow Q, L_1, \cdots, L_k$ is stable we have one of the following cases:

(a) $y \in FV(Q)$;

[2]Another difference is, of course, the fact that we allow only the Prolog selection rule.

(b) $\exists j \in \{1, \cdots, i-1\} \big[y \in FV(L_j) \big]$.

Case (a): $y \in FV(Q)$

Because $FV^+(Q) = \emptyset$ we must have $y \in FV^-(Q)$. This means $x \in FV^-(Q\vartheta_{n+1})$ and then also $x \in FV^-(R\vartheta_{n+1})$. Therefore we have a variable $z \in FV^-(R)$ with $x \in FV\big(\vartheta_{n+1}(z)\big)$. Because C_{n+1} is stable we have

$$z \in FV^+(R) \quad \text{or} \quad \exists j \in \{1, \cdots, m\} \big[z \in FV(K_j) \big].$$

Due to the fact that $Q\vartheta_{n+1} \equiv R\vartheta_{n+1}$, $FV^+(Q) = \emptyset$ and $x \in FV\big(\vartheta_{n+1}(z)\big)$ hold, the assumption $z \in FV^+(R)$ leads to a contradiction. So it is true that $z \in FV(K_j)$ for a $j \in \{1, \cdots, m\}$, from which we can conclude $x \in FV(K_j \vartheta_{n+1})$, which settles the claim in this case.

Case (b): $\exists j \in \{1, \cdots, i-1\} \big[y \in FV(L_j) \big]$

Because of $x \in FV\big(\vartheta_{n+1}(y)\big)$ we can conclude $x \in FV(L_j \vartheta_{n+1})$ for a $j < i$ and so we are done.

Case 2: $G_n \equiv\ \leftarrow \neg Q, L_1, \cdots, L_k$.

In this case we know $G_{n+1} \equiv\ \leftarrow L_1, \cdots, L_k$. If $x \in FV^+(L_i)$ for an $i \in \{1, \cdots, n\}$, the fact that G_n is stable guarantees that

$$x \in FV(\neg Q) \quad \text{or} \quad \exists j \in \{1, \cdots, i-1\} \big[x \in FV(L_j) \big]$$

holds. Since $FV(\neg Q) = FV^+(\neg Q) = \emptyset$ there must be a $j \in \{1, \cdots, i-1\}$ with $x \in FV(L_j)$, which finishes the first part of the proof.

To prove that every computed answer is a ground substitution we take an arbitrary LDNF–refutation of $P \cup \{G\}$. It consists of a sequence $G_0, G_1, \cdots, G_n$ of normal goals (where $G_0 = G$ and $G_n = \Box$), a sequence $C_1, C_2, \cdots, C_n$ of variants of program clauses of P, and a sequence $\vartheta_1, \vartheta_2, \cdots, \vartheta_n$ of substitutions. The computed answer is then $\vartheta_1 \cdots \vartheta_n |_{FV(G)}$. If $\{x_1, \cdots, x_k\}$ is the set of variables occurring in G, we take a new k–ary predicate symbol r and define a goal G' by appending the literal $\neg r(x_1, \cdots, x_k)$ at the end of G. More formally, if $G \equiv\ \leftarrow L_1, \cdots, L_m$, then G' is defined as $\leftarrow L_1, \cdots, L_m, \neg r(x_1, ..., x_k)$. For $i \in \{0, \cdots, n\}$ the goals G'_i are defined analogously by appending the literal $\neg r(x_1, \cdots, x_k)$ to the end of G_i. Now it is quite obvious that the sequence $G'_0, G'_1, \cdots, G'_n$ of normal goals together with the sequence $C_1, C_2, \cdots, C_n$ of clauses and the sequence $\vartheta_1, \vartheta_2, \cdots, \vartheta_n$ of substitutions is an LDNF–computation of $P \cup \{G'\}$. The last goal of this sequence has the form

$$G'_n \equiv\ \leftarrow \neg r\big(\vartheta(x_1), \cdots, \vartheta(x_k)\big).$$

The definition of stableness shows immediately that $P \cup \{G'\}$ is stable. Therefore we know already that this LDNF–computation does not flounder. Since the literal $\neg r\big(\vartheta(x_1), \cdots, \vartheta(x_k)\big)$ is negative, it has to be ground. But this implies that $\vartheta(x)$ is ground for all $x \in FV(G)$. $\Box$

3 How to avoid infinite derivations

In this section we tackle the problem of avoiding infinite derivations. Classically this can be done by requiring P to be *hierarchical* [Cla78] and thus banning any form of recursion. However, if we restrict our attention to stable queried programs $P \cup \{G\}$, we can introduce the notion of an ordered program, which lies between the hierarchical and the *stratified* [ABW87] programs. We will be able to show that for stable queried programs $P \cup \{G\}$ such that the program P is ordered all

LDNF–derivations are finite. The usefulness of ordered programs stems from the fact that they allow for a certain controlled form of recursion, which is sufficient for many applications.

Throughout this section we fix a queried program $P \cup \{G\}$ and an I/O–specification σ and assume that $P \cup \{G\}$ is stable. In order to be able to introduce *ordered programs* we have to fix a mapping $\underline{\text{level}}$. This function divides the predicate symbols from Σ into disjoint classes by assigning to each predicate symbol a natural number:

$$\underline{\text{level}} : \Sigma \to \mathbf{N}.$$

The function $\underline{\text{level}}$ is extended to literals in the obvious way. If L is a literal with predicate symbol p then $\underline{\text{level}}(L) := \underline{\text{level}}(p)$.

Before we are able to give the definition of an ordered program we have to introduce the concept of a *simplification ordering* [Der82].

Definition 5 (Simplification ordering) *A partial ordering $\prec$ on the set $\mathcal{T}$ of all terms is a* simplification ordering *iff it has the following properties:*

1. $s \prec t$ *implies* $f(\cdots, s, \cdots) \prec f(\cdots, t, \cdots)$.
2. $s \prec f(\cdots, s, \cdots)$.

If we have furthermore for any substitution ϑ

$$s \prec t \quad \Rightarrow \quad s\vartheta \prec t\vartheta,$$

then $\prec$ is called a stable *simplification ordering.*

It is well known [Der82, DeJo90] that every simplification ordering is well–founded. Examples of stable simplification orderings are the *Knuth–Bendix ordering* [KnBe70], the *lexicographic path ordering* [KaLe80], and the *recursive path ordering* proposed by Dershowitz [Der82].

In the following we assume that $\prec$ is a fixed stable simplification ordering. We extend this ordering to the set $\mathbf{N} \times \mathcal{T}$ of all pairs of natural numbers and terms lexicographically:

$$(m, s) \prec_{lex} (n, t) \quad \text{iff} \quad m < n \quad \text{or} \quad (n = m \ \ \& \ \ s \prec t).$$

This extension is easily seen to be stable and well–founded.

To proceed, we assign to every atom $p(t_1, \cdots, t_n)$ a multiset denoted by $\text{mset}\big(p(t_1, \cdots, t_n)\big)$ as follows:

$$\text{mset}\big(p(t_1, \cdots, t_n)\big) \ := \ \big\{(\underline{\text{level}}(p), t_i) : \ \sigma_p(i) = \text{“}+\text{”} \ \& \ i \in \{1, \cdots, n\}\big\}$$

For a negative literal $\neg p(t_1, \cdots, t_n)$ we define $\text{mset}\big(\neg p(t_1, \cdots, t_n)\big) := \text{mset}\big(p(t_1, \cdots, t_n)\big)$.

If $(X, \prec)$ is a partially ordered set, then the ordering $\prec$ can be extended to the set of finite multisets with elements from X. For two finite multisets A and B with elements from X, we define $B \prec_{mset} A$ iff the following two conditions hold:

1. $A \neq B$.
2. $\forall x \in B \, \big(x \in A \ \vee \ \exists y \in A \, (y \notin B \wedge x \prec y)\big)$.

Obviously, the ordering $\prec_{mset}$ is just the usual extension of $\prec$ to finite multisets. It is well known [Der82] that the ordering $\prec_{mset}$ is well–founded if $\prec$ has this property.

We have given the definition above to be able to extend the ordering $\prec_{lex}$ on $\mathbf{N} \times \mathcal{T}$ to the set of finite multisets of elements from $\mathbf{N} \times \mathcal{T}$. This ordering will again be denoted by $\prec_{mset}$. We have already noted that this ordering is well–founded. It is easily seen to be stable[3] under substitutions.

We are now in a position to define the concept of an ordered program.

Definition 6 (Ordered programs)
Let P be a normal program, σ an I/O–specification for P, $\underline{\text{level}}$ a function assigning a level to every predicate symbol in P and $\prec$ a stable simplification ordering on the set $\mathcal{T}$ of all terms.

Then the normal program P is ordered *(via the I/O–specification σ, the function $\underline{\text{level}}$ and the ordering $\prec$) iff for every clause from P of the form*

$$Q \leftarrow L_1, \cdots, L_n$$

the following holds:

1. *If the literal L_i is negative for $i \in \{1, \cdots, n\}$, then $\underline{\text{level}}(L_i) < \underline{\text{level}}(Q)$.*
2. *The inequation $\text{mset}(L_i) \prec_{mset} \text{mset}(Q)$ is valid for all $i \in \{1, \cdots, n\}$.*
3. *If we define the set I as $I := \{i \in \{1, \cdots, n\} \colon \underline{\text{level}}(L_i) = \underline{\text{level}}(Q)\}$, then*

$$FV^+(L_i) \subseteq_{multi} FV^+(Q) \qquad \textit{holds for all } \; i \in I.$$

Here $\subseteq_{multi}$ stands for multiset inclusion.

Observation: If $Q \leftarrow L_1, \cdots, L_n$ is a clause from a program P which is ordered, then we have

$$\text{mset}(L_i\vartheta) \prec_{mset} \text{mset}(Q\vartheta) \quad \text{for all} \;\; i \in \{1, \cdots, n\}$$

for every substitution ϑ. This is due to the stability of $\prec_{mset}$ under substitutions. Due to the first and second condition every ordered program is stratified[4]. Therefore if P is an ordered program, then $comp(P)$ is consistent [ABW87].

Before we are able to prove that there are no infinite LDNF–computations for a queried program which is ordered and stable we have to introduce a variant of the function mset. We define a function $\underline{\text{mset}}$ as

$$\underline{\text{mset}}(L) \;:=\; \begin{cases} \text{mset}(L) & \text{if} \;\; FV^+(L) = \emptyset; \\ \{(\underline{\text{level}}(L), \top)\} & \text{otherwise.} \end{cases}$$

Here $\top$ is a symbol satisfying $t \prec \top$ for any term t.

The function $\underline{\text{mset}}$ has the following important property: for every literal L and every substitution ϑ we have

$$\underline{\text{mset}}(L\vartheta) \preceq_{mset} \underline{\text{mset}}(L).$$

The reason for this is that either $FV^+(L)$ is empty whence $\underline{\text{mset}}(L)$ and $\underline{\text{mset}}(L\vartheta)$ are the same or $FV^+(L)$ is not empty. But then $\underline{\text{mset}}(L)$ is equal to $\{(\underline{\text{level}}(L), \top)\}$ and this is at least as big as $\underline{\text{mset}}(L\vartheta)$.

[3]This is true because we are working with multisets. If we had used sets instead of multisets, then this property would fail since a set might be collapsed by a substitution because different elements of a set can be unified.

[4]The definition of an ordered program can be slightly generalized by dropping the first clause. Then it is no longer true that every ordered program is stratified. Therefore in that case later on one has to prove independently that for every ordered program P the completion $comp(P)$ is consistent.

Lemma 7 *Let P be a normal program which is stable and ordered.*

Then if $R \leftarrow K_1, \cdots, K_m$ is a clause from P and ϑ is a substitution such that $FV^+(R\vartheta) = \emptyset$, then we have

$$\underline{\text{mset}}(K_i\vartheta) \prec_{mset} \underline{\text{mset}}(R\vartheta) \quad \textit{for all } i \in \{1, \cdots, n\}.$$

Proof: Since P is ordered we have

$$(1) \qquad \text{mset}(K_i) \prec_{mset} \text{mset}(R) \quad \text{for all } i \in \{1, \cdots, n\} \qquad \text{and}$$

$$(2) \qquad FV^+(K_i) \subseteq_{multi} FV^+(R) \qquad \text{holds for all } i \in I,$$

where the set I is defined as $I := \{i \in \{1, \cdots, m\} : \underline{\text{level}}(K_i) = \underline{\text{level}}(R)\}$. Since $\prec_{mset}$ is stable we conclude from (1)

$$\text{mset}(K_i\vartheta) \prec_{mset} \text{mset}(R\vartheta) \quad \text{for all } i \in \{1, \cdots, n\}.$$

This implies of course

$$(3) \qquad \text{mset}(K_i\vartheta) \prec_{mset} \text{mset}(R\vartheta) \quad \text{for all } i \in I.$$

As $FV^+(R\vartheta) = \emptyset$, the functions mset and $\underline{\text{mset}}$ agree on $R\vartheta$. From (2) it follows that $FV^+(K_i\vartheta)$ is empty for all $i \in I$, so that mset and $\underline{\text{mset}}$ agree on $K_i\vartheta$ for $i \in I$. Therefore (3) can be written as

$$\underline{\text{mset}}(K_i\vartheta) \prec_{mset} \underline{\text{mset}}(R\vartheta) \quad \text{for all } i \in I.$$

To proceed, we just have to notice that for all $i \in \{1, \cdots, n\} - I$ the elements of $\underline{\text{mset}}(R\vartheta)$ have the form (n, t) where $n = \underline{\text{level}}(R)$ and t is a term, while the elements of $\underline{\text{mset}}(K_i\vartheta)$ have the form (m, s) where $m < n$ and s is either a term or the symbol $\top$. Therefore it is clear that

$$\underline{\text{mset}}(K_i\vartheta) \prec_{mset} \underline{\text{mset}}(R\vartheta) \quad \text{for all } i \in \{1, \cdots, n\} - I$$

holds and together with (3) this establishes our claim. □

The next proposition establishes the central property of ordered programs.

Proposition 8
Let P be a normal program, σ an I/O–specification for P, $\underline{\text{level}}$ a function assigning a level to every predicate symbol in P and $\prec$ a stable simplification ordering on the set $\mathcal{T}$ of all terms.

Then if $P \cup \{G\}$ is stable and P is ordered, then there is no infinite LDNF–derivation of $P \cup \{G\}$.

Proof: The proof is indirect and starts from the assumption that there is an infinite LDNF–derivation which consists of a sequence $\langle H_n : n \in \mathbf{N} \rangle$ of normal goals (where $H_0 = G$), a sequence $\langle C_n : n \in \mathbf{N} - \{0\} \rangle$ of variants of program clauses of P, and a sequence $\langle \vartheta_n : n \in \mathbf{N} - \{0\} \rangle$ of substitutions. The idea is to assign a multiset $o(H_n)$ consisting of finite multisets of pairs from $\mathbf{N} \times \mathcal{T}$ to every goal H_n such that for all $n \in \mathbf{N}$

$$(1) \qquad o(H_{n+1}) \lll_{mset} o(H_n),$$

where $\lll_{mset}$ is the usual extension of $\prec_{mset}$ to multisets of multisets. Then $\langle o(H_n) : n \in \mathbf{N} \rangle$ would be an infinite decreasing sequence of multisets, which is impossible since $\prec_{mset}$ and therefore also $\lll_{mset}$ is well–founded. The definition of the function o is as follows:

$$o\big(\leftarrow L_1, \cdots, L_n\big) \ := \ \big\{\underline{\text{mset}}(L_1), \cdots, \underline{\text{mset}}(L_n)\big\}.$$

Here $\big\{\underline{\text{mset}}(L_1), \cdots, \underline{\text{mset}}(L_n)\big\}$ is regarded as multiset.

Now we can begin with the proof of (1). As in the proof of Proposition 4, our further considerations will depend on whether the first literal is positive or negative. We perform a case distinction accordingly.

Case 1: $H_n \equiv \leftarrow Q, L_1, \cdots, L_r$, so the first literal in H_n is positive.

Then C_{n+1} takes the form $R \leftarrow K_1, \cdots, K_m$ and $\vartheta_{n+1} = \text{mgu}(Q, R)$. This implies that

$$H_{n+1} \ \equiv \ \leftarrow (K_1, \cdots, K_m, \ L_1, \cdots, L_r)\vartheta_{n+1}.$$

After the definition of $\underline{\text{mset}}$ we have already noted that for every literal L and every substitution ϑ the inequality $\underline{\text{mset}}(L\vartheta) \preceq_{mset} \underline{\text{mset}}(L)$ holds. Obviously, this property carries over to the function o: if G is a goal and ϑ is any substitution then we have

$$o(G\vartheta) \ \underline{\lll}_{mset} \ o(G).$$

Therefore we have

$$\begin{array}{rcl} o(H_n) & \underline{\ggg}_{mset} & o(H_n\vartheta_{n+1}) \\ & = & o(\leftarrow Q\vartheta_{n+1}, L_1\vartheta_{n+1}, \cdots, L_r\vartheta_{n+1}) \\ & = & \big\{\underline{\text{mset}}(Q\vartheta_{n+1}), \ \underline{\text{mset}}(L_1\vartheta_{n+1}), \ \cdots, \ \underline{\text{mset}}(L_r\vartheta_{n+1})\big\} \end{array}$$

Since $o(H_{n+1}) \ = \ \big\{\underline{\text{mset}}(K_1\vartheta_{n+1}), \cdots, \underline{\text{mset}}(K_m\vartheta_{n+1}), \ \underline{\text{mset}}(L_1\vartheta_{n+1}), \cdots, \underline{\text{mset}}(L_r\vartheta_{n+1})\big\}$, it is sufficient to show that

(3) $\qquad \underline{\text{mset}}(K_i\vartheta_{n+1}) \prec_{mset} \underline{\text{mset}}(Q\vartheta_{n+1}) \qquad$ for all $i \in \{1, \cdots, m\}$

holds in order to show $o(H_{n+1}) \lll_{mset} o(H_n)$.

To establish (3) we note that since $P \cup H_n$ is stable we have $FV^+(Q) = \emptyset$. This implies trivially $FV^+(Q\vartheta_{n+1}) = \emptyset$ and since ϑ_{n+1} is a unifier of R and Q also $FV^+(R\vartheta_{n+1}) = \emptyset$. Now we can apply Lemma 7 to conclude

$$\underline{\text{mset}}(K_i\vartheta_{n+1}) \prec_{mset} \underline{\text{mset}}(R\vartheta_{n+1}) \quad \text{for all} \ \ i \in \{1, \cdots, n\}.$$

But this is just (3), since ϑ_{n+1} is a unifier of Q and R, finishing the proof in this case.

Case 2: $H_n \ \equiv \ \leftarrow \neg Q, L_1, \cdots, L_m$.

Then we have $H_{n+1} \ \equiv \leftarrow L_1, \cdots, L_m$. This implies

$$o(H_n) \ = \ \big\{\underline{\text{mset}}(\neg Q)\big\} \ \uplus \ o(H_{n+1}),$$

which entails (1). $\square$

In order to apply our abstract considerations we now have to find a suitable stable simplification ordering $\prec$ on the set of all terms. To this end we define a function $\underline{\text{weight}}$ from the set Λ of all function symbols to the set of all positive integers:

$$\underline{\text{weight}} : \Lambda \to \mathbf{N} - \{0\}.$$

The function $\underline{\text{weight}}$ enables us to define the length $\text{ln}(t)$ of a term t inductively:

$$\begin{aligned}
\ln(t) &:= 0 && \text{if } t \text{ is a variable;}\\
\ln(t) &:= \underline{\text{weight}}(f) + \sum_{i=1}^{n} \ln(t_i) && \text{if } t = f(t_1, \cdots, t_n).
\end{aligned}$$

Now we can define the order relation $\prec$ as

$$s \prec t \quad \text{iff} \quad \ln(s) < \ln(t) \;\;\&\;\; FV(s) \subseteq_{multi} FV(t).$$

This ordering is in fact a simplified version of the Knuth–Bendix ordering [KnBe70]. One can easily verify that $\prec$ is a simplification ordering. To see that $\prec$ is stable we need the following lemma.

Lemma 9 (Stability of $\prec$) *If*

$$\ln(s) < \ln(t) \qquad \textit{and} \qquad FV(s) \subseteq_{multi} FV(t)$$

holds, then we have for every substitution τ

$$\ln(s\tau) < \ln(t\tau) \qquad \textit{and} \qquad FV(s\tau) \subseteq_{multi} FV(t\tau).$$

Proof: To prove this lemma one first shows by an easy induction on the structure of t that for any term t the equation

$$\ln(s\tau) \;=\; \ln(s) + \sum_{x \in FV(s)} \ln\bigl(\tau(x)\bigr)$$

is valid, provided that $FV(s)$ is interpreted as a multiset. This shows that $\ln(s\tau) < \ln(t\tau)$ holds under the assumptions $\ln(s) < \ln(t)$ and $FV(s) \subseteq_{multi} FV(t)$.

Another induction on the structure of t shows that for any term t

$$FV(s\tau) \;=\; \biguplus_{x \in FV(s)} FV\bigl(\tau(x)\bigr)$$

is valid. This shows that $FV(s\tau) \subseteq_{multi} FV(t\tau)$ holds under the assumption $FV(s) \subseteq_{multi} FV(t)$. Therefore $\prec$ is stable. □

In the following we will always assume that the ordering $\prec$ is defined as above by assigning a weight to every function symbol.

We conclude this Section with a definition.

Definition 10 (Consistent queried programs)
Let $P \cup \{G\}$ be a queried program, σ an I/O–specification for P and $\prec$ a stable simplification ordering.

Then $P \cup \{G\}$ is consistent (w.r.t. σ and $\prec$) iff

1. *$P \cup \{G\}$ is stable and*
2. *P is ordered.*

In order to have a handy notation we will always assume in the future that σ and $\prec$ are either clear from the context or else unimportant when we speak of queried program $P \cup \{G\}$ which is consistent w.r.t. an I/O–specification σ and an ordering $\prec$. This enables us to speak just of a consistent program.

4 Completeness results for LDNF–resolution

Let us assume that we have a queried program $P \cup \{G\}$, an I/O–specification σ for P and an ordering $\prec$ such that $P \cup \{G\}$ is consistent. Then Propositions 4 and 8 show that the LDNF–tree for the queried program $P \cup \{G\}$ exists and is finite. Either it is failed or not. Due to its finiteness we can check this effectively. Therefore we can actually decide whether there is an LDNF–refutation of $P \cup \{G\}$. This gives us our first completeness result.

Theorem 11 (Syntactical completeness of LDNF–resolution)
Let P be a normal program, Q a closed atom, σ an I/O–specification for P and $\prec$ a stable simplification ordering.

Then if $P \cup \{\leftarrow Q\}$ is a consistent queried program, then there exists either an LDNF–refutation of $P \cup \{\leftarrow Q\}$ or an LDNF–refutation of $P \cup \{\leftarrow \neg Q\}$.

The syntactical completeness of LDNF–resolution is one of the main ingredients of the proof of our final completeness theorem. But it is not sufficient; we also need a lifting lemma for LDNF–resolution. We will give that lemma next.

Lemma 12 (Lifting lemma)
Let $P \cup \{G\}$ be a queried program, σ an I/O–specification for P, and τ a substitution.

If $P \cup \{G\}$ is stable and there is an LDNF–computation of $P \cup \{G\tau\}$ consisting of a sequence

$$G_0, G_1, \cdots, G_n \qquad (\textit{where } G_0 = G\tau)$$

of normal goals, a sequence $C_1, C_2, \cdots, C_n$ of variants of program clauses of P or ground negative literals, and a sequence $\vartheta_1, \vartheta_2, \cdots, \vartheta_n$ of substitutions, then there exists a substitution ϱ and an LDNF–computation of $P \cup \{G\}$ of the same length consisting of a sequence $H_0, H_1, \cdots, H_n$ of normal goals (where $H_0 = G$), the same sequence $C_1, C_2, \cdots, C_n$, and a sequence $\varphi_1, \varphi_2, \cdots, \varphi_n$ of substitutions such that

$$\tau\vartheta_1 \cdots \vartheta_n = \varphi_1\varphi_2 \cdots \varphi_n\varrho \qquad \textit{and} \qquad G_n \equiv H_n\varrho.$$

Proof: We prove the claim by induction on n. If $n = 0$ we can take $\varrho := \tau$ and the claim follows. In the induction step we may assume that the claim is true for all LDNF–computations of length less than $n+1$. So we already have a sequence $H_0, H_1, \cdots, H_n$ of normal goals with $H_0 = G$ and a sequence $\varphi_1, \varphi_2, \cdots, \varphi_n$ of substitutions satisfying the requirements for an LDNF–computation. We make a case distinction according to the form of the first literal in G_n.

Case 1: $G_n \equiv \leftarrow Q, L_1, \cdots, L_k$, so the first literal in G_n is positive.
This implies that G_{n+1} has been derived from G_n and the input clause C_{n+1} using ϑ_{n+1}. Therefore C_{n+1} has the form

$$C_{n+1} \equiv \big(R \leftarrow K_1, \cdots, K_m\big)$$

and $\vartheta_{n+1} = \mathrm{mgu}(Q, R)$. By the induction hypothesis we know that there is a substitution ν such that

$$\tau\vartheta_1 \cdots \vartheta_n = \varphi_1\varphi_2 \cdots \varphi_n\nu \qquad \text{and} \qquad G_n \equiv H_n\nu.$$

Then H_n is necessarily of the form $\leftarrow \hat{Q}, \hat{L}_1, \cdots, \hat{L}_k$, which implies $\hat{Q}\nu = Q$. Since the variables in C_{n+1} are new, they have not appeared in τ, $\vartheta_1, \cdots, \vartheta_n$ or $\varphi_1, \cdots, \varphi_n$. Therefore we may conclude

$$dom(\nu) \cap FV(C_{n+1}) = \emptyset.$$

This implies $R\nu \equiv R$ and therefore we see that $\hat{Q}$ and R are unifiable via the substitution $\nu\vartheta_{n+1}$:

$$\hat{Q}\nu\vartheta_{n+1} \equiv Q\vartheta_{n+1} \equiv R\vartheta_{n+1} \equiv R\nu\vartheta_{n+1}.$$

If we define $\varphi_{n+1} := \mathrm{mgu}(\hat{Q}, R)$ it follows that there is a substitution ϱ such that the identity $\nu\vartheta_{n+1} = \varphi_{n+1}\varrho$ holds. This gives us

$$\tau\vartheta_1 \cdots \vartheta_n\vartheta_{n+1} \stackrel{\text{i.h.}}{=} \varphi_1\varphi_2 \cdots \varphi_n\nu\vartheta_{n+1} = \varphi_1\varphi_2 \cdots \varphi_n\varphi_{n+1}\varrho.$$

If we now derive H_{n+1} from H_n and C_{n+1} using φ_{n+1}, then it is a trivial matter to check $G_{n+1} \equiv H_{n+1}\varrho$, and so we are done.

2. Case: $G_n \equiv \leftarrow \neg Q, L_1, \cdots, L_k$.

Then $\neg Q$ must be ground and there is a finitely failed LDNF–tree for $P \cup \{\leftarrow Q\}$. Furthermore G_{n+1} is $\leftarrow L_1, \cdots, L_k$ and ϑ_{n+1} is the identity substitution. By the induction hypothesis there is a substitution ϱ such that

$$\tau\vartheta_1 \cdots \vartheta_n = \varphi_1\varphi_2 \cdots \varphi_n\varrho \qquad \text{and} \qquad G_n \equiv H_n\varrho.$$

Therefore H_n has the form $\leftarrow \neg\hat{Q}, \hat{L}_1, \cdots, \hat{L}_k$. We already know by Proposition 4 that the LDNF–computation $H_0, \cdots, H_n$ does not flounder, so $\neg\hat{Q}$ must be ground. But we have $\hat{Q}\varrho \equiv Q$ and if $\hat{Q}$ is ground this implies $\hat{Q} \equiv Q$. Therefore we can define

$$H_{n+1} \ := \ \leftarrow \hat{L}_1, \cdots, \hat{L}_k \qquad \text{and} \qquad \varphi_{n+1} := \mathrm{id}.$$

Then the sequence $H_0, \cdots, H_{n+1}$ together with $C_1, \cdots, C_{n+1}$ and $\varphi_1, \cdots, \varphi_{n+1}$ is an LDNF–computation of $P \cup \{G\}$ and we have

$$\tau\vartheta_1 \cdots \vartheta_n\vartheta_{n+1} = \tau\vartheta_1 \cdots \vartheta_n \stackrel{\text{i.h.}}{=} \varphi_1\varphi_2 \cdots \varphi_n\varrho = \varphi_1\varphi_2 \cdots \varphi_n\varphi_{n+1}\varrho$$

as well as $G_{n+1} \equiv H_{n+1}\varrho$, finishing the proof. $\square$

Theorem 13 (Semantical completeness of LDNF–resolution for consistent programs)

Let $P \cup \{G\}$ be a queried program and σ an I/O–specification for P.
Then if

1. *the queried program $P \cup \{G\}$ is consistent and*
2. *the substitution τ is a correct answer for comp$(P) \cup \{G\}$,*

then there exists a computed answer φ such that

$$x\varphi \ \equiv \ x\tau \qquad \textit{for all } x \in FV(G).$$

Proof: Let $\{x_1, \cdots, x_n\}$ be the set of all free variables occurring in $G\tau$ and take G to be the goal $\leftarrow L_1, \cdots, L_m$.

Because τ is a correct answer for *comp*$(P) \cup \{G\}$ it follows that

$$(1) \qquad comp(P) \ \models \ (L_1 \wedge \cdots \wedge L_m)\tau.$$

Choosing new constants $a_1, \cdots, a_n$ we define a substitution

$$\alpha \ := \ [x_1/a_1, \cdots, x_n/a_n].$$

With this definition, (1) implies

$$(2) \qquad comp(P) \ \models \ (L_1 \wedge \cdots \wedge L_m)\tau\alpha.$$

Because $P \cup \{G\}$ is stable, so too is $P \cup \{G\tau\alpha\}$. Therefore we know that the LDNF–tree for $P \cup \{G\tau\alpha\}$ exists and is finite. Now there are two possibilities.

Case 1: The LDNF–tree for $P \cup \{G\tau\alpha\}$ is failed.

We will show that this is impossible due to the soundness of LDNF–resolution. To this end, we add a new 0–ary predicate symbol r to our language and to P we add the clause

$$r \ \leftarrow \ L_1\tau\alpha, \cdots, L_m\tau\alpha.$$

The resulting program will be called P^+. If we look at the LDNF–tree for $P^+ \cup \{\leftarrow r\}$ we see that it consists of the root $\leftarrow r$ and the LDNF–tree for

$P \cup \{G\tau\alpha\}$ planted on top of that root. This shows that the LDNF–tree for $P^+ \cup \{\leftarrow r\}$ is finitely failed. By the soundness of LDNF–resolution it follows that

$$(3) \qquad comp(P^+) \models \neg r$$

holds. Now the relation between $comp(P)$ and $comp(P^+)$ is given by

$$(4) \qquad comp(P^+) = comp(P) \cup \{r \leftrightarrow L_1\tau\alpha \wedge \cdots \wedge L_m\tau\alpha\} \cup \mathrm{EqAx}.$$

(Here EqAx is a set of equality axioms for the new constants.) Therefore (3) implies

$$(5) \qquad comp(P^+) \models \neg(L_1 \wedge \cdots \wedge L_m)\tau\alpha,$$

while (2) gives us

$$(6) \qquad comp(P^+) \models (L_1 \wedge \cdots \wedge L_m)\tau\alpha.$$

It is easily seen that P^+ can be turned into a stratified program simply by assigning a high enough level to r. But then $comp(P^+)$ is consistent, in contradiction to (5) and (6). This shows that the first case is impossible.

Case 2: The LDNF–tree for $P \cup \{G\tau\alpha\}$ contains a success branch.
This success branch can be viewed as an LDNF–refutation of $P \cup \{G\tau\alpha\}$. Because $G\tau\alpha$ is ground, the computed answer of this refutation is the identity substitution. Since the constants $a_1, \cdots, a_n$ do not appear in P nor in $G\tau$, we can replace them with $x_1, \cdots, x_n$. This gives us an LDNF–refutation of $P \cup \{G\tau\}$ with the identity substitution as the computed answer. Let $\vartheta_1, \cdots, \vartheta_k$ be the sequence of substitutions used in this refutation. Because the computed answer is the identity substitution, we know

$$(7) \qquad x\vartheta_1 \cdots \vartheta_k \equiv x \qquad \text{for all } x \in FV(G\tau).$$

Applying the lifting lemma we find an LDNF–refutation of $P \cup \{G\}$ using a sequence $\varphi_1, \cdots, \varphi_k$ of substitutions together with a substitution ϱ such that

$$(8) \qquad \tau\vartheta_1 \cdots \vartheta_k = \varphi_1 \cdots \varphi_k \varrho.$$

The computed answer of this LDNF–refutation is the restriction of $\varphi_1 \cdots \varphi_k$ to $FV(G)$. We denote this substitution by φ:

$$\varphi := \varphi_1 \cdots \varphi_k|_{FV(G)}.$$

Now (7) and (8) together imply

$$(9) \qquad \tau(x) \equiv \varrho\big(\varphi(x)\big) \qquad \text{for all } x \in FV(G).$$

Since $P \cup \{G\}$ is stable, Proposition 4 tells us that $\varphi(x)$ is ground for every $x \in FV(G)$. This implies that the substitution ϱ is of no use in (9) and therefore we have

$$\tau(x) \equiv \varphi(x) \qquad \text{for all } x \in FV(G),$$

proving our claim. □

5 Computational adequacy of consistent queried programs

In this final section we establish the important result that every primitive recursive function can be computed by an appropriate ordered program. Although the class of all computable functions is not exhausted by the class of all primitive recursive functions, it is arguable that all functions computable in practice, i.e. on an existing computer, are in fact primitive recursive. This is due to the fact that any recursive function, which is not primitive recursive is so complicated that any attempt to compute it readily exceeds the capacity of any computer.

Theorem 14 (Computational adequacy of consistent queried programs)

Let f be an n–ary primitive recursive function. Then there exists a normal program P_f with an $(n+1)$–ary predicate symbol p_f and an I/O–specification σ for P such that:

1. *For all nonnegative integers $k_1, \cdots, k_n$ the queried program*

$$P_f \cup \{\leftarrow p_f(s^{k_1}(0), \cdots, s^{k_n}(0), x)\}$$

   is stable.

2. *P_f is ordered.*

3. *For all nonnegative integers $k_1, \cdots, k_n$, and k we have*

$$f(k_1, \cdots, k_n) = k \quad \textit{iff}$$

   the substitution $[x/s^k(0)]$ is a computed answer for $P_f \cup \{\leftarrow p_f(s^{k_1}(0), \cdots, s^{k_n}(0), x)\}$.

Proof: The program we are going to construct will be the one given by J. Sebelik and P. Stepanek [SeSt82]. It is also given in the book of J. W. Lloyd [Llo87]. We will only have to prove 1 and 2 because 3 is Theorem 9.6 on page 53 of [Llo87].

First we will fix the I/O–specification σ. We do this by defining the mode for every $(n+1)$–ary predicate symbol p_f to be given by the declaration

$$p_f(\underbrace{+, \cdots, +}_{n}, -).$$

This definition of σ_{p_f} just reflects the intuition behind the concept of an *I/O*–specification, because, clearly, the first n places of p_f will serve for input, while the last place is intended for output.

To be able to talk of ordered programs we have to fix a function $\underline{\text{weight}}$, assigning to every function symbol occurring in P_f a positive integer. Since our alphabet has just the two function symbols 0 and s, we may define

$$\underline{\text{weight}}(s) := 1 \quad \text{and} \quad \underline{\text{weight}}(0) := 1.$$

We also have to fix the function $\underline{\text{level}}$ operating on the set of predicate symbols. But its construction will go hand in hand with the inductive definition of P_f, which follows the inductive construction of f from applications of compositions and primitive recursions.

Case 1: $f(k) = 0$ for all $k \in \mathbf{N}$.

We take P_f to be the program consisting of the single clause

$$p_f(x, 0) \leftarrow .$$

Defining $\underline{\text{level}}(p_f) := 0$ we see at once that 1 and 2 are true.

Case 2: $f(k) = k + 1$ for all $k \in \mathbf{N}$.

This time P_f consists of the single clause

$$p_f(x, s(x)) \leftarrow .$$

Again we take $\underline{\text{level}}(p_f) := 0$.

Case 3: $f(k_1, \cdots, k_n) = k_j$ where $j \in \{1, \cdots, n\}$.

We define P_f to be the program consisting of

$$p_f(x_1, \cdots, x_n, x_j) \leftarrow$$

and let $\underline{\text{level}}(p_f) := 0$.

Case 4: $f(k_1, \cdots, k_n) = h\big(g_1(k_1, \cdots, k_n), \cdots, g_m(k_1, \cdots, k_n)\big)$ for all $k_1, \cdots, k_n \in \mathbf{N}$.

By the induction hypothesis, corresponding to each g_i there is an ordered program P_{g_i} and a predicate symbol p_{g_i} satisfying the properties of the theorem. Similarly, corresponding to h, there is an ordered program P_h and a predicate symbol p_h satisfying the properties of the theorem. We can suppose without loss of generality that the programs $P_{g_1}, \cdots, P_{g_m}$ and P_h do not have any predicate symbols in common. We define P_f as the union of $P_{g_1}, \cdots, P_{g_m}$ and P_h together with the clause

$$p_f(x_1, \cdots, x_n, z) \leftarrow p_{g_1}(x_1, \cdots, x_n, y_1), \cdots, p_{g_m}(x_1, \cdots, x_n, y_m),\ p_h(y_1, \cdots, y_m, z).$$

It is straightforward to see that the new clause is stable. To turn P_f into a program which is ordered we simply assign to p_f a level which is bigger than the level of all the other predicate symbols occurring in P_f.

Case 5: Suppose that f is defined recursively by

$$\begin{array}{lcl} f(k_1, \cdots, k_n, 0) & = & h(k_1, \cdots, k_n), \\ f(k_1, \cdots, k_n, l+1) & = & g\big(k_1, \cdots, k_n, l, f(k_1, \cdots, k_n, l)\big), \end{array}$$

where g and h are primitive recursive functions. By the induction hypothesis, corresponding to h (resp., g), there is an ordered program P_h (resp., P_g) and a predicate symbol p_h (resp., p_g) satisfying the properties of the theorem. Again we can suppose that P_h and P_g do not have any predicate symbols in common. Define P_f to be the union of P_h and P_g together with the clauses

$$\begin{array}{lcl} p_f(x_1, \cdots, x_n, 0, z) & \leftarrow & p_h(x_1, \cdots, x_n, z) \qquad \text{and} \\ p_f\big(x_1, \cdots, x_n, s(y), z\big) & \leftarrow & p_f(x_1, \cdots, x_n, y, u),\ p_g(x_1, \cdots, x_n, y, u, z). \end{array}$$

Again it is easy to check that the two new clauses are stable. To turn P_f into an ordered program we assign to p_f a level higher than the level of any other predicate symbol occurring in P_f. Then the first clause is trivially seen to be ordered. Checking the second clause requires a little more work. First we write the second clause as $Q \leftarrow L_1, L_2$. Then we see that

$$I \ := \ \big\{i \in \{1,2\} \colon \underline{\text{level}}(L_i) = \underline{\text{level}}(Q)\big\} \ = \ \{1\}.$$

Furthermore we have

$$FV^+(Q) \ = \ FV^+\Big(p_f\big(x_1, \cdots, x_n, s(y), z\big)\Big) \ = \ \{x_1, \cdots, x_n, y\} \qquad \text{and}$$

$$FV^+(L_1) \ = \ FV^+\big(p_f(x_1, \cdots, x_n, y, u)\big) \ = \ \{x_1, \cdots, x_n, y\}.$$

Then it is clear that

$$FV^{+}(L_1) \subseteq_{multi} FV^{+}(Q)$$

holds. If we define $k := \underline{\text{level}}(p_f)$ and $l := \underline{\text{level}}(p_g)$ it remains to be proved that

$$\begin{array}{lcll} \{(k,x_1),\cdots,(k,x_n),(k,y)\} & \prec_{mset} & \{(k,x_1),\cdots,(k,x_n),(k,s(y))\} & \text{and} \\ \{(l,x_1),\cdots,(l,x_n),(l,y),(l,u)\} & \prec_{mset} & \{(k,x_1),\cdots,(k,x_n),(k,s(y))\} & \end{array}$$

holds. These inequation are valid since we have $s(y) \succ y$ and $k > l$. □

6 Conclusion

The last section shows that our approach can handle a large class of programs. But since our technique is syntactically and the question whether a given program terminates is undecidable, there have to be classes of programs for which the method described in this paper will fail. One such class consists of all those programs describing relations which are too complicated from a complexity point of view to be captured by our notions. More specifically, Urs–Martin Künzi [Kün92] has shown that functions which are not primitive–recursive can not be computed by an ordered program.

The other class of programs for which our method will fail consists of all those programs containing *local variables*. An example program will illustrate this point.

```
quicksort( [], [] ).
quicksort( [ X | Xs ], Y ) :-
        split( X, Xs, Z1, Z2 ),
        quicksort( Z1, Y1 ),
        quicksort( Z2, Y2 ),
        append( Y1, [ X | Y2 ], Y ).

split( _, [], [], [] ).
split( X, [ Y | Ys ], [ Y | Us ], Vs ) :-
        Y < X,
        split( X, Ys, Us, Vs ).
split( X, [ Y | Ys ], Us, [ Y | Vs ] ) :-
        Y >= X,
        split( X, Ys, Us, Vs ).

append( [], X, X ).
append( [ X | Xs ], Ys, [ X | Zs ] ) :-
        append( Xs, Ys, Zs ).
```

Example 1: The quicksort algorithm.

Let us assume that `quicksort/2` is called with its first argument being a list of natural numbers. Then the variables `Z1` and `Z2` on the first argument positions of the recursive calls to `quicksort/2` prevent us from directly comparing a recursive call with the head of the clause. This predicate can not be ordered in our sense, since for an ordered program we must have

$$\{Z1\} \subseteq_{multi} \{X,\ Xs\} \qquad \text{and} \qquad \{Z2\} \subseteq_{multi} \{X,\ Xs\},$$

which is apparently wrong. To deal with programs containing local variables one has to work with *abstract interpretations* [Fl66]. For a recent application of these ideas to Prolog see [VeSc91] or [Plü90]. Both of the aforementioned works propose the use of linear inequalities to prove termination of recursive Prolog procedures. While this certainly is the current state of the art, linear inequalities can only be successfull if the number of recursive calls of a procedure is bounded by some linear function. Although this seems to be true for most programs occurring in practice, from a more theoretical point of view this is somehow unsatisfactory.

For programs containing no local variables one can guarantee termination by restricting attention to *structural recursive procedures* [Plü90]. However, this notion is weaker than our notion since it does not allow for mutual recursion.

There is one final interesting point worth mentioning: It is decidable whether a program P with a given I/O–specification σ can be ordered via some functions $\underline{\text{level}}$ and $\underline{\text{weight}}$. This is so because there are essentially only finitely many ways to define the function $\underline{\text{level}}$ and once the function $\underline{\text{level}}$ is fixed we only have to find out whether there is a function $\underline{\text{weight}}$ such that certain inequalities between the weights of some terms are valid. But this last question can be reduced to checking the validity of a formula in Presburger arithmetic, which is a decidable theory [En72]. Let us illustrate this point by an example. Suppose we have a program consisting of the clauses.

```
p( h(X,Y) ) :- p( f(g(X),g(Y)) ).
p( g(f(f(X,Y),Y)) ) :- p( h(X,Y) ).
```

and assume that the predicate symbol p has the mode $p(+)$. In order to verify that this program can be ordered we have to find weights w_h, w_f and w_g such that

$$w_f + 2 * w_g \ < \ w_h \qquad \text{and} \qquad w_h < 2 * w_f + w_g$$

holds. Therefore we have to check whether the formula

$$\exists w_f \exists w_g \exists w_h \left((w_f + 2 * w_g \ < \ w_h) \ \wedge \ (w_h < 2 * w_f + w_g) \right)$$

is valid in Presburger arithmetic. (It turns out to be true; for example with $w_h = 8$, $w_f = 5$ and $w_g = 1$ the inequalities are satisfied.)

7 Acknowledgements

I wish to thank Prof. Dr. Justus Diller for the supervision of my Ph.D. thesis from which the current work has been extracted. Prof. Dr. John Addison carefully read a draft version of this paper and helped me with the English.

References

[ApPe90] Apt, K. R. and Pedreschi, D., Studies in Pure Prolog: Termination, in: *Proc. Esprit Symp. on Computational Logic*, Springer–Verlag, Berlin, 1990.

[ABW87] Apt, K. R., Blair, H. A. and Walker, A., Towards a Theory of Declarative Knowledge, in: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufman, Los Altos, 1987.

[Cav89] Cavedon, L. and Lloyd, J. W., A completeness theorem for SLDNF–resolution, *J. Logic Programming* **7**(4):177–193 (1989).

[Cav91] Cavedon, L., Acyclic logic programs and the completeness of SLDNF–resolution, *Theoretical Computer Science* **86**:81–92 (1991).

[Cla78] Clark, K. L., Negation as Failure, in: H. Gallaire and J. Minker (eds.), *Logic and Databases*, Plenum Press, New York, 1978.

[Der82] Dershowitz, N., Orderings for term–rewriting systems, *Theoretical Computer Science* **17**(3):279–301 (1982).

[DeJo90] Dershowitz, N. and Jouannaud, J.–P., Rewrite Systems, in: J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science (Vol. B: Formal Models and Semantics)*, Amsterdam, North–Holland, 1990.

[DeMa85] Dembinski, P. and Maluszynski, J., AND–parallelism with intelligent backtracking for annotated logic programs, in: *Proceedings of the International Symposium on Logic Programming*, Boston, 1985.

[En72] Enderton, H. B., *A Mathematical Introduction to Logic.*, Academic Press, New York, 1972.

[Fl66] Floyd, R. W., Assigning Meanings to Programs, in: J. T. Schwartz (ed.), *Mathematical Aspects of Computer Science*, Proc. Symposia in Applied Mathematics, Vol XIX, 1966, Providence, Rhode Island, 1967.

[He84] Heck, N., Abstrakte Datentypen mit automatischen Implementierungen, Ph.D. Thesis, Universität Kaiserslautern, Kaiserslautern, Germany, 1984.

[HeAv85] Heck, N. and Avenhaus, J., On logic programs with data–driven computations, *Proc. Eurocal '85*, LNCS 204, Springer–Verlag, Berlin, 1985.

[KaLe80] Kamin, S. and Levy, J.–J., Two generalizations of the recursive path ordering, *unpublished note*, Dept. of Computer Science, Univ. of Illinois, Urbana, IL, 1980.

[KnBe70] Knuth, D. E. and Bendix, P. B., Simple word problems in universal algebras, in: J. Leech (ed.), *Computational Problems in Abstract Algebra*, Pergamon Press, Oxford, 1970.

[Kun89] Kunen, K., Signed data dependencies in logic programs, *J. Logic Programming* **7**(4):231–245 (1989).

[Kün92] Künzi, U.–M., *Logic Programs for Primitive Recursive Sets*, Draft to be submitted to the Journal of Logic Programming (1992).

[LlTo86] Lloyd, J. W. and Topor, R., A basis for deductive database systems II, *J. Logic Programming* **3**(1):55–67, 1986.

[Llo87] Lloyd, J. W., *Foundations of Logic Programming*, 2nd ed., Springer–Verlag, Berlin, 1987.

[Plü90] Plümer, L., *Termination Proofs for Logic Programs*, Lecture Notes in Artificial Intelligence **446**, Springer–Verlag, Berlin, 1990.

[SeSt82] Sebelik J. and Stepanek P., Horn clause programs for recursive functions, in: K. L. Clark and S.–A. Tärnlund (eds.), *Logic Programming*, Academic Press, New York, 1982.

[Str91] Stroetmann, K., Vollständige Resolutionskalküle für PROLOG, Ph.D. Dissertation, Westfälische Wilhelms–Universität Münster, Münster, Germany, 1991.

[VeSc91] Verschaetse, K. and de Schreye, D., Deriving Termination Proofs for Logic Programs, Using Abstract Procedures, in: *Proc. Int. Conf. on Logic Programming*, Paris, 1991.