Note

Left termination turned into termination

Kees Doets

Department of Mathematics and Computer Science, University of Amsterdam, Plantage
Muidergracht 24, 1018 TV Amsterdam, The Netherlands

Communicated by G. Jäger
Received December 1990
Revised April 1991

Abstract

Doets, K., Left termination turned into termination, Theoretical Computer Science 124 (1994) 181–188

A transformation of programs is given that turns left terminating behaviour into terminating behaviour.
Subsequently this is used to obtain terminating programs computing total recursive functions.

1. Summary and introduction

This note presents a simple transformation * of (definite) logic programs P to logic programs P* with the following properties: (1) bodies of P*-rules consist of at most one atom (and so there is no selection rule problem for P*), and (2) if P is terminating under the Prolog left-most selection rule on each goal in a class AG of atomic goals, then P* terminates on each goal in AG with the same result (as regards success, failure and computed answer) as P.

Applying * to suitable left-terminating programs for total recursive functions (slightly extending results of Blair [3]), we obtain a more direct proof for a theorem of Bezem [2].

For unexplained notions, refer to [5] and [1].

Note that we do not write → backwards.
2. Coding sequences and numbers

Assume an individual constant [,] and a binary function symbol [,·].

2.1. Coding finite sequences

[,] codes the empty sequence.

As to terms [,s], if s codes a (finite) sequence then [,t,s] codes the sequence obtained from the one coded by s by the addition of t in front of it.

We have the usual notations:

(i) [,t₁,...,tₙ|s] := [,t₁,t₂,...,[tₙ|s]...],
(ii) [,t] := [t|nil] (sequence of length 1),
(iii) [,t₁,...,tₙ] := [,t₁,[t₂,...,[tₙ]...]] (sequence of length n).

In order to avoid cluttering of [, and ] and for ease of reading, below < and > are used for the same purpose as [, and ]. E.g., <[%1,3,...,%,%5]|w> := [%1,...,%5]|w|.

2.2. Coding numbers

(iv) √0 : := [,],
(v) √n+1 : := [%n]|%n|%n| (=%[n],%n−1%,%n−2%,...,√0%).

(The somewhat simpler √n+1 : := [%n] would work as well.)

3. The transformation

The transformation * translates an arbitrary logic program P in a language L into a program P* whose language is L∪{[,],[,]·,GOAL} (where GOAL is a new one-place relation symbol), as follows.

Suppose that P is a program in L. Assume that R₁,...,Rₘ is a complete list of all relation symbols of L.

3.1. Translation rules

Suppose that

R₁(t₁,...,tₙ),...,Rₘ(s₁,...,sₖ) → Rₖ(u₁,...,uₚ)

is any rule of P. The translation of this rule (using [,],[,]· and GOAL) reads

GOAL<[%n,%,%,%,...,%,%5]|w> → GOAL<[%n,%,%,...,%,%5]|w>,

where w is a new variable.

Note that, in the translated rule, atoms and clauses of the old rule have been coded into terms in an obvious fashion; a relation symbol is identified by its index.
3.2. Definition of the transformation

The transformed program $P^*$ consists of all translations of the $P$-rules, together with the following $m + 1$ rules (where $m$ is the total number of relation symbols of $P$):

- for $i = 1, \ldots, m$, we have the following start-rule:
  \[
  \text{GOAL}(\langle \forall i \setminus, x_1, \ldots, x_n \rangle) \rightarrow R_i(x_1, \ldots, x_n)
  \]
  (assuming $R_i$ to be $n$-place);

- finally we have the one bodyless end-rule:
  \[
  \rightarrow \text{GOAL}([])
  \]

**Theorem 1.** Suppose that $P$ is left terminating on a class of atomic goals $AG$. Then $P^*$ is terminating on $AG$ with the same results as $P$.

In other words, $P^*$ produces a finite SLD-tree for every goal $N$ in $AG$; the tree fails iff $P$ fails finitely on $N$ under the left-most rule and it succeeds iff $P$ succeeds on $N$ left-most, yielding the same answer substitutions as does $P$.

**Proof.** Each derivation from $P^*$ starting with an initial goal in the language $L$ corresponds in a unique fashion to a left-most rule SLD-derivation from $P$. The point of the translation is forcing left-most rule application, so to speak. $\square$

4. Terminating computation of total recursive functions

Suppose that an individual constant 0 and a unary function symbol $\cdot + 1$ are available. (In the setting of Section 2.2, we might just define $0 := []$ and $x + 1 := [x|x]$.)

Below, the natural number $n \in \mathbb{N}$ is identified with its canonical term $((0+1)+\cdots)+1$ in the Herbrand universe $\mathcal{U} = \mathcal{U}_p$ of closed terms.

A program $P$ computes the function $f: \mathbb{N}^k \rightarrow \mathbb{N}$ in the $(k+1)$-place relation symbol $F$ iff for all $n_1, \ldots, n_k, m$:

$$F(n_1, \ldots, n_k, m) \in T^\uparrow \iff f(n_1, \ldots, n_k) = m;$$

here, $T = T_p$ is the immediate consequence operator associated with $P$ and $T^\uparrow$ is its least fixed point.

If $P$ computes $f$ in $F$, the terms $t_1, \ldots, t_k$ are called the arguments of the atom $F(t_1, \ldots, t_k, s)$ and $s$ its value.

$AGG$ is the class of atomic goals of which the arguments are ground.

We are going to prove the following theorem.

**Theorem 2.** Every total recursive function can be computed by a program which is left terminating on the class $AGG$. 
This result only slightly strengthens (the formulation of) a result of [3], that total recursive functions may be determinately computed, cf. Section 5.2. We use a somewhat simpler implementation of minimalisation.

4.1.

Suppose that $f$ is a total recursive function.

Fix a recursive definition of $f$, that is: a list $f_0, \ldots, f_q = f$ such that for each $k \leq q$ one of the following six possibilities applies:

1. $f_k = \lambda n. n + 1$;
2. $f_k = \lambda n_0, \ldots, n_{i-1} \cdot n_j$ where $j < i$;
3. $f_k = \lambda n. 0$;
4. $f_k = \lambda n. f_j(f_j(0)(n), \ldots, f_j(p-1)(n))$ where $j, j(0), \ldots, j(p-1) < k$ (composition);
5. $f_k$ is the unique function satisfying $f_k(n, 0) = f_j(n)$ $f_k(n, p+1) = f_i(n, p, f_k(n, p))$ where $j, i < k$ (primitive recursion);
6. $f_k = \lambda n. \mu m [f_j(n, m) = 0]$ for some $j < k$ such that $\forall n \exists m [f_j(n, m) = 0]$ (minimalisation).

(Cf. [6]). We could dispense with (5) at the cost of adding a few basic functions, but that would not change much.

4.2.

Every recursive definition of a function $f = f_q$ in the sense of 4.1 can be transformed into a program which is left terminating on AGG, by associating rules with 4.1(1)-(6) as follows. (The rules are completely standard, except for the ones corresponding to (6)).

The program is written in a language with relation symbols $F_i$ in which the program computes $f_i (i \leq q)$; in the case of (6), we need an additional relation symbol $\text{ZERO}_k$.

R1 \quad \rightarrow F_k(x, x + 1)

R2 \quad \rightarrow F_k(x_0, \ldots, x_{i-1}, x_j)

R3 \quad \rightarrow F_k(x, 0)

R4 \quad F_{j(0)}(x, y_0), \ldots, F_{j(p-1)}(x, y_{p-1}), F_j(y_0, \ldots, y_{p-1}, z) \rightarrow F_k(x, z)

R5.1 \quad F_j(x, y) \quad \rightarrow F_k(x, 0, y)

2 \quad F_k(x, y, z), F_j(x, y, z, u) \rightarrow F_k(x, y + 1, u)$.
The desiderata for R1–5 are obvious. However, let me explain R6, which is slightly simpler than the solutions I came across in the literature (e.g., [4, 3]). (N.B.: remarkably, the μ-implementation which appears to be most straightforward – cf. [1] – has bad terminating behaviour.)

Let $g$ be a (say, binary) number-theoretic function.

Define the three-place relation zero by

$$\text{zero}(n, i, j) \equiv \forall j' < j [g(n, i + j') \neq 0] \land g(n, i + j) = 0.$$  

i.e., zero($n, i, j$) holds iff $i + j$ is the first zero $\geq i$ of the function $\lambda m. g(n, m)$; hence, zero is the graph of the minimalisation $\lambda n, i, j [g(n, i + j) = 0]$. (This may very well be only a partial function, even if the more special minimalisation $\lambda n, m [g(n, m) = 0]$ we are aiming at is total. In fact, this function is total iff for each $n, \lambda m. g(n, m)$ has infinitely many zeros.)

Observe that the following constitutes a recursive definition of zero (the recursion is on the last argument):

$$\text{zero}(n, i, 0) \iff g(n, i) = 0,$$

$$\text{zero}(n, i, j + 1) \iff g(n, i) \neq 0 \land \text{zero}(n, i + 1, j).$$

Of course, this is not a primitive recursion since the second argument has not been kept fixed.

Now, R6.1–2 constitute the obvious translation of this recursion, assuming that $g$ is computed in the symbol $F_j$. R6.3 needs no explanation.

4.3.

It is rather clear that these rules compute what they are supposed to. (Note that only the rules R6.1–2 can be responsible for branching of left-most rule SLD trees for atomic goals $F_k(n, i)$ in AGG. It may be instructive to see R6.1–2 compute a μ-defined function $\mu m [g(m) = 0]$: “determine $g(0), g(1), g(2), \ldots$ until the first zero is met”.)

It remains to verify termination on atomic goals from AGG under the left-most selection rule. (N.B.: Zero-atoms are not admitted in AGG. If $f_k$ has been obtained from $f_j$ by the μ-operator, R6.2 will produce an infinite regress on $\text{ZERO}_k(n, m, z)$ in case $\forall m' \geq m [f_j(n, m') \neq 0]$.)

Termination is checked using induction on indices of the relation symbols $F_k$ involved. Only the rules R4, R5 and R6 have to be considered.

Application of R4 to an atom in AGG leaves a resolvent in which all atoms but the last one are in AGG as well. If the program does not left-most fail finitely on these, then by induction hypothesis it succeeds. Since the rules are correct for the intended
interpretation, the computed answer must provide ground answers for the arguments of the last atom. After that, the induction hypothesis applies.

Termination of the R5-rules is easily seen using a secondary induction on the argument with respect to which the primitive recursion is carried out. When put to work on $F_k(n, t) \in AGG$, the R6-rules will produce resolvents consisting of $F_j$-atoms in $AGG$ and $ZERO_k$-atoms with all but the last argument ground. By induction hypothesis, the program is left-most rule terminating on atoms of the first type. Therefore, only R6.2 can be responsible for an infinite computation. Along such an infinite computation, by the repeated application of R6.2 starting with the R6.3-produced atom $ZERO_k(n, 0, t)$ atoms $F_j(n, 0, u^0 + 1)$, $F_j(n, 1, u^1 + 1)$, $F_j(n, 2, u^2 + 1)$, ... must occur. However, if $k = \mu m [f_j(n, m) = 0]$ (and this number $k$ must exist for the $\mu$-operator to be applicable!), the program must fail left-most rule on the $(k + 1)$st atom $F_j(n, k, u^k + 1)$; whence the computation cannot have been infinite.

4.4. Terminating programs for total recursive functions

We have programs left terminating on $AGG$ and computing all total recursive functions.

Clearly, these programs are left terminating on all ground atomic goals. For ground instances of $AGG$-atoms this is obvious. As to $ZERO$-atoms, just note that, for a specific ground case, rule R6.2 can be applied only a finite number of times.

Applying 3.3, we obtain the following result of [2], proved there using Shepherdson’s translation from register machines:

**Theorem 3.** Every total recursive function can be computed by a program which is terminating on ground atomic goals.

**Proof.** If $P$ is a program constructed in 4.2, $P^*$ is terminating on ground atomic goals of the $P$-language; and it is easily checked that it will terminate on ground atomic goals of the extended language (with $\cdot\cdot$, $\cdot\cdot\cdot$ and $GOAL$) as well. □

**Remark.** Our result is weaker than Bezem’s in the sense that we need the addition of a binary function symbol $\cdot\cdot\cdot$ for coding-purposes, whereas Bezem does not extend the Herbrand universe.

4.5. Computation of partial recursive functions

The implementation of the $\mu$-operator works as well for partial recursive functions (which are obtained on leaving out the existence condition in 4.1(6)).
5. Remarks

5.1.

By the above, the "natural" program rules computing a primitive recursion are left terminating on the class $AGG$. Of course, this is true also for recursions much more general than just primitive recursion (if the bodies of the rules have been written in the "right" order).

An example forms the recursion for the zero-function above (under a strengthened existence condition). A much easier one (its program does not produce branching in SLD-trees as do the R6.1–2 rules) is that of the Ackermann function. [2] presents a terminating program computing it; however, our translation effortlessly produces such a program. Of course, it fails to have the additional niceties ("multidirectionality") of the one of Bezem.

5.2.

A program $P$ is called determinate by [3] in case $T\downarrow T^\uparrow$ ($T= T_\rho$ the immediate consequence operator of $P$). [2] shows this to be tantamount to what he calls weak recurrency of $P$, demanding a rank-function $\rho$ associating natural numbers with ground atoms and satisfying the condition that for every ground-atom $A\notin T^\uparrow$:

for every ground instance $C\rightarrow A$ of a $P$-rule,

there exists $B\in C$ such that $B\notin T^\uparrow$ and $\rho B < \rho A$.

Note that weak recurrency of the rules R1–6 for total recursive functions is witnessed by the rank associating with a ground atom the size of its leftmost SLD-tree. This rank depends on arguments of the atom only; it does not differ essentially from the number of steps needed to calculate the value of the corresponding function (on the arguments given) according to the definition of the function in the sense of 4.1.

Alternatively, determinateness of the R1–6 rules for total recursive functions is straightforward also from the observation that each program which is terminating on atomic ground goals under some selection rule has complementary success and finite failure sets.

Acknowledgment

Thanks are due to Marc Bezem, Krzysztof Apt, Johan van Benthem and the referee for comments, suggestions and discussions.
References


