New Generation Computing, 15 (1997) 293-338
OHMSHA, LTD. and Springer-Verlag

Proving Termination of GHC Programs*

M. R. K. KRISHNA RAO**
Computer Science Group,
Tata Institute of Fundamental Research,
Homi Bhabha Road, Colaba, Bombay 400 005, India.
D. KAPUR***
Department of Computer Science,
SUNY at Albany, NY 12222, USA.
R. K. SHYAMASUNDAR
Computer Science Group,
Tata Institute of Fundamental Research,
Homi Bhabha Road, Colaba, Bombay 400 005, India.

Received 30 March 1995
Revised manuscript received 27 February 1997

Abstract A transformational approach for proving termination of parallel logic programs such as GHC programs is proposed. A transformation from GHC programs to term rewriting systems is developed; it exploits the fact that unifications in GHC-resolution correspond to matchings. The termination of a GHC program for a class of queries is implied by the termination of the resulting rewrite system. This approach facilitates the applicability of a wide range of termination techniques developed for rewrite systems in proving termination of GHC programs. The method consists of three steps: (a) deriving moding information from a given GHC program, (b) transforming the GHC program into a term rewriting system using the moding information, and finally (c) proving termination of the resulting rewrite system. Using this method, the termination of many benchmark GHC programs such as quick-sort, merge-sort, merge, split, fair-split and append, etc., can be proved.

Keywords: Parallel Logic Programming, Verification, Termination

* This is a revised and extended version of Ref. 12). The work was partially supported by the NSF Indo-US grant INT-9416687.

** Address for correspondence: School of Computing and Information Technology, Faculty of Science and Technology, Griffith University, Nathan, Brisbane, Australia, 4111. e-mail: **krishna@cit.gu.edu.au**

*** Kapur was partially supported by NSF Grant nos. CCR-8906678 and INT-9014074.

§1 Introduction

The success of logic programming in the recent times is often attributed to its simple and elegant declarative semantics and the clean separation of logic and control aspects (which corresponds to the view point: *Algorithm* = *Logic* + *Control*). Owing to the clean separation of logic and control, a program can be written declaratively by the programmer and executed procedurally by the computer. Yet another attraction of logic programming is the notion of *logic variables* that leads to nice implementations of communication protocols (like remote procedure calls, dynamic process migration etc.), and makes logic programs particulary suitable for concurrent programming.[9)]

A logic program is a finite set of universally quantified Horn clauses. A goal statement is used to invoke a computation, which can be regarded as a proof of the goal statement from the given set of clauses. Many proof strategies can be used for proving the goal from the program; among these, Prolog's left-to-right selection strategy and parallel strategies are of great interest. The latter corresponds to interpretation of logic programs under parallel control. There has been a great amount of research on this topic and several parallel logic programming languages have been proposed in the last decade.

The main motivation for designing parallel languages is to fully utilize the fast developing hardware architectures. Because of its semantic clarity and inherent parallelism, logic programming paradigm is regarded as a good candidate for fully utilizing the power of parallel architectures. The reader is referred to Ref. 21) for an extensive discussion on several issues in parallel logic programming and Ref. 23) for a brief introduction to parallel logic programming languages.

As in the case of Prolog programs, the issue of termination plays an important role for parallel (concurrent) logic programs both from the point of view of reasonig about programs and developing sound design methodologies for parallel (concurrent) logic programming. In fact, the study of termination of parallel logic programs in a systematic way is vital since some issues of the parallel-control strategies make termination detection quite complicated. For example, a program containing the following clause

$$p(f(X)) \leftarrow p(X)$$

has an infinite SLD-derivation starting with query $\leftarrow p(Y)$. However when the same program is considered as a *Guarded Horn Clause* (GHC) program, it has no infinite GHC-derivations starting with any query. Though termination of Horn clause programs is well-studied in the literature (see Ref. 2) for a comprehensive survey), there are not many attempts towards studying termination of concurrent (parallel) logic programs; to the best of our knowledge, Refs. 19) and 12) seem to be the only published works on this subject.

In this paper, we propose a method for proving termination of parallel

logic programs, in particular *Guarded Horn Clauses*. The method reduces the termination problem of GHC programs to that of term rewriting systems, similar to the approach taken in our earlier papers[10,22] on termination of logic programs with sequential control. Since the termination problem of term rewriting systems has been extensively studied in the literature (see Ref. 4) for a survey) and several mechanical systems (e.g. Refs. 8), 13) and 14)) and heuristics are available for checking termination of term rewriting systems, we can readily apply those results and use rewrite-rule based theorem provers for proving termination of GHC programs.

The transformation involves a data-flow analysis of parallel logic programs and uses input-output moding information of predicates. To keep track of the complicated data-flow in parallel computations, we enforce a discipline by placing some syntactic conditions on the programs. This discipline in fact supports the style advocated by Ueda while introducing the Guarded Horn Clause programming. We first develop the transformation with an assumption that moding information is given, and subsequently, we show how the moding information can be derived from a given GHC program. We show that a given GHC program terminates for a class of queries if the derived rewrite system is terminating.

We do not make any assumptions about the relative speeds of the processes or the fairness of their scheduling. Termination of the derived rewrite system implies termination of the original GHC program under every possible scheduling irrespective of the relative speeds of various processes. As can be expected from such strong implications of the termination of the derived rewrite system, our result gives only a sufficient condition for termination of GHC programs. That is, the rewrite systems derived from some terminating GHC programs are nonterminating, and hence, our results cannot be used in proving each and every terminating GHC program. However, our method applies to a large class of interesting programs; further, the practicality of the method is evident from the fact that the termination of a prototype compiler for **ProCoS** language PL_0 can be established using our tool,[11] while it is beyond the scope of the other known tools for proving termination of logic programs.

The above results extend to other parallel (or concurrent) logic programming languages discussed in Ref. 20). In particular, the results hold for *safe* parallel logic programming languages identified by Takeuchi and Furukawa,[23] in which instantiations of variables in a goal are not allowed during head unification and evaluation of guards of a clause.

The rest of the paper is organized as follows. The next section briefly introduces GHC programming and Section 3 gives definitions of well-moded and weakly well-moded GHC programs and discusses the assumptions made. Termination of GHC programs is related to termination of term rewrite systems in Sections 4 and 5. Section 6 discusses the automatic derivation of moding information from GHC programs, and Section 7 describes the extension of the

results for other parallel logic programming languages. The paper concludes with a discussion and comparison with related works.

§2 Preliminaries

2.1 Guarded Horn Clause (GHC) Programs

Languages in the family of parallel logic programming languages share many common features such as syntax, declarative meaning etc. and mainly differ in the synchronization mechanism used in synchronizing the And-processes. Guarded Horn Clause programming proposed by Ueda[24] is one of the elegant and pragmatic models proposed for parallel logic programming. In this paper, we mainly deal with termination of GHC programs, and briefly sketch how our techniques can be used for other languages. We provide a brief introduction to the GHC language, and its synchronization mechanism (mainly, the rules of suspension and commitment) in the following.

Definition 1 (GHC programs)

(1) A Guarded Horn Clause (GHC) program[24] is a finite set of Guarded Horn Clauses of the form:

$$H \leftarrow G_1, \ldots, G_m \mid B_1, \ldots, B_n \qquad m \geq 0,\ n \geq 0$$

where H (head), G_i's (guard atoms) and B_i's (body atoms) are atomic formulae.

(2) The part of a guarded clause before the *commitment operator* '|' is called the *guard* and the part after '|' is called the *body*.
Note that the clause head is included in the guard.

(3) A goal clause has the form $\leftarrow Q_1, \ldots, Q_n$, $n \geq 0$.

A clause with an empty body is called *unit* clause. The set of all clauses (in a program) whose heads have the predicate symbol p is called the *procedure* for p. A subgoal with the predicate symbol p is said to *call* p. Further, a binary predicate '=' is predefined in GHC and is used for unification of two terms (its two inputs). Atoms with predicate '=' in the body of a clause are called unification atoms.

A clause $H \leftarrow G_1, \ldots, G_m \mid B_1, \ldots, B_n$ is declaratively 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 the *parallel input resolution* governed by the following *rules of suspension* and *rule of commitment* described in Ref. 24).

- **Rules of suspension**

  (1) Any piece of unification invoked directly or indirectly in the guard of a clause cannot bind a variable appearing in the caller of that clause with

(i) a non-variable term or
(ii) another variable appearing in the caller.

(2) Any piece of unification invoked directly or indirectly in the body of a clause cannot bind a variable appearing in the guard of that clause with
(i) a non-variable term or
(ii) another variable appearing in the guard.
until that clause is selected for commitment.

Any piece of unification which can succeed only by making such bindings is *suspended* until it can succeed without making such bindings. The first rule essentially says that the guard of a clause cannot export any bindings to its caller. The guard can be seen as a precondition for the application of the clause. This rule is used for synchronization. The second rule says that a clause cannot export any bindings to the guard of that clause before commitment, i.e., the body can be concurrently executed with the guard as long as it does not bind any variable in the guard (including the head) before commitment.

Rule of commitment
When some clause succeeds in solving its guard for a given goal, that clause tries to get selected exclusively for subsequent execution of the goal. To be selected, the clause must first confirm that no other clause belonging to the same procedure has been selected for the same goal. If confirmed, the clause is selected indivisibly, and we say that the goal is committed to the clause (or the clause is selected for commitment).[24)]

As usual, a goal is said to *succeed* when it reduces to an empty goal.

Under these rules, many things can be done in parallel: conjunctive goals can be executed in parallel; candidate clauses for a goal can be tested in parallel; head unification involved in resolution can be done in parallel; head unification and the execution of guard atoms can be done in parallel.

The following example (from Ref. 24)) illustrates the rules of suspension and commitment.

Example 1
Consider the following GHC program and goal.

```
Goal:      ← p(X), q(X)
Clauses:   p(a) ← true | ...        (i)
           q(Z) ← true | Z = a      (ii)
```

Clause (i) cannot instantiate the argument X of its caller to constant a, since this unification is executed in the guard. This clause has to wait until X is instantiated to a by some other goal. On the other hand, clause (ii) can instantiate X to a after it is selected for *commitment*, and it can be selected almost immediately

(guard is *true* and succeeds). □

Two important consequences of the above rules of suspension and commitment are:

(a) Unifications involved in the parallel input resolution are essentially *matchings* (i.e. the unifier of the goal A and the head of a clause C can only instantiate the variables in the head of C but not the variables in A).

(b) Any unification intended to export bindings to the caller (goal) of a clause through its head arguments must be specified in the body using the predefined predicate '=' (cf. Ref. 24)).

The above consequences, particularly (a), makes our transformational approach very suitable for proving termination of GHC programs. Before discussing the termination issues, we define GHC derivations.

Definition 2 (GHC derivations)

(1) Let $G_i = \leftarrow A_1, ..., A_n$ be a GHC goal such that

(a) there is an atom (say, A_k) in G_i which matches with the head of a clause c:

$H \leftarrow guard \mid B_1, ..., B_m$

with a matching substitution σ (i.e., $A_k \equiv H\sigma$) and $(guard)\sigma$ succeeds with answer substitution θ without instantiating any variable in A_k (i.e., $A_k\theta \equiv A_k$). Then, the goal derived from G_i and c is $G_{i+1} = \leftarrow A_1, ..., A_{k-1}, B_1\sigma\theta, ..., B_m\sigma\theta, A_{k+1}, ..., A_n$.

If there are many clauses satisfying the above requirements (i.e., eligible for commitment), one clause among them is chosen nondeterministically.

OR

(b) an atom (say, A_k) in G_i is a unification atom of the form $t_1 = t_2$ such that t_1 and t_2 unify with an mgu σ. Then, the goal derived from G_i is $G_{i+1} = \leftarrow A_1\sigma, ..., A_{k-1}\sigma, A_{k+1}\sigma, ..., A_n\sigma$.

Derivation of G_{i+1} from G_i is called a GHC resolution (or a single derivation) step.*

(2) A GHC derivation of $P \cup \{G_0\}$ is a sequence $G_0, G_1, ..., G_n$ of goals such that for each $i \in [1, n]$, G_i is derived from G_{i-1} through a GHC derivation step, and all the clauses used in the derivation are from P.

[1] Flat GHC

Flat GHC, a subset of the GHC language has been found to be adequate

* Actually, GHC allows parallel reduction of atoms; that is, if there are multiple atoms in G_i which match with the heads of clauses such that the corresponding guards succeed, all of them are allowed to be reduced simultaneously. However, our syntactic conditions ensure that the parallelism is captured through all possible interleavings. Therefore, we only consider the derivation steps where only one atom (without any fixed selection rule) is reduced.

for most of the applications, and can be easily implemented as compared to the full GHC language.[26]

Definition 3
A predicate p is called a *test predicate* if all the clauses defining p are unit clauses.

Since unit clauses do not contain any unification subgoals, calls to test predicates do not instantiate any variables.

Definition 4
A GHC clause is *flat* if its guard atoms are restricted to unification atoms and atoms with test predicates. A *Flat GHC program* is a set of flat GHC clauses.

For convenience, we denote the multiset of unification atoms in the guard by G_U and the multiset of other atoms by G_N. Similarly, we divide the body into B_U and B_N.

Definition 5
A Flat GHC clause $Head \leftarrow G_U \cup G_N \mid B_U \cup B_N$ is said to be in *normal form* (or, *is normalized*) if $G_U = \phi$ and B_U is of the form $X_1 = t_1, \ldots, X_n = t_n$ such that (i) each $X_i \in Var(Head)$ and (ii) X_i's do not occur on the right side of the equations.

Ueda and Furukawa[25] showed that any given (Flat) GHC program can be easily transformed into an equivalent *normalized* (Flat) GHC program. In the sequel, we only consider *normalized* programs.

2.2 Term Rewriting Systems

In this subsection, we briefly explain the basic concepts of term rewriting systems.

Definition 6
A *term rewriting system* (TRS, for short) $\mathcal{R}$ is a pair $(\mathcal{F}, R)$ consisting of a set $\mathcal{F}$ of function symbols and a set R of rewrite rules of the form $l \rightarrow r$ satisfying:

(1) $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, the set of terms built from functions in $\mathcal{F}$ and variables in $\mathcal{X}$,
(2) left-hand-side l is not a variable and
(3) $Var(r) \subseteq Var(l)$.

A rule $l \rightarrow r$ applies to term t in $\mathcal{T}(\mathcal{F}, \mathcal{X})$, if a subterm s of t matches with l through some substitution σ, i.e., $s \equiv l\sigma$, and the rule is applied by replacing the subterm s in t by $r\sigma$ resulting in a new term u. This is formalized in the following definitions.

Definition 7
A *context* $C[,\ldots,]$ is term in $\mathcal{T}(\mathcal{F} \cup \{\Box\}, \mathcal{X})$. If $C[,\ldots,]$ is a context containing

n occurrences of $\Box$ and $t_1, \ldots, t_n$ are terms then $C[t_1, \ldots, t_n]$ is the result of replacing the occurrences of $\Box$ from left to right by $t_1, \ldots, t_n$. A context containing precisely 1 occurrence of $\Box$ is denoted $C[]$.

Definition 8
The *rewrite relation* $\Rightarrow_{\mathcal{R}}$ induced by a TRS $\mathcal{R}$ is defined as follows: $s \Rightarrow_{\mathcal{R}} t$ if there is a rewrite rule $l \rightarrow r$ in $\mathcal{R}$, a substitution σ and a context $C[]$ such that $s \equiv C[l\sigma]$ and $t \equiv C[r\sigma]$.
We say that s reduces to t in *one rewrite* (or reduction) *step* if $s \Rightarrow_{\mathcal{R}} t$ and say s *reduces to* t if $s \Rightarrow^*_{\mathcal{R}} t$, where $\Rightarrow^*_{\mathcal{R}}$ is the reflexive-transitive closure of $\Rightarrow_{\mathcal{R}}$.

Definition 9
A term rewriting system $\mathcal{R}$ is *terminating* if there is no infinite rewriting derivation $t_1 \Rightarrow_{\mathcal{R}} t_2 \Rightarrow_{\mathcal{R}} t_3 \Rightarrow_{\mathcal{R}} \ldots$.

§3 Well-Moded GHC Programs

In this section, we introduce the notion of well-moded GHC programs and explain the assumptions made on well-moded programs for studying their termination.

Definition 10
A *mode* m of an n-ary predicate p is a function from $\{1, \ldots, n\}$ to the set $\{in, out\}$. The sets $\{i \mid m(i) = in\}$ and $\{o \mid m(o) = out\}$ are the set of input and output positions of p respectively.
Notation: The terms $invar(A)$ and $outvar(A)$ denote the sets of variables occurring in the input and output terms of atom A respectively.

Definition 11
Let c be a Guarded Horn Clause $A \leftarrow B_1, \ldots, B_k \mid B_{k+1}, \ldots, B_n$ and X be a variable in c. Atom B_i, $1 \leq i \leq n$, is a *consumer* (*producer*) of X if $X \in invar(B_i)$ ($X \in outvar(B_i)$), respectively).

Definition 12
The producer-consumer relation of a clause c: $A \leftarrow B_1, \ldots, B_k \mid B_{k+1}, \ldots, B_n$ is defined as $\{\langle B_i, B_j \rangle \mid B_i$ is a producer of a variable X in c and B_j is a consumer of $X\}$.

Definition 13
A GHC clause c is *well-moded* if (a) its producer-consumer relation is *acyclic*, and (b) every variable in $Var(c) - invar(head)$ has at least one producer. A program P is *well-moded* if every clause in it is well-moded. A *well-moded query* is a well-moded clause without a head.

Definition 14
A GHC clause c is *weakly well-moded* if its producer-consumer relation is *acyclic*. A program P is *weakly well-moded* if every clause in it is weakly

well-moded. A *weakly well-moded query* is a weakly well-moded clause without a head.

Example 2
Consider the following quick-sort program.

moding: match(out, in); q(in, out); s(in, in, out, out) and a(in, in, out)

1. $q(nil, S) \leftarrow true \mid match(S, nil)$
2. $q(c(H, L), S) \leftarrow true \mid s(L, H, A, B), q(A, A1), q(B, B1), a(A1, c(H, B1), S)$
3. $s(nil, Y, L_1, L_2) \leftarrow true \mid match(L_1, nil), match(L_2, nil)$
4. $s(c(X, Xs), Y, L_1, L_2) \leftarrow X \leq Y \mid match(L_1, c(X, L_1')), s(Xs, Y, L_1', L_2)$
5. $s(c(X, Xs), Y, L_1, L_2) \leftarrow X > Y \mid match(L_2, c(X, L_2')), s(Xs, Y, L_1, L_2')$
6. $a(nil, X, Z) \leftarrow true \mid match(Z, X)$
7. $a(c(H, X), Y, Z) \leftarrow true \mid match(Z, c(H, Z')), a(X, Y, Z')$

The producer-consumer relations of clauses 1, 3 and 6 are empty. For clause 2, it is given by $\{\langle s(L, H, A, B), q(A, A1)\rangle, \langle s(L, H, A, B), q(B, B1)\rangle, \langle q(A, A1), a(A1, c(H, B1), S)\rangle, \langle q(B, B1), a(A1, c(H, B1), S)\rangle\}$. For clauses 4, 5 and 7, it is $\{\langle s(Xs, Y, L_1', L_2), match(L_1, c(X, L_1'))\rangle\}$, $\{\langle s(Xs, Y, L_1, L_2'), match(L_2, c(X, L_2'))\rangle\}$ and $\{\langle (a(X, Y, Z'), match(Z, c(H, Z'))\rangle\}$, respectively. It is easy to see that every variable (which is not occurring in the input terms of the head) in each clause has a producer and the producer-consumer relation of each clause is acyclic. So, the quick-sort program is well-moded. □

The transformation procedure presented in a later section is based on the data-flow analysis of GHC programs. To keep track of the data-flow (in particular, the 'direction' of the data-flow) information, we enforce the restriction that "input variables of the goal do not get instantiated in the execution and only output variables get instantiated (possibly, their bindings involve input variables)" and assume that the unification goals in each clause are either (i) matchings (i.e., one-sided unification) or (ii) syntactic identity. Correspondingly, we use two predefined predicates '= =' (equality check, with modimg (in, in)) and '⇐' (matching, with moding (out, in)) instead of '=' (unify, with moding (out, out)) introduced in the original definition of GHC. For notational convenience, we write $match(t_1, t_2)$ instead of $t_1 \Leftarrow t_2$ in our GHC programs, and it stands for matching t_1 with t_2, i.e., finding σ such that $t_1\sigma = t_2$. It is of interest to note that the above restriction in fact supports the style advocated by Ueda.[24] To quote

> In most cases, bi-(or multi-)directionality of a logic program is only an *illusion*; it seems far better to specify the data flow which we have in mind and to enable us to read it from a given program.[24]

Formally, the above restriction that "input variables of the goal do not get

instantiated in GHC derivations" is enforced through the following two assumptions on GHC programs.

A1 Input (output) variables of the head do not occur in the output (input respectively) positions of the body/guard atoms in the clause.

A2 No variable occurs in more than one output position in the guard/body.

It is not difficult to see that input variables of a goal can get instantiated during the execution if input variables of the head occurs in output positions of the guard/body of a clause in the program, i.e., the above assumption A1 is naturally needed to enforce the above restriction. However, it is not straghtforward to see that assumption A2 is also needed and assumption A1 alone is not sufficient to enforce the above restriction. This fact is illustrated through the following example.

Example 3

Consider the following well-moded GHC program violating assumption A2.

moding: p(in, out); s(in, out); q(in, out); q_1(in, out) and q_2(in, out)

p(f(X), Y) ← true | s(f(X), U1), q(U1, U2), p(U1, Y)
q(X, Y) ← true | q_1(X, Y), q_2(X, Y)
q_1(X, Y) ← true | match(Y, X)
q_2(X, Y) ← true | match(Y, f(b))
s(f(a), Y) ← true | match(Y, a)

The following GHC derivation instantiates input variable U1 of the initial goal.

← q(U1, U2), p(U1, Y)
← q_1(U1, U2), q_2(U1, U2), p(U1, Y)
← match(U2, U1), q_2(U1, U2), p(U1, Y)
← q_2(U1, U1), p(U1, Y)
← match(U1, f(b)), p(U1, Y)
← p(f(b), Y)

The execution of subgoal q_1(U1, U2) instantiates the output variable U2 to U1. Now, U1 occurs in both input and output positions of the subgoal q_2(U1, U1) and gets instantiated to f(b) in the execution of q_2(U1, U1). □

Assumptions A1 and A2 described above allow us to study termination of non-well-moded GHC programs. In particular, they allow us to relax the requirement (of well-modedness) that every variable in the clause has at least one producer and even allows a kind of cycles in the producer-consumer relation. Now, we formally prove that input variables of a goal satisfying A2 do not get instantiated in any GHC derivation of a weakly well-moded GHC program satisfying A1 and A2.

Definition 15
A weakly well-moded flat GHC program satisfying assumptions A1 and A2 is called a *T-program* (a program for which the transformational approach can be applied straightaway). We also call $P \cup \{Q\}$, a weakly well-moded program.

Theorem 1
Let P be a T-program, Q be a weakly well-moded query satisfying A2 and X be a variable in Q that does not occur in output terms of Q. Then, each goal in a GHC derivation of $P \cup \{Q\}$ (1) is weakly well-moded, (2) satisfies assumption A2, and (3) does not contain X in output terms.

Proof
Induction on length l of the derivation.*
Basis: $l = 1$. The derivation has just one goal (i.e., Q), which satisfies all the requirements.
Induction hypothesis: Assume that the theorem holds for all derivations of length $l < m$.
Induction step: Now, we prove that the theorem holds for all derivations of length $l = m$. Let $G_{m-1} \equiv \leftarrow A_1, ..., A_i, ..., A_n$ be the last but one goal in the derivation and A_i be the atom reduced in the m-th step. By induction hypothesis, G_{m-1} satisfies all the requirements. We have two cases: (a) A_i is a match atom or (b) A_i is not a match atom.

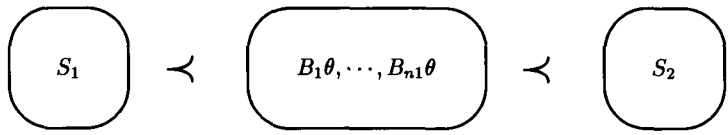
Case (a): A_i is of the form match(t_1, t_2). Let $X_1, ..., X_k$ be variables in t_1. The result of this match atom is a substitution σ with $domain(\sigma) = \{X_1, ..., X_k\}$ and $G_m \equiv \leftarrow A_1\sigma, ..., A_{i-1}\sigma, A_{i+1}\sigma, ..., A_n\sigma$. By assumption A2, variables $X_1, ..., X_k$ can only occur in input positions of atoms $A_1, ..., A_{i-1}, A_{i+1}, ..., A_n$. Hence, σ can only instantiate input terms of these atoms. Therefore, it follows from the induction hypothesis that G_m satisfies assumption A2 and the variable X does not occur in output positions of G_m.
Now, we prove that G_m is weakly well-moded. Since G_{m-1} is weakly well-moded (by induction hypothesis), its producer-consumer relation induces a partial order (say, $<$) on aotms in G_{m-1} such that $A_j < A_{j'}$ if a variable occurs in input terms of $A_{j'}$ and output terms of A_j. Now, we show that the producer-consumer relation of G_m is a sub-relation of $<$ and thus establish that G_m is weakly well-moded. Let S be the set of atoms in G_{m-1} having some of the variables X_1, ..., X_k occurring in input terms. Only those atoms in S get instantiated by σ and $A_i < A_{j'}$ for each $A_{j'} \in S$. No atom in S is a producer of a variable in t_2 (otherwise, the producer-consumer relation of G_{m-1} would be acyclic). Therefore, new pairs in the producer-consumer relation will be of the form $\langle A_j, A_{j'}\sigma \rangle$ such that (i) $A_{j'} \in S$ (ii) $A_j \notin S$ and (iii) A_j is a producer of a variable in t_2. By (iii), it follows that $A_j < A_i$ and since $A_i < A_{j'}$, it follows that $A_j < A_{j'}$.

* Here, by length, we mean the number of resolution steps and executions of match atoms.

Case (b): A_i is not a match atom. If $H \leftarrow guard \mid B_1, ..., B_{n1}$ is the clause applied on G_{m-1}, then $G_m \equiv \leftarrow A_1, ..., A_{i-1}, B_1\theta, ..., B_{n1}\theta, A_{i+1}, ..., A_n$, where θ is the matching substitution of A_i and H, i.e., $A_i \equiv H\theta$. It follows from induction hypothesis that no variable occurs in two different output terms of $A_1, ..., A_i, ..., A_n$. Since no varialbe occurs in two different output terms of A_i (by assumption A2) and output variables of the body atoms do not occur in input terms of H (by assumption A1), no variable can occur in two output terms in $B_1\theta, ..., B_{n1}\theta$. It is obvious that no variable occurring in output terms of $A_1, ..., A_{i-1}, A_{i+1}, ..., A_n$ occurs in output terms of $B_1\theta, ..., B_{n1}\theta$. Hence, G_m satisfies assunption A2. By induction hypothesis, X does not occur in output positions of $A_1, ..., A_i, ..., A_n$. If X occurs in input terms of A_i, assumption A1 (no input variable of H occurs in output positions of $B_1, ..., B_{n1}$) ensure that X does not occur in output terms of $B_1\theta, ..., B_{n1}\theta$.

Now, we prove that G_m is weakly well-moded. Let $X1$ (and $X2$) be the set of variables occurring in input (and *output* respectively) terms of A_i. Since G_{m-1} is weakly well-moded (by inductyion hypothesis), $X1 \cap X2 = \phi$. Let $\prec$ be the partial order induced by the producer-consumer relation of G_{m-1} and S_1 (and S_2) be the atoms having variables from $X1$ (and $X2$) in output (and input respectively) terms. It is easy to see that the following facts hold.

- $A_j \prec A_i$ and $A_i \prec A_k$ for each $A_j \in S_1$ and $A_k \in S_2$.
  Since G_{m-1} is weakly well-moded, $A_k \not\prec A_j$ for each $A_j \in S_1$ and $A_k \in S_2$.
- The new pairs in the producer-consumer relation are of the form (i) $\langle B_{i_1}\theta, B_{i_2}\theta\rangle$, (ii) $\langle B_{i_1}\theta, A_k\rangle$ and (iii) $\langle A_j, B_{i_2}\theta\rangle$ where $A_j \in S_1$ and $A_k \in S_2$.
- There will be no pair of the form $\langle B_{i_1}\theta, A_j\rangle$, $A_j \in S_1$ since no input variable of S_1 occurs in output terms of A_i and no input variable of A_i occurs in output terms of $B_{i_1}\theta$ (by assumption A1).
- There will be no pair of the form $\langle A_k, B_{i_2}\theta\rangle$, $A_k \in S_2$ since no output variable of S_2 occurs in input terms of A_i and due to assumption A1.



From the above diagram, it is easy to see that the producer-consumer relation of G_m is acyclic. This completes the proof. □

The following theorem establishes that input variables of a goal do not get instantiated in any GHC derivation of a T-program.

Theorem 2
Let P be a T-program and Q be a weakly well-moded query satisfying A2. If X is a variable in Q which does not occur in output terms of Q then X will not get instantiated in any GHC derivation of $P \cup \{Q\}$.

Proof
A variable gets instantiated in a GHC derivation only when a match atom containing that variable in output position is executed. By the above theorem, X will never occur in output position of any match atom and hence will not get instantiated. □

The following two examples show that both assumptions A1 and A2 are needed to preserve weak well-modedness of goals.

Example 4
Consider the following weakly well-moded GHC program violating assumption A1.

moding: p(in, out); q(in, in, out); q_1(in, out) and q_2(in, out, out)

p(X, Y) ← true | ...
q(X1, X2, Y) ← true | q_1(X2, U), q_2(U, X1, Y)
q_1(X, Y) ← true | ...
q_2(X, Y1, Y2) ← true | ...

The following GHC derivation starting from a weakly well-moded query has a query which is not weakly well-moded — note the cycle $\langle p(X1, X2), q_1(X2, U)\rangle \langle q_1(X2, U), q_2(U, X1, Y)\rangle \langle q_2(U, X1, Y), p(X1, X2)\rangle$ in the producer-consumer relation of the second query.

← p(X1, X2), q(X1, X2, Y)
← p(X1, X2), q_1(X2, U), q_2(U, X1, Y)
⋮

Example 5
Consider the following weakly well-moded GHC program violating assumption A2.

moding: q(in, out) and p(in, out) for each p ∈ {q_1, q_2, q_3, q_4}

q(X, Z2) ← true | q_1(X, Y), q_2(Y, Z1), q_3(Z1, Z2), q_4(Y, Z2)
q_1(X, Y) ← true | ...
q_2(X, Y) ← true | ...
q_3(X, Y) ← true | ...
q_4(X, Y) ← true | match(Y, X)

The following GHC derivation starting from a weakly well-moded query has a query which is not weakly well-moded — note the cycle in the producer-consumer relation of the third query, i.e., q_2(Y, Z1) is a producer of Z1 and

consumer of Y and $q_3(Z1, Y)$ is a producer of Y and consumer of Z1.

$\leftarrow q(X, Z2)$
$\leftarrow q_1(X, Y), q_2(Y, Z1), q_3(Z1, Z2), q_4(Y, Z2)$
$\leftarrow q_1(X, Y), q_2(Y, Z1), q_3(Z1, Y)$
$\vdots$

3.1 Weakly Well-Moded and Well-Moded GHC Programs

The transformation procedure presented in the next section works for well-moded programs. The property that every variable has at least one producer in well-moded clauses (which is not true for weakly well-moded clauses) is essential to get rewrite rules satisfying the property $Var(rhs) \subseteq Var(lhs)$, which is assumed by almost all the termination techniques of term rewriting systems. In the following, we study termination properties of a given weakly well-moded program P by associating a well-moded GHC program P' obtained by replacing every local variable without any producer by a fresh constant symbol $\diamondsuit$.

Example 6

Consider the following statement: '*If some one has a property X* (*a variable, standing for any property*), *then Almighty* (*a constant*) *certainly has that property, and every one has some intelligence*'. This can be translated to the following weakly well-moded GHC program.

moding: has(in, out)

has(Almighty, P) $\leftarrow$ true | has(Y, P)
has(X, Z) $\leftarrow$ true | match(Z, intelligence)

We associate the following well-moded program with this weakly well-moded program.

has(Almighty, P) $\leftarrow$ true | has($\diamondsuit$, P)
has(X, Z) $\leftarrow$ true | match(Z, intelligence) □

The following theorem relates weakly well-moded GHC programs and their associated well-moded programs.

Theorem 3

Let P be a weakly well-moded flat GHC program satisfying assumptions A1 and A2, P' be its associated well-moded GHC program and G_0 be a weakly well-moded query satisfying A2. Then, for every GHC derivation $G_0, G_1, ..., G_n$ of $P \cup \{G_0\}$, there is a GHC derivation $G_0, G_1', ..., G_n'$ of $P' \cup \{G_0\}$ such that $G_i' \equiv G_i\sigma_i$, where σ_i is a substitution of the form $\{X_1/\diamondsuit, ..., X_{k_i}/\diamondsuit\}$ and $\{X_1, ..., X_{k_i}\}$ is a subset of the set of variables in G_i not occurring in output positions.

Proof

Induction on n.

Basis: $n = 0$. There is nothing to prove.
Induction hypothesis: Assume that the theorem holds for all $n < m$.
Induction step: Now, we prove the theorem for $n = m$. Let $G_{m-1} \equiv \leftarrow A_1, ..., A_i, ..., A_l$ be the last but one goal in the GHC derivation, and A_i be the atom reduced in m-th step. By induction hypothesis, $G'_{m-1} \equiv G_{m-1}\sigma_{m-1}$ is the corresponding goal in the GHC derivation of $P' \cup \{G_0\}$. There are two cases: (a) A_i is a match atom or (b) A_i is not a match atom.

Case (a): A_i is of the form match(t_1, t_2). Let θ be the resulting substitution of this match atom. Then, $G_m \equiv \leftarrow A_1\theta, ..., A_{i-1}\theta, A_{i+1}\theta, ..., A_n\theta$. Since σ_{m-1} replaces only local variables not occurring in output positions, $A_i\sigma_{m-1} \equiv$ match(t_1, $t_2\sigma_{m-1}$). It is easy to see that $\theta\sigma_{m-1}$ is the resulting substitution of the match atom match(t_1, $t_2\sigma_{m-1}$), and $G'_m \equiv G_m\sigma_m$ where $\sigma_m = \sigma_{m-1}$. By Theorem 2, the variables which do not occur in output terms of G_{m-1} do not occur in output terms of G_m, and hence, σ_m replaces variables not occurring in output terms of G_m.

Case (b): A_i is not a match atom. If $H \leftarrow guard \mid B_1, ..., B_{n1}$ is the clause applied on G_{m-1}, then $G_m \equiv \leftarrow A_1, ..., A_{i-1}, B_1\theta, ..., B_{n1}\theta, A_{i+1}, ..., A_n$, where θ is the matching substitution of A_i and H, i.e., $A_i \equiv H\theta$. Corresponding to this clause, we have $H \leftarrow guard \mid B_1\sigma, ..., B_{n1}\sigma$ in P', where σ replaces *local* variables without producers by $\diamondsuit$. It is easy to see that the matching substitution of $A_i\sigma_{m-1}$ and H is $\theta\sigma_{m-1}$, and $G'_m \equiv G_m\sigma_m$ where $\sigma_m = \sigma_{m-1}\sigma$. By Theorem 2, variables that do not occur in output terms of G_{m-1} do not occur in the output terms of G_m, and hence, σ_m replaces variables not occurring in the output terms of G_m. □

The following theorem relating termination of P and P' follows from the above theorem.

Theorem 4
Let P be a weakly well-moded flat GHC program satisfying assumptions A1 and A2, P' be its associated well-moded GHC program and G_0 be a weakly well-moded query satisfying A2. If P has an infinite GHC derivation starting with G_0 then P' has an infinite GHC derivation starting with G_0.

Proof
By contradiction. □

The *acyclicity* of the producer-consumer relation is crucial for the termination of the transformation procedure presented in the next section. It is however possible to handle certain kinds of cycles in the producer-consumer relation by transforming them into (weakly) well-moded programs.

Definition 16
A cycle in the producer-consumer relation of a clause is called *self-cycle* if it contains a single atom in which precisely *one* variable occurs both in input and

output positions.

It is easy to see that assumption A1 allows only local variables to occur both in input and output positions.

The following example illustrates execution of a program with self-cycles.

Example 7

Consider the following program

Moding: p(in, out); q(in, out); r(in, in, out, out) and s(in, in, out).

p(X, Y) ← true | q(X, Z_1), r(X, Z_2, Z_2, Z_3), s(Z_1, Z_3, Y)
q(X, Z) ← true | match(Z, f(X))
r(X, Y, Z, W) ← true | match(Z, a), match(W, g(Y))
s(f(X), g(Y), Z) ← true | match(Z, h(X, Y))

Consider a GHC-derivation starting with query $\leftarrow p(X, Y)$. Using the first clause, this goal will be reduced to $\leftarrow q(X, Z_1), r(X, Z_2, Z_2, Z_3), s(Z_1, Z_3, Y)$. Heads of clauses 2 and 3 match with subgoals $q(...)$ and $r(...)$. We get $\leftarrow s(Z_1, Z_3, Y)$ with substitution $\{Z_1/f(X), Z_2/a, Z_3/g(Z_2)\}$. Effectively, the goal is $\leftarrow s(f(X), g(a), Y)$ which matches with the head of clause 4, succeeding with answer substitution $\{Y/h(X, a)\}$. (Here, the output Z_3 of $r(...)$ depends on the value (binding) of its second (input) argument Z_2. And Z_2 gets instantiated during the exection of $r(...)$ itself). □

Associating weakly well-moded programs with programs having self-cycles: Given a GHC program with self-cycles in producer-consumer relations, we derive a weakly well-moded GHC program by replacing every atom $p(t_{i_1}, ..., t_{i_k}, t_{o_1}, ..., t_{o_{k'}})$ having a local variable X both in input and output positions by two atoms $p(t_{i_1}\sigma_1, ..., t_{i_k}\sigma_1, t_{o_1}\sigma_2, ..., t_{o_{k'}}\sigma_2)$, $p(t_{i_1}, ..., t_{i_k}, t_{o_1}\sigma_3, ..., t_{o_{k'}}\sigma_3)$, where σ_1 is a substitution $\{X/\diamond\}$, σ_2 is a substitution replacing all the output variables (except X) by fresh variables, and σ_3 is a substitution $\{X/X'\}$ where X' is a fresh variable.

The basic idea behind replacing the original atom by two atoms can be explained as follows. The first atom $p(t_{i_1}\sigma_1, ..., t_{i_k}\sigma_1, t_{o_1}\sigma_2, ..., t_{o_{k'}}\sigma_2)$ serves as the producer of variable X and its execution instantiates variabel X to term t if the execution of the original atom instantiates X to t and $t \not\equiv X$. If $t \equiv X$, execution of $p(t_{i_1}\sigma_1, ..., t_{i_k}\sigma_1, t_{o_1}\sigma_2, ..., t_{o_{k'}}\sigma_2)$ either does not instantiate X or instantiates X to $\diamond$. Since X has only one producer, instantiation of X (in input terms) to $\diamond$ does not lead to any inconsistency and since $\diamond$ is fresh constant (not occurring in the original program), this binding does not lead to suspension of any atom. The second atom $p(t_{i_1}, ..., t_{i_k}, t_{o_1}\sigma_3, ..., t_{o_{k'}}\sigma_3)$ serves as the producer of other variables in output positions and it takes the bindings of X produced by the first atom. The variables are renamed by σ_2 and σ_3 so that assumption A2 (that every variable has at most one producer) is satisfied. We can derive in this way a weakly well-moded program from any given GHC program with self-cycles.

Example 8

Using the above approach, we associate the following weakly well-moded program

Moding: p(in, out); q(in, out); r(in, in, out, out) and s(in, in, out).

p(X, Y) ← true | q(X, Z_1), r(X, ◇, Z_2, Z_3'), r(X, Z_2, Z_2', Z_3), s(Z_1, Z_3, Y)
q(X, Z) ← true | match(Z, f(X))
r(X, Y, Z, W) ← true | match(Z, a), match(W, g(Y))
s(f(X), g(Y), Z) ← true | match(Z, h(X, Y))

with the program (having self-cycles) given in the above Example. □

To relate termination of the associated program with that of the original program, we first establish that corresponding to every atom in a GHC derivation of $P \cup \{Q\}$, there are two atoms in a GHC derivation of $P \cup \{Q'\}$, where $Q = \leftarrow p(t_{i_1}, ..., t_{i_k}, t_{o_1}, ..., t_{o_{k'}})$, and $Q' = \leftarrow p(t_{i_1}\sigma_1, ..., t_{i_k}\sigma_1, t_{o_1}\sigma_2, ..., t_{o_{k'}}\sigma_2), p(t_{i_1}, ..., t_{i_k}, t_{o_1}\sigma_3, ..., t_{o_{k'}}\sigma_3)$.

Lemma 1

Let P be a T-program and $Q = \leftarrow p(t_{i_1}, ..., t_{i_k}, t_{o_1}, ..., t_{o_{k'}})$ be a query having variable X in both input terms as well as output terms and satisfying assumption A2. Let $Q' = \leftarrow p(t_{i_1}\sigma_1, ..., t_{i_k}\sigma_1, t_{o_1}\sigma_2, ..., t_{o_{k'}}\sigma_2), p(t_{i_1}, ..., t_{i_k}, t_{o_1}\sigma_3, ..., t_{o_{k'}}\sigma_3)$, where σ_1 is a substitution $\{X/\diamond\}$, σ_2 is a substitution replacing all the output variables (except X) by fresh variables and σ_3 is a substitution $\{X/X'\}$ where X′ is a fresh variable. Then corresponding to every GHC derivation $G_0, G_1, ..., G_n$ (call it, D) of $P \cup \{Q\}$, there is a GHC derivation $G_0', G_1', ..., G_{2n}'$ (call it, D') of $P \cup \{Q'\}$ such that the following holds for each $i \in [1, n]$:

> Corresponding to every atom $q(s_{i_1}, ..., s_{i_l}, s_{o_1}, ..., s_{o_{l'}})$ in G_i, there are two atoms $q(s_{i_1}, ..., s_{i_l}, s_{o_1}\sigma_3, ..., s_{o_{l'}}\sigma_3)\sigma$ and $q(s_{i_1}\sigma_1, ..., s_{i_l}\sigma_1, s_{o_1}\sigma_2, ..., s_{o_{l'}}\sigma_2)\ \sigma$ in G_{2i}' such that σ is the substitution $\{X/\diamond, X'/\diamond\}$ if a match atom match(X, X) is executed in the derivation $G_0, G_1, ..., G_i$, and σ is the identity substitution otherwise.

Proof

Induction on i. It may be noted that the assumptions (i) P is a T-program and (ii) Q satisfies A2, ensure that the fresh variables introduced by σ_2 and σ_3 occur only in the output terms throughout the derivation (note that in the statement of lemma, these substitutions are applied only on output terms).

Basis: $i = 0$. Trivially follows as $G_0 = Q$ and $G_0' = Q'$.

Induction hypothesis: Assume that the lemma holds for all $i < m$.

Induction step: Now, we prove the lemma for $i = m$. Let $G_{m-1} \equiv \leftarrow A_1, ..., A_j, ..., A_k$ and $A_j \equiv q(s_{i_1}, ..., s_{i_l}, s_{o_1}, ..., s_{o_{l'}})$ be the atom reduced in m-th step. There are two cases; (1) no match atom match(X, X) is executed in the derivation $G_0, G_1, ..., G_{m-1}$ or (2) a match atom match(X, X) is executed in $G_0, G_1, ..., G_{m-1}$.

Case (1): By induction hypothesis, there are two atoms $q(s_{i_1}, ..., s_{i_k}, s_{o_1}\sigma_3, ..., s_{o_{l'}}\sigma_3)$ and $q(s_{i_1}\sigma_1, ..., s_{i_k}\sigma_1, s_{o_1}\sigma_2, ..., s_{o_{l'}}\sigma_2)$ in G'_{2m-2}. Now we have two subcases: (a) $A_i \not\equiv$ match(X, X) and (b) $A_i \equiv$ match(X, X). It is very easy to see that in subcase (a), the input clause applied on $q(s_{i_1}, ..., s_{i_k}, s_{o_1}, ..., s_{o_{l'}})$ in D can also be applied on both the atoms $q(s_{i_1}, ..., s_{i_k}, s_{o_1}\sigma_3, ..., s_{o_{l'}}\sigma_3)$ and $q(s_{i_1}\sigma_1, ..., s_{i_k}\sigma_1, s_{o_1}\sigma_2, ..., s_{o_{l'}}\sigma_2)$ in D' to get G'_{2m} satisfying the lemma. Let us now consider subcase (b). By induction hypothesis, there are atoms match(X, $\diamondsuit$) and match(X', X) in G'_{2m-2} corresponding to match(X, X) in G_{m-1}. Execution of match(X, X) in D results in $G_m = \leftarrow A_1, ..., A_{j-1}, A_{j+1}, ..., A_k$ and execution of atoms match(X, $\diamondsuit$) and match(X', X) in D' results in the substitution $\sigma = \{X/\diamondsuit, X'/\diamondsuit\}$ and $G'_{2m} = \leftarrow A_1\sigma, ..., A_{j-1}\sigma, A_{j+1}\sigma, ..., A_k\sigma$. Hence, the lemma holds in this case as well.

Case (2): is similar to Case (1). □

The following theorem relates the associated weakly well-moded program with the original program having self-cycles.

Theorem 5

Let P be a flat GHC program satisfying assumptions A1 and A2 such that all the cycles in producer-consumer relation are self-cycles, P' be its associated weakly well-moded GHC prgram and G_0 be a weakly well-moded query. Then

(1) If $P \cup \{G_0\}$ has a GHC derivation of length n then $P' \cup \{G_0\}$ has a GHC derivation of length at least n.
(2) If P has an infinite GHC derivation starting with G_0 then P' has an infinite GHC derivation starting with G_0.

Proof

Part (1) follows from the above lemma through induction on the number of applications of clauses with self-cycles. Part (2) follows from Part (1). □

Remark 1

It may be noted that the above transformation on programs with self-cycles is not proposed as a programming technique. The transformation is only meant for termination analysis. The above theorem shows that termination of the resulting program implies termination of the original program. Therefore, we can apply our termination results on the resulting program for proving termination of the original program.

§4 Termination of Weakly Well-Moded GHC Programs

As shown in the previous section, we can associate a weakly well-moded GHC program P' with any given GHC program P having only self-cycles and then associate a well-moded GHC program P'' with the weakly well-moded GHC program P'. By Theorems 4 and 5, it follows that P terminates for any query if P'' terminates for it. In other words, we can reduce the termination problem of weakly well- moded GHC programs with self-cycles to that of

well-moded GHC programs. Now, we describe a transformation procedure to derive term rewriting systems from well-moded GHC programs, and establish that the termination of the derived rewrite system implies termination of the GHC program for a class of queries.

The transformation procedure is based on the idea of eliminating local variables by introducing Skolem functions. Moding information of the predicates is used for determining how many Skolem functions should be introduced and what their arities should be. For each n-ary predicate p with k output positions, we introduce k new function symbols p^1, ..., p^k of arity $n - k$; if $k = 0$, we introduce a n-ary function symbol p^0. The following example illustrates the key idea of the transformation.

Example 9
Consider the following multiplication program.

Moding: add(in, in, out) and mult(in, in, out).

add(0, Y, Z) ← true | match(Z, Y)
add(s(X), Y, Z) ← true | add(X, Y, Z1), match(Z, s(Z1))
mult(0, Y, Z) ← true | match(Z, 0)
mult(s(X), Y, Z) ← true | mult(X, Y, Z1), add(Z1, Y, Z)

From these clauses and the moding information, we obtain the following rewriting rules:

(1) The output of predicate add for inputs 0 and Y is Z, where Z is the output of match with input Y. From this, we get **$\mathbf{add^1(0, Y) \rightarrow match^1(Y)}$**.
(2) Similarly from the third clause, we get **$\mathbf{mult^1(0, Y) \rightarrow match^1(0)}$**.
(3) In the second clause, the output of add for inputs s(X) and Y is Z, where Z is the output of match with input s(Z1). Z1 in turn is the output of add for the inputs X and Y.
We get the rule **$\mathbf{add^1(s(X), Y) \rightarrow match^1(s(add^1(X, Y)))}$**.
(4) In the last clause, the output of mult for inputs s(X) and Y is Z, where Z is the output of add for the inputs Z1 and Y. Z1 is the output of mult for inputs X and Y.
We get the rule **$\mathbf{mult^1(s(X), Y) \rightarrow add^1(mult^1(X, Y), Y)}$**. □

Though input and output positions of a predicate can mix together in all possible ways, for notational convenience, we write all input positions first followed by all output positions. We write $p(t_{i_1}, ..., t_{i_j}, t_{o_1}, ..., t_{o_k})$ to denote an atom $p(...)$ containing the terms t_{i_1}, ..., t_{i_j} in input positions and t_{o_1}, ..., t_{o_k} in output positions.

For each clause c, the transformation procedure needs the following:

(1) $Prod(X) = \{\langle p^l(t_{i_1}, ..., t_{i_j}), t_{o_l}\rangle \mid X$ occurs in l-th output position of a body or guard atom $p(t_{i_1}, ..., t_{i_j}, t_{o_1}, ..., t_{o_k})$ of the clause$\}$ is the set of

producers of variable X.

(2) $Consvar = \{X \in Var(c) - in(head) \mid X \in out(head)$ or X occurs in an input position of a body-atom or guard-atom of the clause$\}$ is the set of variables consumed at least once.

(3) $Unsry = \{p^l(t_{i_1}, ..., t_{i_j}) \mid Var(t_{o_l}) \cap Consvar = \phi$ and $p(...)$ is a body or guard atom in $c\}$ is the set of output arguments of the body atoms, which do not contribute to the outputs of the head directly or indirectly. We call the computation of these arguments, unnecessary computations.

algorithm TRANSFORM (P: in; R_P: out);
begin
  $R_P := \phi$; {* R_P contains rewrite rules *}
  for each clause c: $a(t_{i_1}, ..., t_{i_k}, t_{o_1}, ..., t_{o_{k'}}) \leftarrow guard \mid body \in P$ **do**
    begin $INhead := Var(\{t_{i_1}, ..., t_{i_k}\})$
      Compute $Consvar$ and $Unsry$;
      Compute $Prod(X)$ for every variable in $Var(c) - INhead$;
      for $j := 1$ **to** k' **do**
        begin
          S := ELIMINATE-LOC-VARS($\{t_{o_j}\}$); {* S contains the right-hand sides of the rules in R_P *}
          $R_P := R_P \cup \{a^j(t_{i_1}, ..., t_{i_k}) \rightarrow t \mid t \in S\}$
        end;
    {* Following code derives rewrite rules corresponding to unnecessary computations. *}
      S := ELIMINATE-LOC-VARS($Unsry$);
      $R_P := R_P \cup \{a^{k'}(t_{i_1}, ..., t_{i_k}) \rightarrow \#(t) \mid t \in S\}$
    end;
end TRANSFORM.

function ELIMINATE-LOC-VARS(T)
{* This function goes on replacing the local variables in the set of terms T by the terms corresponding to their producers as long as there are local variables. Since producer-consumer relation of every well-moded clause is acyclic, this function is guaranteed to terminate. *}
begin $V := Var(T) - INhead$;
  while $V \neq \phi$ **do**
    begin
      for each $X \in V$ **do**
        begin $T' := \phi$;
          for each $\langle p^l(...), t\rangle \in Prod(X)$ **do**
            if $t = X$ **then** $T' := T' \cup T.\{X/p^l(...)\}$ {* Replace local var X by its producer-term. *}
              else if $t = f(X)$ **then**
                begin
                  $T' := T' \cup T.\{X/f^{-1}(p^l(...))\}$; {* Introduce inverse function *}
                  $R_P := R_P \cup \{f^{-1}(f(X)) \rightarrow X\}$

```
                end;
            T := T'
        end;
    V := Var(T) − INhead
end;
Return(T)
end ELIMINATE-LOC-VARS;
```

Handling predefined symbols:
The two predfined symbols '==' and '⇐' (we refer to this as *match*) of GHC programs can be handled easily. Since '==' has moding (in, in), an atom involving '==' cannot be a producer of any variable, and needs no processing. The predicate '⇐' has the moding (out, in), and an atom with this predicate will be a producer of variables occurring in its first argument. An atom match(Z, t) is a producer of Z and the intended meaning is that Z gets instantiated to t. The transformation treats this predicate just like any other predicate and a rewrite rule $\text{match}^1(X) \rightarrow X$ is added to the rewrite system.

Example 10
Let us illustrate the transformation with the quick-sort program (given in Example 2), from which it derives the following rewrite system. We explain how rule (2) is derived from the second clause and other rules can be derived in similar fashion. The head $q(c(H, L), S)$ contains $c(H, L)$ in input position and variable S in output position. Left-hand side of the rewrite rule will be $q^1(c(H, L))$ and to construct righ-hand-side term, algorithm TRANSFORM calls ELIMINATE-LOCAL-VARIABLE with argument $T = \{S\}$. Values of T in various iterations (marked **Ite**) of the **while** loop in ELIMINATE-LOCAL-VARIABLE are given below.

Ite. 1 $T = \{a^1(A1, c(H, B1))\}$ variable S is replaced by its producer.
Ite. 2 $T = \{a^1(q^1(A), c(H, q^1(B)))\}$
variables A1 and B1 are replaced by their producers $q^1(A)$ and $q^1(B)$.
Ite. 3 $T = \{a^1(q^1(s^1(L, H)), c(H, (q^1(s^2(L, H)))))\}$
local variables A and B are replaced by their producers.

Since there are no local variables in T afrer 3rd iteration, ELIMINATEL-LOCAL-VARIABLE returns this T to TRANSFORM which produces the rewrite rule (2) given below.

0. $\text{match}^1(X) \rightarrow X$
1. $q^1(\text{nil}) \rightarrow \text{match}^1(\text{nil})$
2. $q^1(c(H, L)) \rightarrow a^1(q^1(s^1(L, H)), c(H, q^1(s^2(L, H))))$
3. $s^1(\text{nil}, Y) \rightarrow \text{match}^1(\text{nil})$
3′. $s^2(\text{nil}, Y) \rightarrow \text{match}^1(\text{nil})$
4. $s^1(c(X, Xs), Y) \rightarrow \text{match}^1(c(X, s^1(Xs, Y)))$

4′. $s^2(c(X, Xs), Y) \rightarrow s^2(Xs, Y)$
5. $s^1(c(X, Xs), Y) \rightarrow s^1(Xs, Y)$
5′. $s^2(c(X, Xs), Y) \rightarrow match^1(c(X, s^2(Xs, Y)))$
6. $a^1(nil, X) \rightarrow match^1(X)$
7. $a^1(c(H, X), Y) \rightarrow match^1(c(H, a^1(X, Y)))$ □

4.1 Relating Termination of GHC Programs and Rewriting Systems

We first study certain properties of the term rewriting systems derived by the transformation procedure from well-moded GHC programs and establish that termination of the derived term rewriting system R_P implies termination of the GHC program P. Towards such a goal, we show that corresponding to each resolution step in the GHC-derivations, there is at least one reduction step in the rewrite derivations.

The proofs of the following three lemmas are similar to the corresponding lemmas in Ref. 10).

Lemma 2
$Var(r) \subseteq Var(l)$ for each rewrite rule $l \rightarrow r$ in R_P.

The above lemma establishes that there are no extra variables on the right-hand side of the rewrite rules in R_P, so that all the termination techniques of rewriting can be used in proving termination of GHC programs.

Notation: Let c be a well-moded clause and V be the set of variables $Var(c) - invar(head)$. We denote by $ELV(X)$, the set of terms returned by the function **ELIMINATE-LOC-VARS** for input $\{X\}$. We denote by Θ, the set of *Skolem substitutions* $\{\sigma \mid X\sigma \in ELV(X)$ for each variable $X \in V\}$. For any term t, $\Theta(t)$ denotes the set $\{\sigma \mid X\sigma \in ELV(X)$ for each variable $X \in Var(t)\}$. Assumption A2 implies that every variable in a well-moded clause has exactly one producer. Therefore, Θ is a singleton set.

Lemma 3
Let c: *head* ← *guard* | *body* be a clause in the well-moded flat GHC program P being transformed. For every atom $p(t_{i_1}, ..., t_{i_k}, t_{o_1}, ..., t_{o_{k'}}) \in$ *guard* | *body*, and for each $1 \leq j \leq k' \neq 0$ or $j = k' = 0$, the term $p^j(t_{i_1}, ..., t_{i_k})\sigma$ occurs as a subterm of right-hand side of a rewrite rule derived from clause c, where σ is the substitution in Θ.

In the following, by a context of inverse functions, we mean a context built from inverse functions introduced by the transformation procedure. If X is a variable in a term $s \equiv C[X]$ for some context $C[]$, we can construct an appropriate context $C'[]$ of inverse functions such that $C'[C[X]]$ can be reduced to X by the rewrite rules defining the inverse functions (cf. Ref. 10)).

Lemma 4
Let c: *head* ← *guard* | *body* be a clause in the well-moded flat GHC program P

being transformed and $p(t_{i_1}, ..., t_{i_k}, t_{o_1}, ..., t_{o_{k'}})$, $q(s_{i_1}, ..., s_{i_n}, s_{o_1}, ..., s_{o_{n'}})$ be two atoms in *guard* | *body* such that a variable X occurs in input terms of $p(...)$ and in l-th output term s_{o_l} of $q(...)$. Then, for each $1 \leq j \leq k' \neq 0$ or $j = k' = 0$, the term $p^j(t_{i_1}, ..., t_{i_k})\sigma$ occurs as a subterm of righ-hand side of a rewrite rule derived from c and $X\sigma \equiv C[q^l(s_{i_1}, ..., s_{i_n})]\sigma$, where C is context of inverse functions such that $C[s_{o_l}] \Rightarrow^* X$, i.e., $C[s_{o_l}]$ reduces to X by the rewrite rules defining inverse functions.

To establish the relationship between GHC-derivations and the rewriting derivations, we first define the notion of a rewrite-tree.

Definition 17
Let P be a well-moded GHC program and $Q = \leftarrow q_1(...), ..., q_n(...)$ be a weakly well-moded query. Let R_P be the rewrite system derived from P and R_Q be the rewrite system derived from the following clause (with moding (in, ..., in) for the new predicate *query*)

$$query(I_1, ..., I_n) \leftarrow q_1(...), ..., q_n(...),$$

where $\{I_1, ..., I_n\}$ is the set of variables in Q not occurring in output positions. The *rewrite-tree* RT_{PQ} of P and Q is given by:

(i) The root of the rewrite tree is $\text{Root}(RT_{PQ}) = query^0(I_1, ..., I_n)$ and
(ii) The set of children of a node $t \in RT_{PQ}$ are $\{s \mid t \Rightarrow_{R_Q \cup R_P} s\}$.

The tree RT_{PQ} contains (as branches) all the rewriting derivations of the term rewriting system $R_Q \cup R_P$ starting from an initial term $query^0(I_1, ..., I_n)$. Since rules in R_Q are applicable only on the initial term $query^0(...)$, RT_{PQ} has an infinite path (derivation) if and only if R_P is nonterminating.

The next two theorems given below establish the relationship between GHC-derivations of a GHC program and the rewriting steps in the corresponding rewrite trees. They show that there are terms (at least one) in RT_{PQ} corresponding to each atom in any GHC derivation of $P \cup \{Q\}$ and hence there are rewrite steps (at least one) corresponding to each resolution step in the GHC derivation. Before proving the results, we illustrate them through an example.

Example 11
Consider the following well-moded GHC program and the derived TRS.

moding: p(in, out); q(in, out) and r(in, out)

p(X, Y) ← true | q(X, g(f(Z), a)), r(Z, Y)
q(X, Y) ← true | match(Y, g(X, a))
r(X, Y) ← true | match(Y, h(X))

The R_P derived by the transformation procedure is the following.

$\text{match}^1(X) \rightarrow X$

$p^1(X) \rightarrow r^1(f^{-1}(g|^{-1}(q^1(X))))$
$q^1(X) \rightarrow match^1(g(X, a))$
$r^1(X) \rightarrow match^1(h(X))$
$f^{-1}(f(X)) \rightarrow X$
$g|^{-1}(g(X, Y)) \rightarrow X$

Let the query Q be $\leftarrow p(f(t), S)$. Then R_Q contains just one rule $query^0 \rightarrow p^1(f(t))$. Consider the following GHC derivation

$\leftarrow p(f(t), S)$
$\leftarrow q(f(t), g(f(Z), a)), r(Z, S)$
$\leftarrow match(g(f(Z), a), g(f(t), a), r(Z, S)$
$\leftarrow r(t, S)$

and the rewriting derivation (a branch in RT_{PQ}) $query^0 \Rightarrow p^1(f(t)) \Rightarrow r^1(f^{-1}(g|^{-1}(q^1(f(t))))) \Rightarrow r^1(f^{-1}(g|^{-1}(match^1(g(f(t), a))))) \Rightarrow r^1(f^{-1}(g|^{-1}(g(f(t), a)))) \Rightarrow r^1(f^{-1}(f(t))) \Rightarrow r^1(t)$. Now, we show the terms corresponding to each atom in the GHC derivation. $p^1(f(t))$ is the term corresponding to $p(f(t))$ in the frist goal. The subterm $q^1(f(t))$ of the third term in the rewriting derivation corresponds to the atom $q(f(t), g(f(Z), a))$ in the second goal of the GHC derivation. These two terms have the input argument of the corresponding atom as a subterm.

The term corresponding to the atom $r(Z, S)$ in the second goal is $r^1(f^{-1}(g|^{-1}(q^1(f(t)))))$. The relationship between this atom and the term can be explained as follows. The atom has variable Z in its input position and $q(f(t), g(f(Z), a))$ is the producer of Z. The output term $g(f(Z), a)$ of the atom $q(f(t), g(f(Z), a)$ can be represented as $C[Z]$, where C is the context $g(f(\square), a))$. It is easy to see that the value of Z is equal to $f^{-1}(g|^{-1}(g(f(Z), a)))$ (the latter term can be reduced to Z by the rewrite rules defining the inverse functions f^{-1} and $g|^{-1}$). In fact the term $r^1(f^{-1}(g|^{-1}(q^1(f(t)))))$ can be seen as $r^1(C^{-1}[q^1(f(t))])$, where C^{-1} is the context $f^{-1}(g|^{-1}(\square))$ of inverse functions such that $C^{-1}[C[s]]$ reduces to s by the rewrite rules defining the inverse functions. In other words, corresponding to atom $r(Z, S)$ there is a term $r^1(C^{-1}[q^1(f(t))])$ in RT_{PQ}, where C^{-1} is a context of inverse functions such that $C^{-1}[C[Z]] \Rightarrow^* Z$, wherein C is a context such that the output term of the producer of Z is $C[Z]$.

Producer of variable Z in the thid goal is $match(g(f(Z), a), g(f(t), a))$ and the term $r^1(f^{-1}(g|^{-1}(match^1(g(f(t), a)))))$ corresponding to the atom $r(Z, S)$ in the third goal can be written as $r^1(C^{-1}[match^1(g(f(t), a))])$. □

The following theorem formalizes the above observations.

Theorem 6

Let P be a well-moded T-program and Q be a weakly well-moded query satisfying A2. Then the following holds for each goal G_M in any GHC derivation $G_0 \equiv Q, G_1, ..., G_M, ...$ of $P \cup \{Q\}$: corresponding to each atom $p(s_{i_1}, ..., s_{i_k}, s_{o_1}, ..., s_{o_{k'}})$ in G_M, there are terms $\{p^i(s_{i_1}, ..., s_{i_k})\sigma \mid 1 \leq i \leq k'\}$ occurring as subterms of nodes in the rewrite-tree RT_{PQ}, where σ is a substitution such that

the following holds for each variable in G_M, $X\sigma \equiv C^{-1}[q^j(t_{i_1}, ..., t_{i_n})\sigma]$ if $t_{o_j} \equiv C[X]$ is the j-th output term of another atom $q(t_{i_1}, ..., t_{i_n}, t_{o_1}, ..., t_{o_{n'}})$ in G_M and C^{-1} is a (possibly empty) context of inverse functions such that $C^{-1}[t_{o_j}] \equiv C^{-1}[C[X]] \Rightarrow^* X$. If $k' = 0$, the term $p^0(s_{i_1}, ..., s_{i_k})\sigma$ occurs as a subterm of a node in RT_{PQ}.

Proof

Induction on M.

Basis: $M = 0$. That is, $p(s_{i_1}, ..., s_{i_k}, s_{o_1}, ..., s_{o_{l'}})$ is an atom in the initial query Q. The theorem follows in this case from Lemma 4 (applied to the rewrite rules in R_Q).

Induction hypothesis: Assume that the theorem holds for all $M < d$.

Induction step: We prove that theorem holds for $M = d$. Let $G_{d-1} = \leftarrow q_1(...), ..., q_l(...)$ and $q_i(u_{i_1}, ..., u_{i_n}, u_{o_1}, ..., u_{o_{n'}})$ be the atom reduced in d-th step. By the induction hypothesis, the theorem holds for all atoms in G_{d-1}. There are two cases; (a) $q_i(...)$ is a match atom or (b) $q_i(...)$ is not a match atom.

Case (a): $q_i(...)$ is of the form $\mathsf{match}(t_1, t_2)$. Let $X_1, ..., X_k$ be variables in t_1. The resultant of this match atom is a substitution θ with $domain(\theta) = \{X_1, ..., X_k\}$ and

$$G_d = \leftarrow q_1(...)\theta, ..., q_{i-1}(...)\theta, q_{i+1}(...)\theta, ..., q_l(...)\theta.$$

By Lemma 1, variables $X_1, ..., X_k$ can only occur in input terms of other atoms.

- The theorem holds for atom $q_j(...)\theta$, $j \neq i$ by induction hypothesis if no variable from $X_1, ..., X_k$ occurs in input terms of $q_j(...)$. In fact, $q_j(...)\theta \equiv q_j(...)$ in this case.
- Let us now consider an atom $q_j(interms, outterms)\theta$, $i \neq j$ containing a variable from $X_1, ..., X_k$ (say, $X_{l'}$) in input terms. By induction hypothesis, there are terms $q_j^a(interms)\sigma$ occurring as subterms of nodes in RT_{PQ}, where σ is a substitution such that $X_{l'}\sigma \equiv C_{l'}^{-1}[\mathsf{match}^1(t_2)\sigma]$ and $C_{l'}^{-1}[t_1] \Rightarrow^* X_{l'}$ for each $X_{l'} \in \{X_1, ..., X_k\}$. The subterm $X_{l'}\sigma \equiv C^{-1}[\mathsf{match}^1(t_2)\sigma]$ of $q_j^a(interms)\sigma$ reduces to $X_{l'}\theta\sigma$ as follows (by the application of the rewrite rule $\mathsf{match}^1(X) \rightarrow X$ and the rules applied in $C^{-1}[t_1] \Rightarrow^* X_{l'}$)

$$X_{l'}\sigma \equiv C^{-1}[\mathsf{match}^1(t_2)\sigma] \Rightarrow C^{-1}[t_2\sigma] \equiv C^{-1}[t_1\theta\sigma] \Rightarrow^* X_{l'}\theta\sigma.$$

It is easy to see that $q_j^a(interms)\sigma$ can be reduced (by reducing the subterms $X_1\sigma$, ..., $X_k\sigma$ to $X_1\theta\sigma$, ..., $X_k\theta\sigma$) to $q_j^a(interms\theta)\sigma$ and hence the theorem holds for this atom as well.

Case (b): Let $q_i(v_{i_1}, ..., v_{i_n}, Y_1, ..., Y_{n'}) \leftarrow guard \mid b_1(...), ..., b_J(...)$ be the input clause and θ be the head-matching substitution. The goal after this resolution step will be

$$G_d = \leftarrow q_1(...), ..., q_{i-1}(...), b_1(...)\theta, ..., b_J(...)\theta, q_{i+1}(...), ..., q_l(...).$$

It is convenient to view θ as $\theta_1\theta_2$, such that $domain(\theta_1) = Var(v_{i_1}, ..., v_{i_n})$ and $domain(\theta_2) = \{Y_1, ..., Y_{n'}\}$. Due to assumption A1, θ_1 is applicable only on input terms of atoms $b_1(...), ..., b_J(...)$ and θ_2 is applicable on output terms.

- Theorem holds for atoms $b_1(...)\theta, ..., b_J(...)\theta$ by Lemma 4.
- Theorem holds for atom $q_j(...)$, $j \neq i$ by induction hypothesis if $q_j(...)$ is not a consumer of any variable in $outvar(q_i(u_{i_1}, ..., u_{i_{i_1}}, u_{o_1}, ..., u_{o_{r_1}}))$. In this case, the producer of each variable occurring in input terms of $q_j(...)$ is the same in G_{d-1} and G_d.
- Let us now consider an atom $q_j(interms, outterms)$, $i \neq j$ containing a variable (say, X) in input terms such that X is a variable in j_1-th output term $u_{o_{j_1}}$ of $q_i(u_{i_1}, ..., u_{i_{i_1}}, u_{o_1}, ..., u_{o_{r_1}})$. By induction hypothesis, there are terms $q_j^a(interms)\sigma$ occurring as subterms of nodes in RT_{PQ} such that $X\sigma \equiv C^{-1}[q_i^{j_1}(u_{i_1}, ..., u_{i_{i_1}})\sigma]$ and $C^{-1}[u_{o_{j_1}}] \Rightarrow^* X$. By the definition of GHC resolution, $u_{o_{j_1}} \equiv Y_{j_1}\theta_2$, where Y_{j_1} is the variable in j_1-th output term of the head. Corresponding to the above clause, there is a rewrite rule $q_i^{j_1}(v_{i_1}, ..., v_{i_{i_1}}) \rightarrow Y_{j_1}\sigma_1$ such that $\sigma_1 \in \Theta$ and $Y_{j_1}\sigma_1 \equiv C_1^{-1}[b_{m'}^m(interms_{b_{m'}})\sigma_1]$, where $b_{m'}(...)$ is an atom in the body having Y_{j_1} in m-th output term t_{o_m} and $C_1^{-1}[t_{o_m}] \Rightarrow^* Y_{j_1}$.
  It is obvious that X occurs in the m-th output term $t_{o_m}\theta_2$ of $b_{m'}(...)\theta$ in G_d. In fact, $t_{o_m}\theta_2 \equiv C_1\theta_2[u_{o_{j_1}}] \equiv C_1\theta_2[C[X]]$, where C_1 and C are contexts such that $t_{o_m} \equiv C_1[Y_{o_{j_1}}]$ and $u_{o_{j_1}} \equiv C[X]$. It is easy to see that

$$C^{-1}[C_1^{-1}[C_1\theta_2[C[X]]]] \Rightarrow^* C^{-1}[C[X]] \Rightarrow^* X.$$

In the following, we show that the terms $q_j^a(interms)\sigma'$ occur as subterms of nodes in RT_{PQ}, where σ' is a substitution such that $X\sigma' \equiv C^{-1}[C_1^{-1}[b_{m'}^m(interms_{b_{m'}}\theta_1)\sigma']]$ and hence prove that the theorem holds for these atoms as well. We do this by showing that $q_j^a(interms)\sigma$ can be reduced to $q_j^a(interms)\sigma'$.
It is easy to see that

$$q_i^{j_1}(u_{i_1}, ..., u_{i_{i_1}}) \equiv q_i^{j_1}(v_{i_1}, ..., v_{i_{i_1}})\theta_1$$
$$\Rightarrow Y_{j_1}\sigma_1\theta_1 \equiv C_1^{-1}[b_{m'}^m(interms_{b_{m'}})\sigma_1\theta_1].$$

Since $domain(\sigma_1)$ is $Var(c) - Var(v_{i_1}, ..., v_{i_n})$, it follows that $domain(\sigma_1) \cap domain(\theta_1) = domain(\sigma_1) \cap range(\theta_1) = \phi$ and hence $b_{m'}^m(interms_{b_{m'}})\sigma_1\theta_1 \equiv b_{m'}^m(interms_{b_{m'}})\theta_1\sigma_1\theta_1$. Therefore,

$$X\sigma \equiv C^{-1}[q_i^{j_1}(u_{i_1}, ..., u_{i_{i_1}})\sigma] \Rightarrow C^{-1}[C_1^{-1}[b_{m'}^m(interms_{b_{m'}}\theta_1)\sigma_1\theta_1\sigma]]$$

and

$$C^{-1}[C_1^{-1}[b_{m'}^m(interms_{b_{m'}}\theta_1)\sigma_1\theta_1\sigma]] \equiv C^{-1}[Y_{j_1}]\sigma_1\theta_1\sigma \equiv X\theta'\sigma_1\theta_1\sigma,$$

where θ' is a substitution such that (i) $domain(\theta') = range(\theta_2)$, (ii) $range(\theta') = domain(\theta_2)$ and (iii) $V\theta' \equiv C_v^{-1}[U]$ if $U\theta_2 \equiv C_v[V]$. It is easy to see that $\sigma' = \theta'\sigma_1\theta_1\sigma$ satisfies the requirement that $X\sigma' \equiv C^{-1}[C_1^{-1}[b_{m'}^m(inter-$

$ms_{b_m}\theta_1)\sigma']]$.

This completes the proof. □

Theorem 7

Let P be a well-moded T-program and Q be a weakly well-moded query satisfying assumption A2. Corresponding to every resolution step in any GHC-derivation of $P \cup \{Q\}$, there is at least one rewrite step in the rewrite-tree RT_{PQ} of P and Q.

Proof

Let us consider a resolution step in which $p(t_{i_1}, ..., t_{i_k}, t_{o_1}, ..., t_{o_{k'}})$ is resolved using an input clause c: $H \leftarrow guard \mid B_1, ..., B_n$. By Theorem 6, corresponding to this atom there are terms $p^j(t_{i_1}, ..., t_{i_k})\sigma$, $1 \leq j \leq k'$ occurring as subterms of nodes in the rewrite tree of P and Q. Since head unification in GHC-resolution is *matching*, $p(...) = H\theta$ for some substitution θ. So $p^j(t_{i_1}, ..., t_{i_k})\sigma = l\theta\sigma$, where l is a left-hand-side term of a rewrite rule derived from clause c and hence this term can be rewritten. If $k' > 0$, there are at least k' terms to be rewritten and if $k' = 0$, there is a term $p^0(t_{i_1}, ..., t_{i_k})\sigma$ to be rewritten in RT_{PQ}. It is very easy to see that corresponding to an execution of a match atom $\mathsf{match}(t_1, t_2)$ in the GHC derivation, there are rewrite steps $\mathsf{match}^1(t_2)\sigma \Rightarrow t_2\sigma$ in RT_{PQ}. This completes the proof. □

Remark 2

The above two theorems show that there is at least one rewrite step in the rewrite tree corresponding to every possible resolution step in any GHC derivation of $P \cup \{Q\}$. No assumption is made about the selection of a candidate clause for commitment. Our proofs assume that any candidate clause can be committed. Further, no assumption is made about scheduling of subgoals and any sufficiently instantiated subgoal can be reduced. In view of this, termination of the derived term rewriting system implies termination of the given GHC program under every possible scheduling and irrespective of the relative speeds of the processes evaluating the guards of the candidate clauses.

Our main result follows from Theorem 7 and Kõnig's lemma.

Theorem 8

Let P be a well-moded T-program and R_P be the term rewriting system derived from P (by the transformation). Then, P has no infinite GHC-derivation starting with any weakly well-moded query satisfying assumption A2 if R_P terminates.

Proof (by contradiction)

Assume that there is an infinite GHC-derivation starting with some query Q. That means there are infinite resolution steps in the GHC-derivation and by Theorem 7, there are infinite rewrite steps in RT_{PQ}, i.e., the tree RT_{PQ} is infinite.

Since there are only finite number of rewrite rules, a term can be rewritten only in finite ways (remember that $Var(r) \subseteq Var(l)$ for each rule $l \rightarrow r \in R_P$) implying that RT_{PQ} is a finitely branching tree. By Kōnig's lemma, there is an infinite path (derivation) in RT_{PQ} contradicting the termination of R_P. □

Remark 3
The above theorem is stronger than the corresponding theorem given in Ref. 10) for logic programs with sequential control; in the sequential case, termination of R_P implies termination of P for well-moded queries only. The stronger result for GHC programs is due to the fact that in GHC-resolution, only matching (i.e. one-sided unification, not full unification) is needed. The above proof of Theorem 7 uses this fact.

To illustrate the above results for proving termination of GHC programs, we need to introduce some termination techniques of rewrite systems.

4.2 Termination of Term Rewriting Systems

Though termination of term rewriting systems in general is undecidable — it has been proved that one can simulate Turing machine using a single rewrite rule[1] — several techniques and tools have been proposed in the literature for proving termination of term rewriting systems. One standard technique is to look for a well-founded ordering $>$ over $\mathcal{T}$ satisfying a condition: *if* $s \Rightarrow^* t$ *then* $s > t$. If the well-founded ordering $>$ has the properties of *montonicity* ($t > u$ implies $C[t] > C[u]$) and *stability* ($t > u$ implies $t\sigma > u\sigma$), it is enough to check that $l > r$ for each rewrite rule $l \rightarrow r$ in the system. Dershowitz[4] has discussed several such well-founded orderings and techniques for proving termination of term rewriting systems. Here we explain two termination techniques, namely, recursive path ordering with status and interpretation based techniques.

Definition 18
A *quasi-ordered* set$(\mathcal{S}, \succsim)$ consists of a set $\mathcal{S}$ and a transitive-reflexive binary relation $\succsim$ defined over $\mathcal{S}$. We define the associated equivalence relation $\approx$ as $s \approx t$ if and only if $s \succsim t$ and $t \succsim s$, and associated strict partial ordering $>$ as $s > t$ if and only if $s \succsim t$ but not $t \succsim s$.

We allow an operator to have one of the following three statuses: multiset, lexicographic left-to-right (LR) and lexicographic right-to-left (RL). Multiset status is taken as default status of an operator if its status is not specified.

Definition 19 (Recursive path ordering with status;[3,7])
Let $\succsim$ be a quasi-ordering (called *precedence*) over a finite set $\mathcal{F}$ of function symbols. The *recursive path ordering* $>_{rpos}$ on the set $\mathcal{T}(\mathcal{F}, \mathcal{X})$ of terms induced by the precedence $\succsim$ is defined recursively as follows:

$s = f(s_1, ..., s_m) \succsim_{rpos} g(t_1, ..., t_n) = t$ if and only if one of the following is true

(i) $s_i \succsim_{rpos} t$ for some $i = 1, ..., m$ or
(ii) $f > g$ and $s >_{rpos} t_j$ for all $j = 1, ..., n$ or
(iii) $f \approx g$, f, g have *multiset* status and $\{s_1, ..., s_m\} \succsim_{rpos} \{t_1, ..., t_n\}$, or
(iv) $f \approx g$, $s >_{rpos} t_j$ for all $j = 1, ..., n$ and

(a) f and g have LR status and $(s_1, ..., s_m) \succsim^*_{rpos} (t_1, ..., t_n)$ or
(b) f and g have RL status and $(s_m, s_{m-1}, ..., s_1) \succsim^*_{rpos} (t_n, t_{n-1}, ..., t_1)$

where $\succsim_{rpos}$ is the multiset ordering induced by $\succsim_{rpos}$ and $\succsim^*_{rpos}$ is the lexicographic ordering induced by $\succsim_{rpos}$.

Theorem 9
For any well-founded quasi-ordering $\succsim$ and status on a given finite set of function symbols, the recursive path ordering with status $>_{rpos}$ is a monotonic and stable well-founded ordering.[3,7]

The above theorem makes the recursive path ordering with status a powerful tool in proving termination of rewrite systems. This ordering has been implemented in theorem provers like RRL, REVE etc. One of the nice properties of the recursive path ordering is that a term is bigger than all its proper subterms. This property is called *subterm property* and we use it frequently in the sequel.

Example 12
This example illustrates the use of recursive path ordering with status in proving termination of term rewriting systems. To prove termination of the following term rewriting system (derived from the multiplication program), take the precedence as $\text{mult}^1 > \text{add}^1 > \text{s} > \text{match}^1$.

$\text{add}^1(0, Y) \rightarrow \text{match}^1(Y)$
$\text{add}^1(\text{s}(X), Y) \rightarrow \text{match}^1(\text{s}(\text{add}^1(X, Y)))$
$\text{mult}^1(0, Y) \rightarrow \text{match}^1(0)$
$\text{mult}^1(\text{s}(X), Y) \rightarrow \text{add}^1(\text{mult}^1(X, Y), Y)$

Proving $l >_{rpos} r$ for rules (1) and (3) is easy. Consider the second rule. Since $\text{add}^1 > \text{match}^1$, to prove that $\text{add}^1(\text{s}(X), Y) >_{\text{rpos}} \text{match}^1(\text{s}(\text{add}^1(X, Y)))$, it is enough to prove $\text{add}^1(\text{s}(X), Y) >_{\text{rpos}} \text{s}(\text{add}^1(X, Y))$. Since $\text{add}^1 > \text{s}$, to prove that $\text{add}^1(\text{s}(X), Y) >_{\text{rpos}} \text{s}(\text{add}^1(X, Y))$, it is enough to prove $\text{add}^1(\text{s}(X), Y) >_{\text{rpos}} \text{add}^1(X, Y)$. To prove $\text{add}^1(\text{s}(X), Y) >_{\text{rpos}} \text{add}^1(X, Y)$, we need to prove that $\{\text{s}(X), Y\} \succ_{\text{rpos}} \{X, Y\}$, which is indeed the case because $\text{s}(X) >_{\text{rpos}} X$ by the subterm property.
Now consider the fourth rule. Since $\text{mult}^1 > \text{add}^1$, to prove $\text{mult}^1(\text{s}(X), Y) >_{\text{rpos}} \text{add}^1(\text{mult}^1(X, Y), Y)$, it is enough to prove that $\text{mult}^1(\text{s}(X), Y) >_{\text{rpos}} \text{mult}^1(X, Y)$ and $\text{mult}^1(\text{s}(X), Y) >_{\text{rpos}} Y$. To prove $\text{mult}^1(\text{s}(X), Y) >_{\text{rpos}} \text{mult}^1(X, Y)$, we need to prove that $\{\text{s}(X), Y\} \succ_{\text{rpos}} \{X, Y\}$, which is indeed the case. Further, $\text{mult}^1(\text{s}(X), Y) >_{\text{rpos}} Y$ by the subterm property. This completes the termination proof. □

Example 13

This example illustrates the use of status information. To prove termination of a term rewriting system with just one rule $(X + Y) + Z \rightarrow X + (Y + Z)$, we use recursive path ordering with status LR for $+$.

Since both the terms have the same function symbol at the top level, to prove that $(X + Y) + Z >_{rpos} X + (Y + Z)$, it is enough to prove (i) $(X + Y, Z) >^*_{rpos} (X, Y + Z)$, (ii) $(X + Y) + Z >_{rpos} X$ and (iii) $(X + Y) + Z >_{rpos} (Y + Z)$. By the subterm property, $X + Y >_{rpos} X$ and hence (i) follows. Again by the subterm property (ii) follows.

To prove (iii), it suffices to prove (a) $(X + Y, Z) >^*_{rpos} (Y, Z)$, (b) $(X + Y) + Z >_{rpos} Y$ and (c) $(X + Y) + Z >_{rpos} Z$. The conditions (b) and (c) follow directly from the suberm property and (a) follows from the fact that $X + Y >_{rpos} Y$. □

Interpretation based termination proofs try to map the set of terms onto a well-founded partially ordered set and prove that left-hand-sides are mapped onto bigger elements than the corresponding right-hand-sides. This is formally described below.

Definition 20

A terminating function τ from a set of terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$ to a well-founded partially ordered set $(\mathcal{W}, >)$ is composed of a set of functions $\{f_\tau: \mathcal{W}^n \rightarrow \mathcal{W}$ — f is a function symbol of arity n in $\mathcal{F}\}$ such that

$\tau(f(t_1, ..., t_n)) = f_\tau(\tau(t_1), ..., \tau(t_n))$ for every term $f(t_1, ..., t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$

and

$w > w'$ implies $f_\tau(...\tau(w)...) > f_\tau(...\tau(w')...)$ for all $w, w' \in \mathcal{W}$ and $f \in \mathcal{F}$.

Basically, terminating function consists of a set of *monotonic* mappings. Polynomial and exponential interpretations are special instances of terminating functions. A terminating function τ from terms to natural numbers, where each f_τ is a polynomial is called a polynomial interpretation. Polynomial interpretations are implemented in REVE.[13] A terminating function τ where each f_τ is a polynomial or an exponential is called an exponential interpretation. Exponential interpretations are implemented in the ORME.[14]

Example 14

To prove termination of the following system over a set of terms constructed from constants 0 and 1 and function symbols $+$ and $\times$

$$X \times (Y + Z) \rightarrow (X \times Y) + (X \times Z)$$
$$(Y + Z) \times X \rightarrow (Y \times X) + (Z \times X)$$
$$(X + Y) + Z \rightarrow X + (Y + Z)$$

we can use the following polynomial interpretation:

$\tau(0) = 2$ $\quad$ $\tau(s \times t) = \tau(s).\ \tau(t)$
$\tau(1) = 2$ $\quad$ $\tau(s + t) = 2\tau(s) + \tau(t) + 1.$ □

4.3 Termination Proofs for GHC Programs

Now, we illustrate the application of our main theorem in proving termination of GHC programs through examples.

Example 15
In Example 12, using recursive path ordering we proved termination of the rewrite system derived from the multiplication program. From Theorem 8, it follows that the multiplication program terminates for all well-moded queries. □

Example 16
Thermination of the quick-sort GHC program for all weakly well-moded queries can be established using Theorem 8 by proving termination of the derived term rewriting system (given in Example 10 above). Termination of this system is proved using the following elementary interpretation on ORME system.[14]

$$[\![a^1]\!]((x, y), (u, v)) = (x + u, 2y + v)$$
$$[\![s^1]\!]((x, y), (u, v)) = (x + u, 2y + v) \qquad [\![q^1]\!]((x, y)) = (2^x, y)$$
$$[\![s^2]\!]((x, y), (u, v)) = (x + u, 2y + v) \qquad [\![match^1]\!]((x, y)) = (x, y)$$
$$[\![c]\!]((x, y), (u, v)) = (x + u + 2, y + v) \qquad [\![nil]\!] = (0, 0)$$

Here, for $f \in \{a^1, s^1, s^2, q^1, c, nil, match^1\}$, the function $[\![f]\!]$ denotes the function f_τ given by the elementary interpretation τ. □

We make a small digression here about the style of programming in parallel logic programming languages and its effect on termination.

Example 17
In Prolog, a program for permutation is usually written as follws.

moding: perm(in, out); app(in, in, out) and split(in, out, out)

app(nil, Y, Y) ←
app(c(H, X), Y, c(H, Z')) ← app(X, Y, Z')

split(X, nil, X) ←
split(c(H, X), c(H, Y), Z) ← split(X, Y, Z)

perm(nil, nil) ←
perm(Xs, c(X, Ys)) ← split(Xs, X1s, c(X, X2s)), app(X1s, X2s, Zs), perm(Zs, Ys)

The predicate split is essentially an inverse of app. It is easy to see that this program terminates for all well-moded queries.
This Prolog program is straightforwardly translated into the following GHC program.

moding: perm(in, out); app(in, in, out) and split(in, out, out)

app(nil, Y, Z) ← true | match(Z, Y)
app(c(H, X), Y, Z) ← true | match(Z,c(H, Z')), app(X, Y, Z')

split(X, Y, Z) ← true | match(Y, nil), match(Z, X)
split(c(H, X), Y, Z) ← true | split(X, Y', Z), match(Y, c(H, Y'))

perm(nil, Y) ← true | match(Y, nil)
perm(Xs, Y) ← true | split(Xs, X1s, c(X, X2s)), app(X1s, X2s, Zs), perm(Zs, Ys),
match(Y, c(X, Ys))

The GHC program has an infinite derivation from any well-moded query involving perm as the recursive clause of perm is always applicable on the recursive literal. This demonstrates that it is not always advisable to transform Prolog programs straightforwardly into GHC programs and programming in parallel logic programming languages needs a different style. It may be noted that the transformation procedure given above derives a nonterminating term rewrite system from this GHC program reflecting its nontermination.
By replacing the recursive clause of perm in the above program by the following clause yields a terminating GHC program for permutation.

perm(c(H, Xs), Y) ← true | perm(Xs, Ys), split(Ys, Y1s, Y2s), app(Y1s, c(H, Y2s),
Y)

The transformation procedure derives the following term rewriting system from the new program.

$app^1(nil, Y) \rightarrow match^1(Y)$
$app^1(c(H, X), Y) \rightarrow match^1(c(H, app^1(X, Y)))$
$split^1(X) \rightarrow match^1(nil)$
$split^2(X) \rightarrow match^1(X)$
$split^1(c(H, X)) \rightarrow match^1(c(H, split^1(X)))$
$split^2(c(H, X)) \rightarrow split^2(X)$
$perm^1(nil) \rightarrow match^1(nil)$
$perm^1(c(H, X)) \rightarrow app^1(split^1(perm^1(X)), c(H, split^2(perm^1(X))))$

The termination of this term rewriting system can be proved using recursive path ordering with precedence $perm^1 > app^1, split^1, split^2 > c > match^1$. □

Example 18
Consider the following nonterminating program:

moding: p(out)

p(X) ← true | match(X, [a | X_1]), p(X_1)
p(X) ← true | match(X, [])

The transformation procedure derives the following rewrite system from this program.

$p^0 \rightarrow match^1([a \mid p^0])$
$p^0 \rightarrow match^1([])$

It is easy to see the nontermination of this rewrite system. □

The following examples show that the above results do not apply for programs violating assumption A1 or A2.

Example 19
Consider the following well-moded GHC program violating assumption A1. In second clause, input variable X of the head is occurring in an output term of the body.

moding: p(in, out); q(in, out); q_1(in, out, out) and s(in, out)

p(f(X), Y) ← true | s(X, U), q(U, V), p(U, Y)
q(X, Y) ← true | q_1(b, X, Y)
q_1(X, Y, Z) ← true | match(Y, f(X)), match(Z, X)
s(a, Y) ← true | match(Y, a)

The transformation derives followting TRS, whose termination can be proved using recursive path ordering with precedence $p^1 > q^1 > \#$; $q^1 > q_1^1 > f > s^1 > match^1$; $q^1 > q_1^2 > match^1$; $q^1 > b$.

$p^1(f(X)) \rightarrow p^1(s^1(X))$
$p^1(f(X)) \rightarrow \#(q^1(s^1(X)))$
$q^1(X) \rightarrow q_1^2(b)$
$q^1(X) \rightarrow \#(q_1^1(b))$
$q_1^1(X) \rightarrow match^1(f(X))$
$q_1^2(X) \rightarrow match^1(X)$
$s^1(a) \rightarrow match^1(a)$

However, the GHC program has the following infinite GHC-derivation starting from well-moded query ← p(f(b), Y).

← p(f(b), Y)
← s(b, U), q(U, V), p(U, Y)
← s(b, U), q_1(b, U, V), p(U, Y)
← s(b, U), match(U, f(b)), match(V, b), p(U, Y)
← s(b, f(b)), match(V, b), p(f(b), Y)
← s(b, f(b)), p(f(b), Y)
⋮

Example 20
Consider the following well-moded GHC program violating assumption A1. In the first clause, output variable Y of the head is occurring in an input term of the body.

moding: p(in, out); s(in, out); q(in, out) and q(in, out)

$p(f(X), Y) \leftarrow true \mid s(X, Y), p(Y, f(b))$
$s(a, Y) \leftarrow true \mid match(Y, a)$

The transformation derives following TRS, whose termination can be proved using recursive path ordering with precedence $p^1 > \#$, $f > s^1 > match^1$.

$p^1(f(X)) \rightarrow s^1(X)$
$p^1(f(X)) \rightarrow \#(p^1(s^1(X)))$
$s^1(a) \rightarrow match^1(a)$

However, the GHC program has the following infinite GHC-derivation starting from well-moded query $\leftarrow p(f(b), f(b))$.

$\leftarrow p(f(b), f(b))$
$\leftarrow s(b, f(b)), p(f(b), f(b))$
$\leftarrow s(b, f(b)), s(b, f(b)), p(f(b), f(b))$
$\leftarrow s(b, f(b)), s(b, f(b)), s(b, f(b)), p(f(b), f(b))$
$\vdots$

Example 21
Consider the following well-moded GHC program violating assumption A2.

moding: p(in, out); s(in, out); q(in, out); q_1(in, out) and q_2(in, out)

$p(f(X), Y) \leftarrow true \mid s(X, U1), q(U1, U2), p(U1, Y)$
$q(X, Y) \leftarrow true \mid q_1(X, Y), q_2(X, Y)$
$q_1(X, Y) \leftarrow true \mid match(Y, f(b))$
$q_2(X, Y) \leftarrow true \mid match(Y, X)$
$s(a, Y) \leftarrow true \mid match(Y, a)$

The transformation derives following TRS, whose termination can be proved using recursive path ordering with precedence $p^1 > q^1 > q_2^1 > match^1$; $p^1 > \#$; $q^1 > q_1^1 > f > s^1 > match^1$; $q_1^1 > b$.

$p^1(f(X)) \rightarrow p^1(s^1(X))$
$p^1(f(X)) \rightarrow \#(q^1(s^1(X)))$
$q^1(X) \rightarrow q_1^1(X)$
$q^1(X) \rightarrow q_2^1(X)$
$q_1^1(X) \rightarrow match^1(f(b))$
$q_2^1(X) \rightarrow match^1(X)$
$s^1(a) \rightarrow match^1(a)$

However, the GHC program has the following infinite GHC-derivation starting from the well-moded query $\leftarrow p(f(b), Y)$.

$\leftarrow p(f(b), Y)$
$\leftarrow s(b, U1), q(U1, U2), p(U1, Y)$
$\leftarrow s(b, U1), q_1(U1, U2), q_2(U1, U2), p(U1, Y)$
$\leftarrow s(b, U1), q_1(U1, U1), p(U1, Y)$

$\leftarrow$ s(b, f(b)), p(f(b), Y)
⋮

The following theorem is a straightforward consequence of Theorems 8, 4 and 5.

Theorem 10
Let P be a flat GHC program satisfying assumptions A1 and A2 such that all the cycles in its producer-consumer relation are self-cycles, P' be the well-moded GHC program associated with P and $R_{P'}$ be the term rewriting system derived from P' (by the transformation). If $R_{P'}$ is a terminating rewrite system, then program P has no infinite GHC-derivation starting with any query Q satisfying assumption A2 such that all the cycles in the producer-consumer relation of Q are self-cycles.

The following example shows that the converse of this theorem does not hold and termination of the derived term rewriting system is only a sufficient condition for termination of the given GHC program.

Example 22
It is easy to see that the following GHC program is a terminating program.

moding: fun(in, out); app(in, in, out) and split(in, out, out)

app(nil, Y, Z) $\leftarrow$ true | match(Z, Y)
app(c(H, X), Y, Z) $\leftarrow$ true | match(Z, c(H, Z')), app(X, Y, Z')

split(X, Y, Z) $\leftarrow$ true | match(Y, nil), match(Z, X)
split(c(H, X), Y, Z) $\leftarrow$ true | split(X, Y', Z), match(Y, c(H, Y'))

fun(nil, Y) $\leftarrow$ true | match(Y, nil)
fun(c(H, X), Y) $\leftarrow$ true | split(X, X1, X2), app(X1, X2, X'), fun(X', Y'), match(Y, c(H, Y'))

The transformation procedure derives the following term rewriting system from this program.

$app^1(nil, Y) \rightarrow match^1(Y)$
$app^1(c(H, X), Y) \rightarrow match^1(c(H, app^1(X, Y)))$
$split^1(X) \rightarrow match^1(nil)$
$split^2(X) \rightarrow match^1(X)$
$split^1(c(H, X)) \rightarrow match^1(c(H, split^1(X)))$
$split^2(c(H, X)) \rightarrow split^2(X)$
$fun^1(nil) \rightarrow match^1(nil)$
$fun^1(c(H, X)) \rightarrow match^1(c(H, fun^1(app^1(split^1(X), split^2(X)))))$

It is easy to see that this term rewriting system is nonterminating. Observe the looping derivation $fun^1([1, 2, 3, 4]) \Rightarrow^* macth^1(c(1, fun^1(app^1([2, 3, 4], [2, 3, 4]))))$. Note that both $split^1([2, 3, 4])$ and $split^2([2, 3, 4])$ can be reduced to [2, 3, 4]. Here,

the list containing elements 1, 2, 3, 4 (resp. 2, 3, 4) is abbreviated as [1, 2, 3, 4] (resp. [2, 3, 4]). □

§5 Beyond Weakly Well-Moded Flat Programs

In this section, we discuss the generalization of our results for GHC programs having cycles in the producer-consumer relation and the non-flat programs.

5.1 Breaking Cycles in the Producer-Consumer Relation

In this subsection, we propose heuristic techniques for breaking cycles in the producer-consumer relation of a clause. This involves an analysis of the atoms in the cycles and unfolding of some of these atoms using the unfold rules presented in Refs. 5) and 25). For this purpose, we construct a literal dependency graph for each clause having cycles in its producer-consumer relation. The nodes in this graph are the body and guard atoms. There is an arc (labeled with a set of variables V) from atom A to atom B if A is producer of every variable in V and B is consumer of every variable in V.

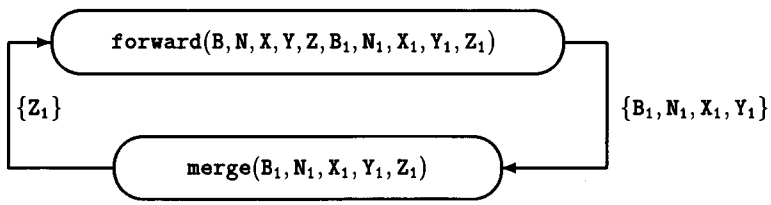
Example 23

Consider the following piece of GHC program (needed in an implementation of Merge-operators with programmable bias).

```
moding: forward(in, in, in, in, out, out, out, out, out, in);
        merge(in, in, in, in, out) and updatebias(in, in, out)

merge(B, N, X, Y, Z) ← N > 0 | forward(B, N, X, Y, Z, B1, N1, X1, Y1, Z1),
                               merge(B1, N1, X1, Y1, Z1)
forward(Bx: By, N, [M|X], Y, Z, B1, N1, X1, Y1, Z1) ← true | match(Z, [M|Z1]),
                               match(B1, Bx1: By), match(N1, N − 1), match(X1, X),
                               match(Y1, Y), updatebias(M, Bx, Bx1)
forward(Bx: By, N, X, [M|Y], Z, B1, N1, X1, Y1, Z1) ← true | match(Z, [M|Z1]),
                               match(B1, Bx: By1), match(N1, N), match(X1, X),
                               match(Y1, Y), updatebias(M, By, By1)
updatebias(started, B, B1) ← true | match(B1, B + 1)
updatebias(halted, B, B1) ← true | match(B1, B − 1)
updatebias(X, B, B1) ← X ≠ started, X ≠ halted | match(B1, B)
```

The first clause has a cyclic producer-consumer relation as depicted in the



following literal-dependency graph.

Consider a GHC-clause $H \leftarrow B_1, \ldots, B_k \mid B_{k+1}, \ldots, B_n$ with a cyclic producer-consumer relation and let $B_{I_1}\langle V_1\rangle, \ldots, B_{I_{m-1}}\langle V_{m-1}\rangle, B_{I_m}\langle V_m\rangle, B_{I_1}$ be the cycle in literal-dependency graph, i.e., B_{I_j} is producer of variables in V_j and consumer of variables V_{j-1}. If the predicate of B_{I_j} is such that the values of V_j do not depend on the values of V_{j-1}, then unfolding B_{I_j} will break the cycle in the producer-consumer relation.

Unfolding: Consider a goal g and a clause C in a logic program. In the sequential programming, g is either reducible or irreducible to the body of C, depending on whether g and head of C are unifiable. In GHC, however, there is a third case, in which g is reducible to the body of C when g is appropriately instantiated. The unfolding rules must correctly deal with such a synchronization condition expressed in the guard of C. For this purpose, the following notation is introduced and two unfolding rules are formulated by Ueda and Furukawa[25]: (i) immediate execution and (ii) case splitting.

Notation: Let g be an atom and $C :: (h \leftarrow G_U \cup G_N \mid B)$ be a clause with G_U (G_N) being unification (non-unification) goals in the guard. Then, we define:

$$
\begin{aligned}
+(g, C) &= \exists \sigma((\sigma \text{ is an mgu of } g = h \cup G_U) \\
&\qquad \wedge (g\sigma \equiv g) \wedge (G_N\sigma \text{ succeeds})) \\
-(g, C) &= \neg \exists \theta(g\theta, C) \\
?(g, C) &= \neg +(g, C) \wedge \exists \theta +(g\theta, C)
\end{aligned}
$$

$+(g, C)$ stands for "g is reducible by C now", $-(g, C)$ stands for "g is not at all reducible by C" and $?(g, C)$ stands for "g is possibly reducible with some more information (instantiation)".

The following two unfolding rules for GHC programs assume that every clause is in normal form.

1. **Immediate execution**
   Let $C \in \mathcal{P}_j$ be

   $$C :: h \leftarrow G_N \mid B_U \cup B_N$$

   and $B_N = \{g\} \cup B'_N$. If the conditions

   (I1) $\forall C_k \in \mathcal{P}_j\ (+(g, C_k) \vee -(g, C_k))$
   (I2) $\exists C_k \in \mathcal{P}_j\ (+(g, C_k))$

   both hold, then for each $C_k :: (h_k \leftarrow G_{N_k} \mid B_k) \in \mathcal{P}_j$ such that $+(g, C_k)$ holds, make a clause

   $$C'_k :: h \leftarrow G_N \mid B_U \cup B_k\theta \cup B'_N$$

   where θ is an mgu of g and h_k and let $\mathcal{P}_{j+1}$ be $(\mathcal{P}_j - \{c\}) \cup \{C'_k \mid +(g, C_k)\}$.

2. Case splitting

Let $C \in \mathcal{P}_j$ be

$$C :: h \leftarrow G_N \mid B_U \cup B_N$$

and $B_N = \{g_1, ..., g_n\}$. If the conditions

(C1) $B_U = \phi$

(C2) $\forall g \forall C' \in \mathcal{P}_j - \{C\}\ (+(g, C) \rightarrow -(g, C'))$

both hold, then make a clause C'_{ik} for each pair $g_i \in B_N$ and $C_k :: (h_k \leftarrow G_{N_k} \mid B_k) \in \mathcal{P}_j$ as follows:

- If g_i and h_k cannot be unified then let $C'_{ik} = \perp$. Otherwise, let θ be an mgu of g_i and h_k. If the condition

  (C3) $(domain(\theta \mid_{g_i}) \cup range(\theta \mid_{g_i})) - Var(h_k) \subseteq Var(h)$

  holds, let C'_{ik} be

$$C'_{ik} :: h\theta \leftarrow G_N\theta \cup G_{N_k}\theta \mid (B_N - \{g_i\})\theta \cup B_k\theta$$

  Otherwise, let $C'_{ik} = \perp$.

Finally, let $\mathcal{P}_{j+1}$ be $(\mathcal{P}_j - \{C\}) \cup \{C'_{ik} \mid C'_{ik} \neq \perp\}$.

Immediate execution rule is provided to deal with unfolding which does not involve synchronization, while case splitting is provided to correctly deal with the synchronization aspects. The following example illustrates the application of these rules in removing cycles in the producer-consumer relations.

Example 24

In the above example, the values of output variables $\{B_1, N_1, X_1, Y_1\}$ of forward(...) do not depend on the input variable Z_1. This can be seen from the clauses defining the predicate forward (This will be more evident when the rewrite rules corresponding to these clauses are written. Right-hand-sides of rewrite rules defining Skolem functions forward^2, forward^3, forward^4 and forward^5 do not contain variable Z_1). Unfolding the atom forward(...) in the above clause using the case-splitting rule gives the following two clauses without cycles.

merge(Bx: By, N, X, [M | Y], Z) ← N > 0 | match(Z, [M | Z_1]),
updatebias(M, By, By_1), merge(Bx: By_1, N, X, Y, Z_1)

merge(Bx: By, N, [M | X], Y, Z) ← N > 0 | match(Z, [M | Z_1], match(N_1, N − 1),
updatebias(M, Bx, Bx_1), merge(Bx_1: By, N_1, X, Y, Z_1)

Replacing first clause of P by these two clauses yields a new (equivalent[25]) program P' whose termination can be studied using our approach. In fact, the transformation procedure derives a term rewriting system whose termination can be proved using lexicographic path ordering[4] (with right-left status for operator merge[1]). This implies that the above program terminates for all weakly well-moded queries. □

Heuristics: *Find an atom A in the body* such that the values of variables in label of the outgoing arc (in the literal dependency graph) do not depend on the values of the variables in the label of incoming arc and *unfold* that atom using the above rules.

However, such a heuristic cannot be applied to break cycles of each and every GHC clause as illustrated by the following pathological case.

Example 25
Consider the following program.

```
moding: producer(in, out) and consumer(in, out).

producer([true | Y], X) ← true | match(X, [a | X1]), producer(Y, X1)
consumer([a | X], Y) ← true | match(Y, [true | Y1]), consumer(X, Y1)
main ← producer([true | Y], X), consumer(X, Y)
```

There is a cycle in the producer-consumer relation of the last clause and none of the two atoms in the body can be unfolded by the above heuristic. Actually, there is not much sense in trying to break the cycle as the program is intended for the execution of producer and consumer as coroutines and the program is obviously non-terminating. □

5.2 Termination of Non-Flat GHC Programs

So far, we have been studying termination of GHC programs with an implicit assumption of flatness. Now, we show that the above results can be easily generalized for proving termination of GHC programs without the assumption of flatness.

Associating a flat GHC program with a given GHC program: With a given (non-flat) GHC program P, we associate a flat GHC program P' derived from P by replacing a clause of the form

$$H \leftarrow B_1, \ldots, B_k \mid B_{k+1}, \ldots, B_n \text{ in } P$$

by a flat GHC clause

$$H \leftarrow B_{l_1}, \ldots, B_{l_{k'}} \mid B_{m_1}, \ldots, B_{m_{k''}}, B_{k+1}, \ldots, B_n \text{ in } P',$$

where $\{B_{l_1}, \ldots, B_{l_{k'}}\} \subseteq \{B_1, \ldots, B_k\}$ is the multiset of atoms with *test predicates* and $\{B_{m_1}, \ldots, B_{m_{k''}}\} = \{B_1, \ldots, B_k\} - \{B_{l_1}, \ldots, B_{l_{k'}}\}$.

Though programs P and P' may have slightly different semantics, termination behavior of P and P' is related as follows: P terminates for any query Q if P' terminates for Q. Further, our transformation derives the same term rewriting system from both P and P' as it does not distinguish between body and guard atoms. The following theorem follows from these two facts and Theorem 10.

Theorem 11

Let P be a (not necessarily flat) GHC program satisfying assumptions A1 and A2 such that all the cycles in its producer-consumer relation are self-cycles. Let P' be the well-moded GHC program associated with P' and $R_{P'}$ be the term rewriting system derived from P' (by the transformation). Then, program P has no infinite GHC-derivation starting with any query satisfying assumption A2 and (possibly) having only self-cycles, if $R_{P'}$ is a terminating rewrite system.

§6 Derivation of Moding Information

Our transformation procedure uses moding information in deriving a rewrite system from a well-moded GHC program. In this section, we explain how one can derive a suitable moding from a given GHC program. Due to the consequence (b) of the rules of suspension and commitment of GHC, the output positions of the head contain only variables (which may be bound using predefined predicate match in the body). So, if a head has a nonvariable term as an argument, that position must be input position. Using this observation and the assumption that 'input (output) variables of the head do not occur in output (input, respectively) positions of other atoms', one can derive the moding information using the following procedure.

(1) Initialize the moding of every n-ary predicate to n-tuple (?, ..., ?), i.e., the mode of every argument position is undetermined.

(2) (a) If a non-variable term occurs in i-th position of head $p(...)$, then $moding(p, i) := in$.
(b) If a variable X occurs in i-th position of the head $p(...)$ and there is an atom $match(X, t)$ in the body of the clause, then $moding(p, i) := out$.

(3) Do the following for each clause in the program:
If a variable X occurs in i-th position of head $p(...)$ and in j-th position of an atom $q(...)$ in the guard or body of the clause such that $moding(p, i) \neq moding(q, j)$, then do the following:

(3. i) if $moding(p, i) \neq ?$ and $moding(q, j) \neq ?$ then raise ERROR,
(3. ii) else if $moding(p, i) \neq ?$ then $moding(q, j) := moding(p, i)$,
(3.iii) else if $moding(q, j) \neq ?$ then $moding(p, i) := moding(q, j)$.

(4) Repeat step (3) until ERROR is found or no improvement in the moding is possible by that step.

If the modes of each argument of all the predicates are determined (i.e. moding of no predicate contains ?) by this procedure, we can transform the GHC program using this moding information and check for the termination of the program for the queries which do not have cycles in the producer-consumer relation under that moding. If the moding derived is incomplete (i.e., modes of some arguments are undetermined), we can consistently extend the moding in all

possible ways—by replacing ? by *in* or *out*—satisfying the condition that input (output) variables of the head do not occur in output (input, respectively) positions of the other atoms of the clause and check for termination of the program for the corresponding class of queries. If an ERROR is found, it is an indication that the program is not written in the style advocated.

Example 26

Consider the following example from Ref. 24) for generating *primes*.

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

The built-in predicates, $=:=$, $\neq$, $\leq$, $>$, have moding (in, in). The modings of the other predicates are derived as follows. Step (1) initialize the modings of all the predicates to (?, ..., ?). The modings after the execution of step (2) of the above procedure are primes(?, ?); gen(?, ?, out); sift(in, out); filter(?, in, out).

First iteration of step (3) does the following. Since variable Ps is occurring in output position of sift (in the body) and in second position of primes (head) of first clause, the predicate primes gets moding (?, out) in step (3.iii). Similarly, due to the presence of $N \leq Max$ in body of the second clause, predicate gen gets moding (in, in, out). Due to the presence of X mod P =:= 0 in the body of the sixth clause, predicate filter gets moding (in, in, out). The second iteration of step (3) gives the moding (in, out) for primes as the variable Max is occurring in the first position of primes and second position of gen (input position) in the first clause.

□

As stated earlier, the above procedure may not be able to determine the modes of all positions of all the predicates. Interestingly, it was able to determine the modes completely when applied to a variety of GHC programs available in the literature (a.o. Ref. 20)) including append, merge, split, permutation, quick-sort and merge-sort.

The above procedure is simple compared to the procedure presented in Ref. 27). This is due to the simplicity of our mode system (giving moding only to the arguments of predicates but not to the subterms of the arguments) over that of Ref. 27). The complex mode system in Ref. 27) was motivated (and justified) by the intended application: optimization of interprocess communication. Our mode system is a subsystem of theirs and is adequate for transforming GHC programs to term rewriting systems. The procedure presented in Ref. 27)

derives the same moding information as our procedure, when restricted to our simple mode system. See Ref. 28) for a very pleasant discussion on the role of moding information in programming and implementation of concurrent logic programming languages. The reader is referred to Refs.15) and 16) for application of mode analysis in sequentialization of concurrent logic programs.

§7 Other Parallel Logic Programming Languages

In this section, we show the applicability of our approach to other parallel (or concurrent) logic programming languages discussed in Ref. 20).

Takeuchi and Furukawa[23] have identified *safe* parallel logic programming languages that share the following property: instantiation of variables in the goal is not allowed during head unification and evaluation of guards of the clause (i.e. before commitment). While GHC, PARLOG and Oc[23] are safe, Concurrent Prolog is not. The property of safeness reduces unification in the parallel input resolution to *matching*. Further, GHC, PARLOG and Oc use *procedure level* representation of input and output[23] (i.e. some kind of (implicit or explicit) mode declarations of the predicates). Thus, it can be easily seen that our approach can also be applied to PARLOG and Oc programs satisfying the assumptions A1 and A2.

Concurrent Prolog adopts a *data level* representation of input and output[23] using read-only annotations of variables. This makes termination analysis of Concurrent Prolog different from that of GHC, PARLOG and Oc. Our method exploits the implicit moding available in GHC programs. In the following, we discuss the derivation of moding information in a Concurrent Prolog program. It can be observed that read-only variables appear in some chosen positions (which can be treated as input positions) of the atoms in the bodies of the clauses. Using such an observation, it is possible to give a procedure for deriving moding annotations for the predicates in a given Concurrent Prolog program so that we can apply our transformation.* Since, unification in the parallel resolution of Concurrent Prolog programs is not necessarily *matching*, we do not get as general a result as Theorem 11, but get a result similar to the main result of Ref. 10); that is, termination of the derived rewrite system implies termination of the corresponding *well-moded* Concurrent Prolog program for all *well-moded queries*.

§8 Discussion

Programs can be broadly classified into two types: transformational and reactive.[6] A transformational program receives some input and produces an output at the end of the computation. On the other hand, reactive programs

* In fact, on all the Concurrent Prolog programs we encountered (for example, those given in Ref. 20)), we could get consistent modings. Interestingly, all the programs in Ref. 20) satisfy our assumption that input (output) variables of the head do not occur in output (input, respectively) positions of guard or body atoms of the clause.

maintain some kind of interaction with its environment and may not stop at all. Operating systems and database management systems are examples of this. Parallel logic programming languages are useful for both these kinds of programs. While it is clear that termination is an important issue for transformational programs, it may not appear so clear for reactive programs. However, reactive programs do need to maintain interaction consistently and each interaction should be finished in finite time. Therefore, termination is important for some parts of any reactive program even if the whole pragram may need not terminate. Another important issue for reactive programs is the deadlock analysis. In this paper, we do not address the deadlock issues, but solely concern with termination issues.

In this paper, we have proposed a transformational approach for establishing the termination of GHC programs. We have proved the results, first for flat GHC programs, and then shown how the results can be generalized to non-flat GHC programs. We have enforced the discipline (which is desirable!) that input variables of the goal do not get instantiated in the computations through the following assumptions: (i) input (output) variables of the head do not occur in output (input) terms of atoms in the guard and body and (ii) no variable occurs in more than two output terms of guard and body. It is very interesting to note that this leads to the following three nice side effects: (a) It supports the style of programming advocated by Ueda,[24] (b) It enables the study termination of non-well-moded GHC programs and (c) Moding information can be derived from the given GHC program. It appears that we can allow output variables of head to occur in input terms of body/guard, provided all the output terms of body/guard are variables.

Plümer[19] studied termination of Flat GHC programs by relating GHC derivations with Prolog-like derivations and then using the techniques proposed in Refs. 17) and 18) for Prolog programs. The class of GHC programs for which his approach is applicable contains Flat GHC programs satisfying the following requirements: (i) no variable occurs in two output positions in the body (ii) no variable occurring in an input position of head can occur in output positions of body atoms (iii) output positions of body atoms are variables and (iv) no variable occurs in both input and output positions of an atom in the body. Our results need requirements (i) and (ii) but do not need (iii) and the results hold for programs having self-cycles in the producer-consumer relation (i.e., they do not satisfy the condition (iv) above) and hence our results are applicable to a larger class of programs. For example, termination of the GHC program given in Example 7 can be proved using our approach while it cannot be proved using the approach of Ref. 19). Further, we feel that our condition relating output variables to input terms, i.e., output variables of head do not occur in the input terms of body/guard is more natural than the Plümer's condition that all the output terms of body/guard are variables.

However, it must be pointed out that his approach is completely auto-

matic, whereas our approach is semi-automatic and needs some aid from the user (like precedence information) in proving termination of the derived rewrite system. As mentioned earlier, termination of the derived rewrite system is only a sufficient condition (and not a necessary condition) for termination of the given GHC program; our transformation procedure does not derive a terminating rewrite system from each and every terminating GHC program. Similarly, there is no guarantee that the approach of Ref. 19) derives useful linear predicate inequalities for proving each and every terminating GHC program. In this respect, there are some programs whose termination can be proved by our approach but not by the approach of Ref. 19) (see Example 7) and some programs whose termination can be proved by the approach of Ref. 19) but not by our approach (see Example 22).

Neither our approach nor Plümer's approach deals with programs having general cycles (with multiple atoms) in the producer-consumer relations. This amounts to saying that both the approaches have some difficulty in treating GHC-coroutining.

Acknowledgements

Thanks go to Prof. K. R. Apt for stimulating our interest in termination of parallel logic programs. We thank Prof. K. R. Apt and Prof. J. W. Klop for their constructive comments on the approach and lively discussions with the first author during his stay at CWI. The first author thanks CWI for partially supporting his visit under the CWI-TIFR exchange programme. We also thank the referees for their thought-provoking comments which have improved the presentation of the paper.

References

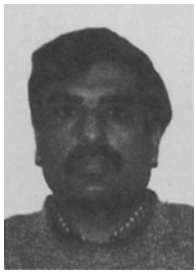
1) Dauchet, M., "Simulation of Turing Machines by a Left-Linear Rewrite Rule," *Proc. of RTA'89, Lecture Notes in Computer Science, 355*, Springer-Verlag, pp. 109-120, 1989.
2) De Schreye, D. and Decorte, S., "Termination of Logic Programs: The Never-Ending Story," *J. Logic Programming, 19/20*, pp. 199-260, 1993.
3) Dershowitz, N., "Orderings for Term-Rewriting Systems," *TCS, 17*, pp. 279-301, 1982.
4) Dershowitz, N., "Termination of Rewriting," *J. Symb. Comp., 3*, pp. 69-116, 1987.
5) Furukawa, K., Okumura, A., and Murakami, M., "Unfolding Rules for GHC Programs," *New Generation Computing, 6*, pp. 143-157, 1988.
6) Harel, D. and Pnueli, A., "On the Development of Reactive Systems," in *Logics and Models of Concurrent Systems* (K. R. Apt, ed.), Springer-Verlag, 1985.
7) Kamin, S. and Levi, J.-J., "Two Generalizations of Recursive Path Ordering," *unpublished note*, Department of Computer Science, University of Illinois, Urbana, 1980.
8) Kapur, D. and Zhang, H., "An Overview of Rewrite Rule Laboratory (RRL)," *Proc. of RTA'89, Lecture Notes in Computer Science, 355*, Springer-Verlag, pp. 559-563, 1989.
9) Kleinman, A., Moscowitz, Y., Pnueli, A., and Shapiro, E., "Communication with Directed Logic Variables," *Proc. 18th ACM Symposium on Principles of Program-*

ming Languages, POPL'91, pp. 221-232, 1991.

10) Krishna Rao, M. R. K., Kapur, D., and Shyamasundar, R. K., "A Transformational Methodology for Proving Termination of Logic Programs," *Proc. of Computer Science Logic, CSL'91, Lecture Notes in Computer Scienc, 626*, Springer-Verlag, pp. 213-226, 1991. Revised version to appear in J. Logic Programming, 1997.
11) Krishna Rao, M. R. K., Pandya, P. K., and Shyamasundar, R. K., "Verification Tools in the Development of Provably Correct Compilers," *Proc. of 5th Symposium on Formal Methods Europe, FME'93, Lecture Notes in Computer Science, 670*, Springer-Verlag, pp. 442-461, 1993.
12) Krishna Rao, M. R. K., Kapur, D., and Shyamasundar, R. K., "Proving Termination of GHC Programs," *Proc. of 10th International Conference on Logic Programming, ICLP'93*, pp. 720-736, 1993.
13) Lescanne, P., "Computer Experiments with the REVE Term Rewriting Systems Generator," *Proc. of 10th ACM POPL'83*, pp. 99-108, 1983.
14) Lescanne, P., "Termination of Rewriting Systems by Elementary Interpretations," *Proc. of Algebraic and Logic Programming, ALP'92, Lecture Notes in Computer Science, 632*, Springer-Verlag, pp. 21-36, 1992.
15) Massey, B. and Tick, E., "Sequentialization of Parallel Logic Programs with Mode Analysis," *Proc. of LPAR'93, Lecture Notes in Computer Science, 698*, Springer-Verlag, pp. 205-216, 1993.
16) Tick, E. and Koshimura., "Static Analysis of Concurrent Logic Programs," *J. of Programming Language Design and Implementation*, 1995.
17) Plümer, L., "Termination Proofs for Logic Programs," *Ph.D. thesis*, University of Dortmund. Also appears as *Lecture Notes in Computer Science, 446*, Springer-Verlag, 1990.
18) Plümer, L., "Automatic Termination Proofs for Prolog Programs Operating on Nonground Terms," *Proc. of International Logic Programming Symposiun, ILPS'91*, pp. 503-517, 1991.
19) Plümer, L., "Automatic Verification of Parallel Logic Programs: Termination" in *Logic Programming: Formal Methods and Practical Applications* (C. Beirle and L. Plümer, ed.), North-Holland, 1994. Preliminary version as "Automatic Verification of GHC-Programs: Termination," *Proc. of FGCS'92*.
20) Shapiro, E., *Concurrent Prolog*, collected papers Volume 1 and 2, MIT Press, 1987.
21) Shapiro E., "The Family of Concurrent Logic Programming Languages," *ACM Computing Surveys, 21* pp. 413-510, 1989.
22) Shyamasundar, R. K., Krishna Rao, M. R. K., and Kapur, D., "Rewriting Concepts in the Study of Termination of Logic Programs," *Proc. of ALPUK'92* Conference (K. Broda, ed.), Workshops in Computing Series, Springer-Verlag, pp. 3-20, 1990.
23) Takeuchi, A. and Furukawa, K., "Parallel Logic Programming Languages," in *Concurrent Prolog* (E. Shapiro, ed.), Chapter 6, pp. 188-201, 1987.
24) Ueda, K., "Guarded Horn Clauses," *Proc. of Logic Programming'85, Lecture Notes in Computer Science, 221*, Springer-Verlag, pp. 168-179, 1985. Also appears as chapter 4 in Ref. 20).
25) Ueda, K. and Furnkawa, K., "Transformation Rules for GHC programs," *Proc. of FGCS'88*, pp. 582-591, 1988.
26) Ueda, K., "Designing a Concurrent Programming Language," *Proc. of InfoJapan*'90 (organized by the Information Processing Society of Japan), pp. 87-94, 1990.
27) Ueda K. and Morita, M., "A New Implementation Technique for Flat GHC," *Proc. of International Conference Logic Programming, ICLP'90*, pp. 3-17, 1990. Revised version appears as "Moded Flat GHC and Its Message-Oriented Implementation Technique," *New Generation Computing, 13*, pp. 3-43.

28) Ueda, K., "I/O Mode Analysis in Concurrent Logic Programming," *Proc. of TPPP'94*, *Lecture Notes in Computer Science*, *907*, Springer-Verlag. pp. 356-368, 1994.

M. R. K. Krishna Rao, Ph.D.: He currently works as a senior research fellow at Griffith University, Brisbane, Australia. His current interests are in the areas of logic programming, modular aspects and non-copying implementations of term rewriting, learning logic programs from examples and conuterexamples and dynamics of mental states in rational agent architectures. He received his Ph.D in computer science from Tata Institute of Fundamental Research (TIFR), Bombay in 1993 and worked at TIFR and Max Planck Institut für Informatik, Saarbrücken until January 1997.



Deepak Kapur, Ph.D.: He currently works as a professor at the State University of New York at Albany. His research interests are in the areas of automated reasoning, term rewriting, constraint solving, algebraic and geometric reasoning and its applications in computer vision, symbolic computation, formal methods, specification and verification. He obtained his Ph.D. in Computer Science from MIT in 1980. He worked at General Electric Corporate Research and Development until 1987. Prof. Kapur is the editor-in-chief of the Journal of Automated Reasoning. He also serves on the editorial boards of Journal of Logic Programming, Journal on Constraints, and Journal of Applicable Algebra in Engineering, Communication and Computer Science.

R. K. Shyamasundar, Ph.D.: He currently works as a professor at Tata Institute of Fundamental Research (TIFR), Bombay. His current intersts are in the areas of logic programming, reactive and real time programming, constraint solving, formal methods, specification and verification. He received his Ph.D in computer science from Indian Institute of Science, Bangalore in 1975 and has been a faculty member at Tata Institute of Fundamental Research since then. He has been a visiting/regular faculty member at Technological University of Eindhoven, University of Utrecht, IBM TJ Watson Research Centre, Pennsylvania State University, University of Illinois at Urbana-Champaign, INRIA and ENSMP, France. He has served on (and chaired) Program Committees of many International Conferences and has been on the Editorial Committees.