PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Automatic Verification of GHC-Programs: Termination

Lutz Plümer
Rheinische Friedrich-Wilhelms-Universität Bonn, Institut für Informatik III
D-5300 Bonn 1, Römerstr. 164
lutz@uran.informatik.uni-bonn.de

Abstract

We present an efficient technique for the automatic generation of termination proofs for concurrent logic programs, taking Guarded Horn Clauses (GHC) as an example. In contrast to Prolog's strict left to right order of evaluation, termination proofs for concurrent languages are complicated by a more sophisticated mechanism of subgoal selection. We introduce the notion of directed GHC programs and show that for this class of programs goal reductions can be simulated by Prolog-like derivations. We give a sufficient criterion for directedness. Static program analysis techniques developed for Prolog can thus be applied, albeit with some important modifications.

1. Introduction

With regard to termination it is useful to distinguish between two types of software systems or programs: transformational and reactive [HAP85]. A transformational system receives an input at the beginning of its operation and yields an output at the end. If the problem at hand is decidable, termination of the process is surely a desirable property. Reactive systems, on the other hand, are designed to maintain some interaction with their environment. Some of them, for instance operating systems and database management systems, ideally never terminate and do not yield a final result at all. Based on the process interpretation of Horn clause logic, concurrent logic programming systems have been designed for many different applications including reactive systems and transformational parallel systems. While for some of them termination is not a desirable property, for others it is. In this paper we discuss how automatic termination proofs for concurrent logic programs can be achieved automatically.

Automatic proof techniques for pure Prolog programs have been described in several papers including [ULG88] and [PLU90a]. Prolog is characterized by a fixed computation rule which always selects the leftmost atom. Deterministic subgoal selection and strict left to right order of evaluation cannot be assumed for the concurrent languages.

Static program analysis techniques, which are well established for sequential Prolog, such as abstract interpretation, inductive assertions and termination proof techniques, substantially depend on the strict left to right order of evaluation in most cases and thus cannot easily be applied to concurrent languages. Concurrent languages delay subgoals which are not sufficiently instantiated. Goals which loop forever when evaluated by a Prolog interpreter may deadlock in the context of a concurrent language. These phenomena may suggest that termination proofs for concurrent logic programs require a different approach. This paper, however, shows that techniques which have been established for pure Prolog are still useful in the context of concurrency.

Our starting point is the question under which conditions reductions of a concurrent logic program can be simulated by Prolog-like derivations. We take Guarded Horn Clauses (GHC, see [UED86]) as an example, but our results can easily be extended to other concurrent logic programming languages such as PARLOG, (Flat) Concurrent Prolog or FCP(:). Our basic assumptions are the restriction of unification to input matching, nondeterministic subgoal selection and resuming of subgoals which are not sufficiently instantiated. Since we consider all possible derivations, the commit operator does not need special attention.

In general simulation is not possible: if there is a GHC-derivation of g' from g, g' cannot necessarily be derived with Prolog's computation rule.

One could now try to augment simulation by program transformation. Let, for instance, P' be derived from P by including all clause body permutations. Although P' may be exponentially larger than P, there are still derivations which are not captured.

Example 1.1:

Program: $p \leftarrow q,r.$ $q \leftarrow s,t.$ $r \leftarrow u,v.$
$s.$ $v.$

Goal: $\leftarrow p$

This goal can be reduced to $\leftarrow t,u$ by nondeterministic subgoal selection, but not by a Prolog like computation, even after adding the following clauses:

$p \leftarrow r,q.$ $q \leftarrow t,s.$ $r \leftarrow v,u.$

The reason is that in order to derive $\leftarrow t,u$, the subderivations of $\leftarrow q$ and $\leftarrow r$ have to be interleaved.

The question arises whether there is an interesting subclass for which appropriate simulations can be defined. Such a class of programs will be discussed in Section 3. The main idea is to assume that if a subgoal p may produce some output on which evaluation of another subgoal q depends, then p is smaller w.r.t. some partial ordering. Whether a program maintains such a property, which we will call directedness, is undecidable. We will then introduce the stronger notion of well-formedness which can be checked syntactically. Well-formedness is related to directionality, which is discussed in [GRE87]. Well-formedness is sufficient but not necessary for directedness, and it will turn out that quite a lot of nontrivial programs (including for instance systolic programs as discussed in [SHA87a] and most of the examples given in [TIC91]) fall into this category. In Section 5 we will demonstrate how termination proof techniques which have been established for pure Prolog can be generalized such that they apply to well-formed GHC programs.

The rest of this paper is organized as follows. Section 2 provides basic notions. Section 3 introduces the notion of directed programs and shows that this property is undecidable. It provides the notion of well-formedness and shows that it is sufficient for directedness. Section 4 discusses oriented and data driven computation and shows that after some simple program transformation derivations with directed GHC-programs can be simulated by Prolog-like derivations. Using the notion of S-models introduced in [FLP89], Sections 5 and 6 show how termination proofs can be achieved automatically.

2. Basic Notions

We use standard notation and terminology of Lloyd [Llo87] or Apt [APT90]. Following [APP90] we will say LD-resolution (LD-derivation, LD-refutation LD-tree) for SLD-resolution (SLD-derivation, SLD-refutation SLD-tree) with the leftmost selection rule characteristic for Prolog.

Next we define GHC programs following [UED87] and [UED88].

A GHC program is a set of guarded Horn clauses of the following form:

$$H \leftarrow G_1,\ldots,G_m \mid B_1,\ldots,B_n. \qquad (m > 0, n > 0)$$

where H, $G_1,\ldots,G_m$ and $B_1,\ldots,B_n$ are atomic formulas. H is called a clause head, the G_i's are called guard goals and the B_i's are called body goals. The part of a clause before '|' is called a guard, and the part after '|' is called a body. One predicate, namely '=', is predefined by the language. It unifies two terms.

Declaratively, the commitment operator '|' denotes conjunction, and the above guarded Horn clause is read as "H is implied by $G_1,\ldots,G_m$ and $B_1,\ldots,B_n$". The operational semantics of GHC is given by parallel input resolution restricted by the following two rules:

Rule of Suspension:

- Unification invoked directly or indirectly in the guard of a clause C called by a goal G (i.e. unification of G with the head of C and any unification invoked by solving the guard goals of C) cannot instantiate the goal G.
- Unification invoked directly or indirectly in the body of a clause C called by a goal G cannot instantiate the guard of C or G until C is selected for commitment.

Rule of Commitment:

- When some clause C called by a goal G succeeds in solving (see below) its guard, the clause C tries to be selected for subsequent execution (i.e., proof) of G. To be selected, C must first confirm that no other clauses in the program have been selected for G. If confirmed, C is selected indivisibly, and the execution of G is said to be committed to the clause C.

An important consequence is that any unification intended to export bindings to the calling goal must be specified in the clause body and use the predefined predicate '='.

The operational semantics of GHC is a sound - albeit not complete - proof procedure for Horn clause programs: if $\leftarrow B$ succeeds with answer substitution θ, then $\forall(B\theta)$ is a logical consequence of the program.

Subsequently, we may find it convenient to denote a goal g by the pair $\langle G;\theta\rangle$, i.e. $g = G\theta$. A single derivation step reducing the i-th atom of G using clause C and applying mgu θ' is denoted by $\langle G;\theta\rangle \rightarrow_{i;C} \langle G';\theta\theta'\rangle$. Subscripts may be omitted.

3. Directed Programs

An *annotation* d_p for an n-ary predicate symbol p is a function from $\{1,\ldots,n\}$ to $\{+,-\}$ where '+' stands for input and '-' for output. We will write p(+,+,-) in order to state that the first two arguments of p are input and the last is output.

A goal atom A *generates (consumes)* a variable v if v occurs at an output (input) position of A. A *is generator* for B, if some variable v occurs at an output position of A and at an input position of B; in this case, B is *consumer of* A.

Let $\hat{t}$ denote a tuple of terms. A derivation $\langle p(\hat{t});\varepsilon\rangle \rightarrow^* \langle G;\theta\rangle$ *respects* the input annotation of p if $v\theta = v$ for every variable v occurring at an input position of $p(\hat{t})$.

A *goal* is *directed* if there is a linear ordering among its atoms such that if A_i is generator for A_j then A_i precedes A_j in that ordering. A *program* is *directed*, if all its derivations *respect directedness*, i.e., all goals derived from a directed goal are directed. Note that directedness of a goal is a static

property which can be checked syntactically. Directedness of a program, however, is a dynamic property.

Theorem 3.1: It is undecidable, whether a program is directed.

Proof: Let $t_M(X)$ be a directed GHC simulation of a Turing machine M for a language L which binds X to halt if and only if M applied to the empty tape halts. Such a simulation is for instance described in [PLU90b]. Next consider the following procedures p_M and q:

$p_M(X,Y) \leftarrow t_M(A), q(A,X,Y).$
$q(halt,X,X).$

and the (directed) goal

$\leftarrow r(X,Y), s(Y,Z), p_M(X,Z).$

The following annotations are given:

$t_M(-).$ $q(+,-,-).$ $p_M(-,-).$ $r(+,-).$ $s(+,-).$

If M halts on the empty tape, $t_M(A)$ will bind A to 'halt', $p_M(X,Y)$ will identify X and Y and thus the given goal can be reduced to the undirected goal $\leftarrow r(X,Y), s(Y,X)$. Decidability of program directedness would thus imply solvability of the halting problem: contradiction. ■

Next we introduce the notion of well-formedness of a program w.r.t. a given annotation and show that this property is sufficient for directedness.

A *goal* is *well-formed* if it is directed, generators precede consumers in its textual ordering, and its output is unrestricted. Output of a *goal* is *unrestricted* if all its output arguments are distinct variables which do not occur (i) at an output position of another goal atom and (ii) at an input position of the same atom.

A *program* P is *well-formed* if the following conditions are satisfied by each clause $H \leftarrow G_1,\ldots,G_m \mid B_1,\ldots,B_n$ in P:

- $\leftarrow B_1,\ldots,B_n$ is well-formed
- the input variables of H do not occur at output positions of body atoms.

The predicate '=' has the annotation '- = -'. It is convenient to have two related primitives: '==' (test) and '⇐' (matching) which have the same declarative reading as '=' but different annotations, namely '+ == +' and '- ⇐ +'.

Note that the goal $\leftarrow r(X,Y),s(Y,Z), p_M(X,Z)$ is not well-formed because its output is restricted: Z has two output occurrences.

The next example is taken from [UED86]:

Example 1: Generating primes

```
primes(Max,Ps)    ← true |
                    gen(2,Max,Ns),sift(Ns,Ps).
gen(N,Max,Ns)     ← N ≤ Max | N1 ⇐ N + 1,
                    gen(N1,Max,Ns1), Ns ⇐[N|Ns1].
gen(N,Max,Ns)     ← N > Max | Ns ⇐ [].
sift([P|Xs],Zs)   ← filter(P,Xs,Ys),sift(Ys,Zs1),
                    Zs ⇐ [P|Zs1].
sift([],Zs)       ← Zs ⇐ [].
filter(P,[X|Xs],Ys) ← X mod P == 0 | filter(P,Xs,Ys).
filter(P,[X|Xs],Ys) ← X mod P ≠0 | filter(P,Xs,Ys1),
                    Ys ⇐ [X|Ys1].
filter(P,[],Ys)   ← Ys ⇐ [].
primes(+,-).  gen(+,+,-).  sift(+,-).  filter(+,+,-).
```

The call primes(Max,Ps) returns through Ps a stream of primes up to Max. The stream of primes is generated from a stream of integers by filtering out the multiples of primes. For each prime P, a filter goal filter(P,Xs,Ys) is generated which filters out the multiples of P from the stream Xs, yielding Ys.

In this example all input terms are italic and all output terms are bold. It can easily be seen that this program is well-formed.

Another example for a well-formed program is quicksort. The call qsort([H|L],S) returns through S an ordered version of the list [H|L]. To sort [H|L] L is split into two lists L_1 and L_2 which are itself sorted by recursive calls to qsort.

Example 2: Quicksort

```
q1: qsort([],L)          ← L ⇐ [].
q2: qsort([H|L],S)       ← split(L,H,A,B),
                           qsort(A,A1), qsort(B,B1),
                           append(A1,[H|B1],S).
s1: split([],X,L1,L2)    ← L1 ⇐ [], L2 ⇐ [].
s2: split([X|Xs],Y,L1',L2) ← X ≤ Y |
                           split(Xs,Y,L1,L2),
                           L1' ⇐ [X|L1].
s3: split([X|Xs],Y,L1,L2') ← X > Y | split(Xs,Y,L1,L2),
                           L2' ⇐ [X|L2].
a1: append([],L1,L2)     ← L2 ⇐ L1.
a2: append([H|L1],L2,L3) ← append(L1,L2,L3'),
                           L3 ⇐ [H|L3'].
split(+,+,-,-).  qsort(+,-).   append(+,+,-).
```

Theorem 3.2: Let P be a well-formed program, g a well-formed goal and $g \rightarrow^* g'$ a GHC-derivation. Then g' is well-formed.

Proof: See [PLU92].

Well-formed programs respect input annotations:

Theorem 3.3: Let $\langle p(\bar{t}),\varepsilon\rangle \rightarrow^* \langle G';\theta\rangle$ be a derivation and v an input variable of $p(\bar{t})$. Then $v\theta = v$.

Proof: Goal variables can only be bound by transitions applying '=' or '⇐', since in the other cases matching substitutions are applied. Since both arguments of '=' are output, and '⇐' also binds only output variables, input variables cannot be bound. ■

4. Oriented and Data Driven Computations

Our next aim is to show that derivations of directed programs can be simulated by derivations which are similar to LD-derivations. In this context we find it convenient to use the notational framework of SLD-resolution and to regard GHC-derivations as a special case.

We say that an SLD-derivation is *data driven*, if for each resolution step with selected atom A, applied clause C and mgu θ either C is the unit clause ($X = X \leftarrow$ true.) or C is $B \leftarrow B_1,\ldots,B_n$ and $A = B\theta$. Data driven derivations are the same as GHC derivations of programs with empty guards. The assumption that guards are empty is without loss of generality in this context.

Next we consider oriented computation rules. Oriented computation rules are similar to LD-resolution in the sense that goal reduction strictly proceeds from left to right. They are more general since the selected atom is not necessarily the leftmost one. However, if the selected atom is not leftmost, its left neighbors will not be selected in any future derivation step.

More formally, we define: A computation rule R is *oriented*, if every derivation $\langle G_0;\varepsilon\rangle \rightarrow \ldots \langle G_i;\theta_i\rangle \rightarrow \ldots$ via R satisfies the following property: If in G_i an atom A_k is selected, and A_j ($j < k$), is an atom on the left of A_k, no further instantiated version of A_j will be selected in any future derivation step.

Our next aim is to show that, for directed programs, any data driven derivation can be simulated by an equivalent data driven derivation which is oriented. To prove the following theorem, we need a slightly generalized version of the switching lemma given in [LLO87]. Here $g \rightarrow_{i;C;\theta} g'$ denotes a single derivation step where the i-th atom of g is resolved with clause C using mgu θ.

Lemma 4.1: Let g_{k+2} be derived from g_k via $g_k \rightarrow_{i;C_{k+1};\theta_{k+1}} g_{k+1} \rightarrow_{j;C_{k+2};\theta_{k+2}} g_{k+2}$. Then there is a derivation $g_k \rightarrow_{j;C_{k+2}';\theta_{k+1}'} g_{k+1}' \rightarrow_{i;C_{k+1}';\theta_{k+2}'} g_{k+2}'$ such that g_{k+2}' is a variant of g_{k+2} and C_{k+1}', C_{k+2}' are variants of C_{k+2} and C_{k+1}.

Proof: [LLO87] The difference between this and Lloyds version is that the latter refers to SLD-refutations, while ours refers to (possibly partial) derivations. His proof, however, also applies to our version. ■

Theorem 4.2: Let P be a directed program and $\langle G_0;\varepsilon\rangle$ a directed goal. Let $D = \langle G_0;\varepsilon\rangle \rightarrow \ldots \langle G_k;\theta_k\rangle$ be a data driven derivation using the clause sequence $C_1,\ldots,C_k$. Then there is another data driven derivation D': $\langle G_0;\varepsilon\rangle \rightarrow \ldots \langle G_k';\theta_k'\rangle$ using a clause sequence $C_{i_1}',\ldots,C_{i_k}'$, where $\langle i_1,\ldots,i_k\rangle$ is a permutation of $\langle 1,\ldots,k\rangle$, each C_i' is a variant of C_i and $G_k'\theta_k'$ is a variant of $G_k\theta_k$, and D' is oriented.

Proof: Let g_j be the first goal in D where orientation is violated, i.e. there is the following situation:

$g_i : \langle B_1,\ldots,R,\ldots,R',\ldots\ldots;\theta_i\rangle$

...

$g_j : \langle B_1,\ldots,R,\ldots\ldots\ldots;\theta_j\rangle$

R' is selected in g_i and R is selected in g_j. Now we switch subgoal selection in g_{j-1} and g_j and get a new derivation D*. In D* we look again for the first goal violating the orientation. After a finite number of iterations, we arrive at a derivation D' which is oriented. It remains to be shown that D* (and thus D') is still data driven.

Note that up to g_{j-1} both derivations are identical. Above, the switching lemma implies that, from g_{j+1} on, the goals of D' are variants of those of D.

Now let Q be the selected goal of G_{j-1}. Since orientation is violated for the first time in G_j, Q is to the right of R. (if $i = j - 1$ then $Q = R'$, and otherwise $j-1$ would have the first violation of orientation.) Since $g_{j-1} = \langle G_{j-1};\theta_{j-1}\rangle$ is directed, $Q\theta_{j-1}$ is not a generator of $R\theta_{j-1}$ and thus $R\theta_{j-1}$ and $R\theta_j$ are variants. Let H be the head of the clause applied to resolve R in $\langle G_j;\theta_j\rangle$. Since D is data driven, $R\theta_{j-1} = H\sigma$ for some σ, and so $R\theta_j = H\sigma'$ for some σ'. Thus D' is data driven. ■

Corollary 4.3: Let P be a directed program and g a directed goal. Then g has an infinite data driven derivation if and only if it has an infinite data driven derivation which is oriented.

According to Corollary 4.3, in our context it is sufficient to consider data driven derivations which are oriented. Such derivations are still not always LD-derivations since the selected atom is not necessarily leftmost. If it is not, however, its left neighbors will never be reactivated in future derivation steps; thus w.r.t. termination they can simply be ignored. The same effect can be achieved by a simple program transformation proposed in [FAL88]:

$Pr_G(P) =$ $\{p(\bar{\bar{X}}) \leftarrow \mid$ p is an n-ary predicate appearing in the body or the head of some clause of P and $\bar{\bar{X}}$ is an n-tuple of distinct variables$\}$

$Part_G(P) =$ $P \cup Pr_G(P)$

Simulation Lemma 4.4: Let $D = G_0 \rightarrow \ldots G_{i-1} \rightarrow G_i$ be an oriented SLD-derivation of G_0 and P where

G_{i-1} $= \leftarrow B_1,\ldots,B_j\ldots,B_n$ and

G_i $= \leftarrow (B_1,\ldots,B_{j-1},C_i^+,B_{j+1},\ldots,B_n)\theta_i$.

C_i^+ is the body of the C_j applied to resolve B_j. Then there is an LD-derivation

D' $= G_0 \ldots \rightarrow \ldots G_{k-1}' \rightarrow G_k'$ with $Part_G(P)$, where

G_{k-1}' $= \leftarrow B_j\ldots,B_n$ and

G_k' $= \leftarrow (C_i^+,B_{j+1}\ldots,B_n)\theta_i$.

Proof: Whenever an atom B is selected in D which is not the leftmost one, first the atoms to the left of B are resolved

away in D' with clauses in $Pr_G(P)$, and then D' resolves B in the same way as D. ■

An immediate implication is the following:

Theorem 4.5: If g has a non-terminating data driven oriented derivation with P, then it has a nonterminating LD-derivation with $Part_G(P)$.

The converse, however, is not true. Consider, for instance, the quicksort example from above, extended by the following clauses

q_0: qsort(_,_).
s_0: split(_,_,_,_).
a_0: append(_,_,_).

While the LD-tree for ← qsort([2,1],X) is finite in the context of the standard definition of qsort, it is no longer true for the extended program. Consider the following infinite LD-derivation:

← qsort([2,1],X)
by q_2: ← split([1],2,A,B), qsort(A,A_1), qsort(B,B_1), append(A_1,[H|B_1],S).
by s_0: ← qsort(A,A_1), qsort(B,B_1), append(A_1,[H|B_1],S).
by q_2: ← split(_,_,_,_), ...
by s_0: ← qsort(_,_),...

This derivation, however, is not data driven: resolving qsort(A,A_1) in the third goal with q_2 yields an mgu which is not a matching substitution.

For *data driven* LD-derivations we get a stronger result:

Theorem 4.6: There is a nonterminating data driven oriented derivation for g with P if and only if there is a nonterminating data driven LD-derivation for g with $Part_G(P)$.

Proof: The only-if part is implied by the simulation lemma. For the if-part, consider a nonterminating, data driven LD-derivation D. By removing all applications of clauses in $Pr_G(P)$, one gets another derivation D'. D' is a nonterminating data driven oriented derivation. ■

Restriction to LD-derivations which are data-driven enlarges the class of goal/program pairs which do not loop forever. In the general case, termination of quicksort requires that the first argument is a list. Termination of append requires that the first or the third argument is a list. Restriction to data-driven LD-derivation implies that no queries of quicksort or append (and many other procedures which have finite LD-derivations only for certain modes) loop forever. However, goals like ← append(X,Y,Z) or ← quicksort(A,B) deadlock immediately.

5. Termination Proofs

In this section, we will give a sufficient condition for terminating data driven LD-derivations. We will concentrate on programs without mutual recursion. In [PLU90b] we have demonstrated how mutual recursion can be transformed into direct recursion. We need some further notions.

For a set T of terms, a *norm* is a mapping $|\ldots|: T \to N$. The mapping $\|\ldots\|: A \to N$ is an input norm on (annotated) atoms, if for all $B = p(t_1,\ldots,t_n)$, $\| B \| = \sum_{i\in I} | t_i |$, where I is a subset of the input arguments of B.

Let P be a well-formed program without mutual recursion. P is safe, if there is an input norm on atoms such that for all clauses $c = B_0 \leftarrow B_1,\ldots,B_i,\ldots,B_n$ the following holds: If B_i is a recursive literal (B_0 and B_i have the same predicate symbol), σ a substitution the domain of which is a subset of the input variables of B_0 and θ is a computed answer for $\leftarrow (B_1,\ldots,B_{i-1})\sigma$, then $\|B_0\sigma\theta\| > \|B_i\sigma\theta\|$.

We can now state the following theorem:

Theorem 5.1: If P is a safe program and $G = \leftarrow A$ is well-formed, then all data driven LD-derivations for G are finite.

PROOF: By contradiction. Assume that there is an infinite data driven LD-derivation D. Then there is an infinite subsequence D' of D containing all elements of D starting with the same predicate symbol p. Let d_i and d_{i+1} be two consecutive elements of D' and

d_i = $\leftarrow p(t_1,\ldots,t_r), \ldots$
d_{i+1} = $\leftarrow p(t'_1,\ldots,t'_r), \ldots$
and c_i =
$p(s_1,\ldots,s_r) \leftarrow B_1,\ldots,B_k,p(s'_1,\ldots,s'_r),\ldots$

be the clause applied to resolve the first literal of d_i, θ_i the corresponding mgu. Then there is a computed answer substitution θ' for $\leftarrow (B_1,\ldots,B_k)\theta_i$ such that $p(t'_1,\ldots,t'_r) = p(s'_1,\ldots,s'_r)\theta_i\theta'$.

Since D is data driven, θ_i is a matching substitution, i.e. $p(t_1,\ldots,t_r) = p(t_1,\ldots,t_r)\theta_i$. Since P is well-formed, Theorem 3.3 further implies $p(t_1,\ldots,t_r) = p(t_1,\ldots,t_r)\theta_i\theta'$. We also have $p(t_1,\ldots,t_r)\theta_i\theta' = p(s_1,\ldots,s_r)\theta_i\theta'$.

Since P is a safe program
$\|p(s_1,\ldots,s_r)\theta_i\theta'\| > \|p(s'_1,\ldots,s'_r)\theta_i\theta'\|$ and thus
$\|p(t_1,\ldots,t_r)\theta_i\theta'\| > \|p(t'_1,\ldots,t'_r)\theta_i\theta'\|$. Since the range of $\|\ldots\|$ is a well-founded set, D' cannot be infinite. Contradiction. ■

The next question is how termination proofs for data driven LD-derivations can be automated. In [PLU90b] and [PLU91], a technique for automatic termination proofs for Prolog programs is described. It uses an approximation of the program's semantics to reason about its operational behavior. The key concept are predicate inequalities which relate the argument sizes of the atoms in the minimal Herbrand model of the program. Now in any program $Part_G(P)$ for every predicate symbol p occurring in P there is a unit clause $p(\vec{X})$. Thus the minimal Herbrand model M_P of P equals the Herbrand base B_P of P, a semantics which is

not helpful. To overcome this difficulty, we will consider S-models which have been proposed in [FLP89] in order to model the operational behaviour of logic programs more closely. The S-model of a logic program P can be characterized as the least fixpoint of an operator T_s which is defined as follows:

$$T_s(I) = \{B \mid \exists\ B_0 \leftarrow B_1,\ldots,B_k \text{ in } P, \exists\ B_1',\ldots,B_k' \in I, \exists\ \delta = mgu((B_1,\ldots,B_k),(B_1',\ldots,B_k')), \text{and } B = B_0\delta\}.$$

We need some notions defined in [BCF90] and [PLU91].

Let Δ be a mapping from a set of function symbols F to $\mathbf{N}$ which is not zero everywhere. A norm $|\ldots|$ for T is said to be *semi-linear* if it can be defined by the following scheme:

$|t| = 0$ if t is a variable

$|t| = \Delta(f) + \sum_{i\in I} |t_i|$ if $t = f(t_1, \ldots t_n)$,

where $I \subseteq \{1,\ldots,n\}$ and I depends on f.

A subterm t_i is called selected if $i \in I$.

A term t is *rigid* w.r.t. a norm $|\ldots|$ if $|t| = |t\theta|$ for all substitutions θ. Let $t[v_{(i)}\leftarrow s]$ denote the term derived from t by replacing the i-th occurrence of v by s. An occurrence $v_{(i)}$ of a variable v in a term t is *relevant* w.r.t. $|\ldots|$ if $|t[v_{(i)}\leftarrow s]| \neq |t|$ for some s. Variable occurrences which are not relevant are called irrelevant. A variable is relevant if it has a relevant occurrence. Rvars(t) denotes the multiset of relevant variable occurrences in the term t.

Proposition 5.2: Let t be a term, $t\theta$ be a rigid term and V be the multiset of relevant variable occurrences in t. Then for a semi-linear norm $|\ldots|$ we have $|t\theta| = |t| + \sum_{v\in V} |v\theta|$.

Corollary 5.3: $|t\theta| \geq |t|$.

Proof: [PLU91]

For an n-ary predicate p in a program P, a *linear predicate inequality* LI_p has the form $\sum_{i\in I} p_i + c \geq \sum_{j\in J} p_j$, where I and J are disjoint sets of arguments of p, and c, the offset of LI_p, is either a natural number or ∞ or a special symbol like γ. I and J are called input resp. output positions of p (w.r.t. LI_p).

Let M_S be the S-model of P. LI_p is called *valid* (for a linear norm $|\ldots|$) if $p(t_1,\ldots,t_n) \in M_S$ implies $\sum_{i\in I} |t_i| + c \geq \sum_{j\in J} |t_j|$.

Let $A = p(t_1,\ldots,t_n)$. With the notations from above we further define:

- $F(A,LI_p) = \sum_{i\in I} |t_i| - \sum_{j\in J} |t_j| + c.$
- $V_{in}(A,LI_p) = \cup\, rvars(t_i)$
- $V_{out}(A,LI_p) = \cup\, rvars(t_j)$
- $F_{in}(A,LI_p) = \sum_{i\in I} |t_i|$
- $F_{out}(A,LI_p) = \sum_{j\in J} |t_j|$

$F(A,LI_p)$ is called the offset of A w.r.t. LI_p.

Theorem 5.4: Let $\sum_{i\in I} p_i + c \geq \sum_{j\in J} p_j$ be a valid linear predicate inequality, $G = \leftarrow p(t_1,\ldots,t_n)\sigma$ a well-formed goal, V and W the multisets of relevant input resp. output variable occurrences of $p(t_1,\ldots,t_n)$ and θ a computed answer for G. Then the following holds:

i) $\sum_{i\in I} |t_i\sigma\theta| + c \geq \sum_{j\in J} |t_j\sigma\theta|$.

ii) $\sum_{v\in V} |v\sigma\theta| + F(p(t_1,\ldots,t_n),LI_p \geq \sum_{w\in W} |w\sigma\theta|$.

Proof: According to [FLP89], $p(t_1,\ldots,t_n)\sigma\theta$ is an instance of an atom $p(s_1,\ldots,s_n)$ in the S-model M_S of P. Since the output of G is unrestricted, $t_j\sigma\theta = s_j$ for all $j\in J$. Proposition 5.2 implies $|t_i\sigma\theta| \geq |t_i|$ for all $i\in I$. Thus $\sum_{i\in I} |t_i\sigma\theta| \geq \sum_{i\in I} |s_i|$ and $\sum_{j\in J} |t_j\sigma\theta| = \sum_{j\in J} |s_j|$ which proves the first part of the theorem. The second part is implied by Prop. 5.2. ■

Theorem 5.4 gives a valid inequality relating variables occurring in a single literal goal. Next we give an algorithm for the derivation of a valid inequality relating variables in a compound goal.

Algorithm 5.5 *goal_inequality(G,LI,U,W,Δ,b)*

Input: A well-formed goal $G = \leftarrow B_1,\ldots,B_n$, a set LI with one inequality for each predicate in G, and two multisets U and W of variable occurrences.

Output: A boolean variable b which will be true if a valid inequality relating U and W could be derived, and an integer Δ which is the offset of that inequality.

begin
  M := W; Δ := 0; V := U;
  For i := n **to** 1 **do**:
  If $M \cap V_{out}(B_i,LI_P) \neq \emptyset$ **then**
    $M := (M \setminus V_{out}(B_i,LI_P)) \cup (V_{in}(B_i,LI_P) \setminus V)$;
    $V := V \setminus V_{in}(B_i,LI_P)$;
    $\Delta := \Delta + F(B_i,LI_P)$. **fi**
  If $M = \emptyset$ **then** b:= true **else** b:= false **fi**
end.

Next we show that the algorithm is correct:

Theorem 5.6: Assume that the inequalities in LI are valid and b is true, σ is an arbitrary substitution such that $G\sigma$ is well-formed and θ is a computed answer substitution for $G\theta$. Then $\sum_{v\in V} |v\sigma\theta| + \Delta \geq \sum_{w\in W} |w\sigma\theta|$ holds.

Proof: See [PLU92].

Algorithm 5.5 takes time O(m) where m is the length of G.

[PLU90b] gives an algorithm for the automatic derivation of inequalities for compound goals based on and/or-dataflow graphs which has exponential runtime in the worst case. Algorithm 5.5 makes substantial use of the fact that G is well-formed: each variable has at most one generator, which makes the derivation of inequalities deterministic.

6. Derivation of inequalities for S-models

In Aection 5 it has been assumed that linear inequalities are given for the predicates of a program P. We now show how these inequalities can be derived automatically. We assume that P is well-formed and free of mutual recursion. Let $p <_\pi q$ if $p \neq q$ and p occurs in one of the clauses defining q. Absence of mutual recursion in P implies that $<_\pi$ defines a partial order which can be embedded into a linear order. Thus there is an enumeration $\{p_1,\ldots,p_n\}$ of the predicates of p such that $p_i <_\pi p_j$ implies $i < j$. We will process the predicates of P in that order, thus in analyzing p we can assume that for all predicates on which the definition of p depends valid inequalities have already been derived. Note that a trivial inequality with offset ∞ always holds.

Let in(A) and out(A) denote the sets of input resp. output variables of an atom or a set of atoms according to the annotation of the given programs.

Algorithm 6.1: predicate_inequalities(P,LI):

Input: A well-formed program P defining $p_1,\ldots,p_n$.

Output: A set LI of valid inequalities for the predicates of P.

begin
  LI := ∅
  For i:= 1 to n do:
    begin
      Let $c_1,\ldots,c_m$ be the clauses defining p_i.
      Let M, N be the input resp. output arguments of p_i.
      li := $\sum_{\mu\in M}|p_\mu| + \gamma \geq \sum_{\nu\in N}|p_\nu|$.
      b_i := true.
      For j:= 1 to m do:
      begin
        Let c_j be $B_0 \leftarrow B_1,\ldots,B_k$.
        goal_inequality($(\leftarrow B_1,\ldots,B_k)$,
          LI∪{li},$V_{in}(B_0)$,$V_{out}(B_0)$, Δ_i,b_i)
        c:= $\Delta_i + F_{out}(B_0,li) - F_{in}(B_0,li)$.
        $\Phi_i := b_i$
        If c contains '∞' **then** $\Phi_i := \Phi_i \wedge$ false
(*)   **elseif** c is an integer **then** $\Phi_i := \Phi_i \wedge (\gamma \geq c)$
(**)   **elseif** $c = \gamma + d \wedge d \leq 0$ **then** $\Phi_i := \Phi_i \wedge$ true
        elseif $c = \gamma + d \wedge d > 0$ **then** $\Phi_i := \Phi_i \wedge$ false
(***)   **elseif** $c = k * \gamma + n \wedge k > 1$,
        then $\Phi_i := \Phi_i \wedge (\gamma \leq n/(1-k))$.
      end
      If Φ_i is satisfiable **then** let δ_i be the smallest value for
          γ which satisfies Φ_i
      else let δ_i be '∞'.
      Replace γ in li by δ_i.
      LI := LI ∪ {li}
    end
end

Theorem 6.2: The inequalities derived by the algorithm are valid.

Proof: By induction on the number of predicates n in P. The case n = 0 is immediate. For the inductive case, assume that the derived inequalities for the predicates $p_1,\ldots,p_{n-1}$ are valid. Let I_0 be the minimal S-model of P restricted to the predicates $p_1,\ldots,p_{n-1}$. In the context of the program which consists of the definition of p_n only, let $T_s^0 = I_0$ and $T_s^m = T_s(T_s^{m-1})$. Its limes equals the minimal S-model of P restricted to the predicates $p_1,\ldots,p_n$. Now we have to show that the inequality li derived for p_n is valid w.r.t. T_s^m. The proof is now by induction on m. The case m = 0 is implied by the induction assumption on n. Assume that the theorem holds for n - 1. We have to show that the inequality for p_n holds for the elements of T_s^m. Now lett $B \in T_s^m$ and $B_0 \leftarrow B_1,\ldots,B_k$ be the clause applied to derive B. We have $B = B_0\theta$, where θ is a computed answer substitution for $\leftarrow B_1,\ldots,B_k$, which is a well-formed goal. Let $V = in(B_0)$ and $W = out(B_0)$. Let LI be the set of inequalities derived by Algorithm 6.1, and Δ be the result of calling goal_inequality($(\leftarrow B_1,\ldots,B_k)$,LI,V,W, Δ, b_i). Theorem 5.6 and the induction assumption imply

(‡) $\quad \sum_{v\in V}|v\theta| + \Delta \geq \sum_{w\in W}|w\theta|$

Since $B = B_0\theta$, we have $F_{in}(B,li) = F_{in}(B_0,li) + \sum_{v\in V}|v\theta|$ and $F_{out}(B,li) = F_{out}(B_0,li) + \sum_{w\in W}|w\theta|$. Let α be the offset of li. We have to show

(‡‡) $\quad F_{in}(B,li) + \alpha \geq F_{out}(B,li)$.

If b_i is false or Δ is ∞, we are done since in that case α is ∞. Three more cases remain. (*) and (**) immediately imply

(‡‡‡) $\quad \alpha \geq \Delta + F_{out}(B_0,li) - F_{in}(B_0,li)$.

(***) implies $\alpha \leq n/(1-k)$ and thus $\alpha \geq n + k*\alpha$ for some n such that $n + k*\alpha = \Delta + F_{out}(B_0,li) - F_{in}(B_0,li)$. Again (‡‡‡) follows. (‡) and (‡‡‡) together now imply (‡‡). ■

Note that Algorithm 6.1 again has run-time complexity O(n), where n is the length of the given program P.

Algorithm 6.1 is not yet able to derive $p_1 \geq p_2$ for a unit clause like p(X,Y) with mode(p(+,-)). This inequality, however, holds since in a well-formed goal the output argument of p will always be unbound. To overcome this difficulty, we assume that before calling predicate_inequalities(P,LI), P will be transformed to P' in the following way:

Define freevars($B_0 \leftarrow B_1,\ldots,B_n$) = $(out(B_0) \setminus out(B_1,\ldots,B_n)) \cup in(B_1,\ldots,B_n) \setminus in(B_0))$.
Now for the clause $c = B_0 \leftarrow B_1,\ldots,B_n$ in P let freevars(c) = $\{Y_1,\ldots,Y_m\}$. Replace c by $B_0 \leftarrow q(Y_1,\ldots,Y_m),B_1,\ldots,B_n$ where a new predicate q is defined by the unit clause $q(X_1,\ldots,X_m)$ with mode(q(+,…,+)). Note that, after that transformation, P' is well-formed if P is well-formed, and if an inequality is valid for P' it is valid for P as well. In the example mentioned above, input for Algorithm 6.1 will be the program $P = \{q(X).\ , p(X,Y) \leftarrow q(Y)\}$ and the output will be $\{0 \geq q_1,\ p_1 \geq p_2\}$.

Another improvement can be made by considering subsets of the input arguments in order to achieve stronger inequalities. This, however, makes the algorithm less efficient.

7. Example

We finally discuss how, with the techniques given so far, it can be shown that the GHC program for quicksort specified in Section 3 terminates for arbitrary goals.

Corollary 4.3 and Theorem 4.5 imply that is suffices to consider data-driven LD-derivations of the extended program for qsort including the clauses s_0, a_0 and q_0. According to Theorem 5.1 we only have to show that the three predicates of the program are safe. This is easy to show for split and append. In fact these procedures are structural recursive. It is more difficult to prove of qsort because in q_2 both recursive calls contain the local variables A and B. For this reason we need a linear predicate inequality for split which has the form $split_1 + \gamma \geq split_3 + split_4$. After the transforamtion mentioned at the end of the last paragraph s_0 will have the following form:

s_0: $split(L_1,L_2,L_3,L_4) \leftarrow q(L_3, L_4)$

Now s_0 and s_1 give $\gamma \geq 0$ (case * in Algorithm 6.1), while s_2 and s_3 give 'true' (case **). Thus we get $split_1 + 0 \geq split_3 + split_4$. In order to prove safety of qsort, we only have to consider q_2. Using this inequality Algorithm 5.5 immediately shows $||qsort([H|L],S)\theta|| > ||qsort(A,A_1)\theta||$ and $||qsort([H|L],S)\theta| > ||qsort(B,B_1)\theta||$ for all answer substitutions θ for split(H,L,A,B). Thus qsort is safe.

Acknowledgment

Part of this work was performed while I was visiting CWI. K. R. Apt stimulated my interest in concurrent logic programming.

References

[APP90] Apt, K. R., Pedreschi, D., Studies in pure Prolog: Termination, Technical Report CS-R9048, Centre for Mathematics and Computer Science, Amsterdam, 1990.

[APT90] Apt, K. R., Introduction to logic programming, in Leeuwen (ed.), *The Handbook of Theoretical Computer Science*, North Holland 1990.

[BCF90] Bossi, A., Cocco, N., Fabris, M., Proving Termination of Logic Programs by Exploiting Term Properties, Technical Report Dip. di Matematica Pura e Applicata, Universita di Padova, 1990.

[FAL88] Falaschi, M., Levi, G., Finite failures and partial computations in concurrent logic languages, *Proc. of the Int. Conf. of Fifth Gen. Comp. Systems*, ICOT 1988.

[FLP89] Falaschi, M., Levi, G., Palamidessi, C., Martelli, M., Declarative Modeling of the Operational Behavior of Logic Languages, *Theoretical Computer Science 69*, 1989.

[GRE87] Gregory, S., Parallel Logic Programming in PARLOG, Addison Wesley, 1987.

[HAP85] Harel, D., Pnueli, A., On the development of reactive systems, in Apt, K. R. (ed.) *Logics and Models of Concurrent Systems*, Springer 1985.

[LLO87] Lloyd, J., Foundations of Logic Programming, Springer Verlag, Berlin, second edition, 1987.

[PLU90a] Plümer, L., Termination proofs for logic programs based on predicate inequalities, in Warren, D.H.D., Szeredi, P. (eds.), *Proceedings of the Seventh International Conference on Logic Programming*, MIT Press 1990.

[PLU90b] Plümer, L., Termination Proofs for Logic Programs, Springer Lecture Notes in Artificial Intelligence 446, Berlin 1990.

[PLU91] Plümer, L., Termination proofs for Prolog programs operating on nonground terms, *1991 International Logic Programming Symposium, San Diego, California*, 1991.

[PLU92] Plümer, L., Automatic Verification of GHC-Programs: Termination, Technical Report, Universität Bonn, 1992.

[SHA87] Shapiro, E., Concurrent Prolog, Collected Papers, MIT Press 1987.

[SHA87a] Shapiro, E., Systolic Programming: A paradigm of parallel processing, in [SHA87].

[TIC91] Tick, E., Parallel Logic Programming, MIT Press 1991.

[UED88] Ueda, K., Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard, in Nivat, M., Fuchi, K. (eds.), *Programming of Future Generation Computers*, North-Holland 1988.

[UED86] Ueda, K., Guarded Horn Clauses, in [SHA87].

[ULG88] Ullman, J. D., van Gelder, A., Efficient Tests for Top-Down Termination of Logical Rules, *Journal of the ACM* 35, 2, 1988.