ICOT Technical Memorandum: TM-0160

TM-0160

On the Operational Semantics of Guarded Horn Clauses

by
K. Ueda

April, 1986

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191－5
Telex ICOT J32964

Institute for New Generation Computer Technology

ON THE OPERATIONAL SEMANTICS OF GUARDED HORN CLAUSES

Kazunori Ueda (上 田 和 紀)

Institute for New Generation Computer Technology, Tokyo, Japan

Abstract: We consider the operational semantics of Guarded Horn Clauses (GHC). The purpose is to clarify what should be regarded as primitive operations of GHC. Such consideration is important because the existing algorithms for unification and resolution have not been considered very well in terms of parallel execution. We begin by showing the operational semantics of unification. It is anarchical in that termination is not guaranteed for a couple of reasons. Although we believe the anarchical semantics is meaningful as it is, we also discuss how to reduce the possibility of nontermination. Lastly, we give the operational semantics of GHC by extending the semantics of unification.

1. INTRODUCTION

The language Guarded Horn Clauses [Ueda 85] (hereafter abbreviated to GHC) is intended to be the standard of logic programming languages which allows parallel execution. GHC has introduced into Horn clauses the concept of guard in order to express causality (or the direction of computation) and choice (don't-care) nondeterminism. This additional mechanism provided GHC with an ability to express important concepts in parallel programming such as processes, communication, and synchronization, and thus augmented the expressive power of the original Horn-clause logic to the level of a practical programming language. From a practical point of view, GHC may look similar to Concurrent Prolog [Shapiro 83] and PARLOG [Clark and Gregory 84]. It has, however, the unique feature of simplicity: GHC is unique in that guard is the only syntactic construct added to Horn clauses. GHC has no multiple environments or backtracking (= multiple environments expanded in the time axis), so its semantics and implementation should be simple compared with Concurrent Prolog and even with sequential Prolog.

One may contend that GHC is not a logic programming language because it has lost the completeness of the Horn-clause logic. However, GHC was undoubtedly born from the investigation of logic programming. Moreover, we have to use choice nondeterminism to write a program which interfaces with a real world, and in such a case the completeness is rather a obstacle. The only way to stay with completeness would be to design another programming language based on another logic which is capable of handling don't-care nondeterminism in its own framework.

Another view of GHC is to regard it as a generalization of nondeterministic dataflow languages. Depending on available binding information, a goal reduces itself to a (possibly empty) set of other goals, possibly generating additional binding information. GHC is a generalization of nondeterministic dataflow languages in the sense that the data structures it handles are not limited to streams and that incomplete data structures can be handled. The capability of handling incomplete data structures enables us to express demand-driven computation naturally without introducing new primitives.

In the following chapters, we first discuss three directions to the semantics of GHC. Then we introduce a nondeterministic unification algorithm which models parallelism of GHC better than previous ones. Lastly, we describe the semantics of GHC by extending the unification algorithm we propose. This paper assumes familiarity with GHC; for more informal description and program examples, the readers are asked to refer to [Ueda 85].

2. THREE DIRECTIONS TOWARDS THE SEMANTICS OF GHC

2.1. Declarative Semantics as a Logic Programming Language

The declarative semantics of a logic program which does not deal with infinite computation is well studied in [van Emden and Kowalski 76] and [Apt and van Emden 82]. [Hagiya 83] and [Lloyd 84] try to capture the declarative semantics of infinite computation by the greatest fixpoint on the extended Herbrand base which includes infinite atoms. This idea was first mentioned in [van Emden and de Lucena 82].

Unfortunately, due to the don't-care nondeterminism resulting from the commitment operator and the restriction of dataflow from guards, the semantics along this line cannot capture all the aspects of GHC. Moreover, it fails to show the causality among bindings which is often the central matter of interest in parallel logic programming. However, it may still be useful for understanding the semantics of a subclass of GHC programs which do not require don't-care nondeterminism.

2.2. Process-Oriented Semantics

As we stated before, GHC can be regarded as a generalization of nondeterministic dataflow languages. A GHC goal generates new bindings between variables and terms depending on, and possibly after waiting for, the bindings generated by other goals. Hence, GHC goals can be regarded as processes interacting with one another by means of instantiation of variables. The semantics of such processes could be given by modifying the semantics of nondeterministic dataflow languages given in [Brock and Ackermann 81] and [Staples and Nguyen 85], for example.

This direction is promising because the obtained semantics will capture in an abstract manner all the aspects of GHC including causality and don't-care nondeterminism. It should provide a theoretical foundation for every kind of mechanical and manual handling of a program including program transformation, verification, compilation and optimization.

2.3. Operational Semantics

A general purpose of operational semantics is to show the guideline for implementation algorithmically. In the case of parallel languages this is especially important, since it shows what should be considered as indivisible or primitive operations. However, operational semantics bears a general difficulty in its abstractness. If it is too specific, it can serve only for a small range of implementations and one cannot distinguish between essential and inessential matters. Features whose implementation is not essential for language definition should be described only functionally: Examples are arithmetics and access to array elements.

On the other hand, if the operational semantics goes too abstract or

functional, it may fail to serve as a guideline of any possible implementation. However, it seems that distinction between essential and inessential matters can have only subjective criteria. Being moderately abstract is especially difficult for new languages and parallel languages, since it is hard to assume in advance all good implementations that may appear in the future.

The operational (or procedural) semantics of a logic programming language is usually identified with some proof procedure of a given formula; in the case of a Horn-clause language it is identified with a refutation procedure. The semantics of GHC also can be based on resolution and it should be the cleanest way to capture the aspect of GHC as a logic programming language, but it must also express the semantics of the additional construct, guard. Moreover, the semantics must clarify what can be executed in parallel in order to serve for fully parallel implementation. Such consideration is important because the existing algorithms for unification and resolution have not been considered very well in terms of parallel execution.

We consider fully parallel execution as the standard and serialization of primitive operations as optimization for the current hardware technology which favors sequential computation. This view is the exact opposite of the usual view of optimization. The reason why we do so is that our purpose is to find the smallest possible granularity and to reveal every possible parallelism. However, fully parallel execution may fall into anarchy and undesirable situations such as deadlock and starvation may result. The anarchy could be avoided by controlling parallelism. It could be achieved by

(1) employing larger units of data and larger unit of operations to handle them, and/or

(2) introducing some apparatus for control.

Sequential execution of some primitive operations falls into Item (2).

Moreover, we allow apparently useless computation as long as it does not change the intended semantics. In parallel computation, it may often be the case that we can gain efficiency by doing some computation in

advance whose result may possibly turn to be unnecessary afterwards. To disallow any useless computation would be very difficult and it would cause serious inefficiency in distributed computation. Hence it seems better to show what can be allowed rather than to show exactly what is needed. This will again be the opposite of the usual manner which considers optimization by means of backup computation only as a consequence of the semantics.

3. THE NONDETERMINISTIC UNIFICATION (SEMI-)ALGORITHM

The most important and delicate operation in GHC is unification. This chapter shows the nondeterministic unification algorithm which will be incorporated into the semantics of GHC. Precisely speaking, it is a semi-algorithm since termination is not guaranteed due to deadlock and other causes. However, we discuss possible ways to guarantee termination for unifiable cases in Section 3.5.

The algorithm gives the base of the semantics of the unification of GHC. The point is that we no longer handle a complex term as an atomic entity nor we consider a variable as atomic. Thus the algorithm is more nondeterministic than the nondeterministic algorithm in [Martelli and Montanari 82]. More importantly, while the algorithm of Martelli and Montanari is sequential, ours allows parallelism.

3.1. The Algorithm

Our formalization basically follows [Martelli and Montanari 82]. Function symbols, variables, terms, and substitution are defined as usual. In examples, we begin function symbols with lowercase letters and variables with uppercase letters. We underline those object-level symbols to distinguish them from metasymbols appearing elsewhere.

The unification problem is a set of equations of the following form

$$S_1=T_1, \ \ldots, \ S_n=T_n.$$

where S_i and T_i are terms. A solution of the unification problem, called a unifier, is any substitution that makes S_i and T_i identical for all i's simultaneously.

Given a problem, the algorithm repeatedly performs any of the following transformations. These transformations can be done in parallel, as long as they do not interfere, i.e., they do not rewrite any part of currently 'chosen' entities. Unless stated otherwise, the chosen entities become unchosen when the specified transformation is complete. We may attach 'marks' to variables to prevent backward rewriting. The algorithm terminates when no transformation applies.

(a) Choose any equation of the form S=T where S and T are not variables. If the two principal function symbols are different, unchoose this equation and stop with failure. Otherwise, the equation is of the form $f(S_1, \ldots, S_n)=f(T_1, \ldots, T_n)$ where f is some n(>=0)-ary function symbol and S_i's and T_i's are terms which are possibly marked variables, and rewrite it to $S_1=T_1, \ldots, S_n=T_n$ in any way but without erasing S_i's and T_i's. When some of S_i's and T_i's are marked, they are unmarked. The condition "without erasing S_i's and T_i's" means that S_i's and T_i's must not disappear from the problem during rewriting.

(b) Choose any equation of the form $X=f(T_1, \ldots, T_n)$ or $f(T_1, \ldots, T_n)=X$ where f is some n(>0)-ary function symbol, T_i's are terms that are not marked variables and X is a variable, and rewrite it to $X=f(X_1^*, \ldots, X_n^*), X_1=T_1, \ldots, X_n=T_n$ in any way but without erasing X and T_i's, where X_i's are distinct variables which are different from the variables in the current problem. The original equation $X=f(T_1, \ldots, T_n)$ or $f(T_1, \ldots, T_n)=X$ becomes unchosen when it is rewritten to $X=f(X_1^*, \ldots, X_n^*)$. Asterisks denote marks, and they are attached to the new variables to prevent backward rewriting toward the original term.

(c) Choose any equation of the form X=X where X is a variable, and erase it.

(d) Choose any equation of the form X=Y where X and Y are distinct variables, and one of the other occurrences of non-marked X. Then replace that occurrence by Y.

(e) Choose any equation of the form $X=f(X_1^*, \ldots, X_n^*)$ where f is an n(>=0)-ary function symbol and X_i^*'s are marked variables, and one of

the other occurrences of unmarked X. Then replace that occurrence by $f(X_1, \ldots, X_n)$.

3.2. Examples

The following examples illustrate some subtle points in the algorithm.

(1) $\underline{X}=\underline{a}$, $\underline{X}=\underline{b}$. The first equation can rewrite the second $\underline{X}$ to '$\underline{a}$', and the second equation can rewrite the first $\underline{X}$ to '$\underline{b}$'. If the second $\underline{X}$ is rewritten first, then the equation $\underline{X}=\underline{b}$ is changed to the equation $\underline{a}=\underline{b}$, which causes failure. If the first $\underline{X}$ is rewritten first, the equation $\underline{X}=\underline{a}$ is changed to the equation $\underline{b}=\underline{a}$, which also causes failure. Therefore, the order of rewriting is independent of the result. When the two equations try to rewrite the other simultaneously, however, deadlock may occur. The problem of deadlock will be discussed in Section 3.5.

(2) $\underline{X}=\underline{a}$, $\underline{X}=\underline{a}$, $\underline{X}=\underline{b}$. This example shows why an equation being chosen must not be rewritten by other equations. Suppose that the first equation and the second equation are simultaneously chosen and each of them rewrites the other to $\underline{a}=\underline{a}$. Then the original problem is rewritten to '$\underline{a}=\underline{a}$, $\underline{a}=\underline{a}$, $\underline{X}=\underline{b}$', and then to '$\underline{X}=\underline{b}$'. This is obviously an erroneous rewriting. If chosen equations are locked, this situation never occurs.

3.3. Properties of the Algorithm

Since the above algorithm does not guarantee termination, what we can show at best is

(1) that when the algorithm terminates, the original problem has a unifier which is evident from the obtained form, and

(2) that when the algorithm stops with failure, the original problem has no unifiers.

We give some theorems (proofs omitted) which together show that our algorithm 'computes' the most general unifier of the original problem when it terminates.

Theorem 1. When the algorithm terminates, the obtained set of equations has the form

$$X_1=T_1, \ldots, X_k=T_k, X_{k+1}=Y_1, \ldots, X_{k+m}=Y_m \qquad (k>=0,\ m>=0)$$

where

(a) $X_1, \ldots, X_{k+m}$ are distinct variables,

(b) The sets $\{X_i\}$ and $\{Y_i\}$ are disjoint, and

(c) T_i's are of the form $f(X_{i1}*, \ldots, X_{il}*)$ where f is an $n(>=0)$-ary function symbol and all X_{ij}'s are distinct variables and $\{X_i\}$ includes $\{X_{ij}\}$.

(end of Theorem 1).

We say that a variable Y is immediately under a variable X in the obtained set iff for some i (i=1, ..., k), $X=X_i$ and Y appears in T_i, and that a variable Y is under a variable X iff Y is immediately under X or there exists a variable Z such that Y is under Z and Z is immediately under X. Then we can prove the following theorem.

Theorem 2. The above set of equations has no cycles, that is, none of X_1, ..., X_k is under itself (end of Theorem 2).

We define an equivalence relation between two substitutions. We say that substitutions 's' and 't' are equivalent with respect to a set of variables U iff s(V)=t(V) for all V's in U. The equivalence relation is denoted by '$=_U$'.

Theorem 3. Every transformation shown in Section 3.1, if successful, preserves the quotient set of all unifiers by the relation '$=_U$', where U is a set of variables appearing in the original problem. If the transformation stops with failure in Step (a), then the set of equations is ununifiable (end of Theorem 3).

We take an equivalence class because Rule (b) introduces new variables. Those variables, which could take arbitrary values before the transformation, are unified with some terms, and so the set of all unifiers is reduced. However, taking the quotient set 'forgets' the differences among unifiers in those new variables.

Theorem 4. Assume the algorithm terminates and the following set of equations is obtained:

$$X_1=T_1,\ \ldots,\ X_k=T_k,\ X_{k+1}=Y_1,\ \ldots,\ X_{k+m}=Y_m.$$

Let a substitution 'g' be defined as

$$\begin{aligned} g(X) &= T_i \quad \text{if } X=X_i \text{ for some } i=1,\ \ldots,\ k \\ g(X) &= Y_i \quad \text{if } X=X_{k+i} \text{ for some } i=1,\ \ldots,\ m \\ g(X) &= X \quad \text{otherwise} \end{aligned}$$

and by g^n we mean the application of g repeated n times. Then, $g^n=g^{n+1}=g^{n+2}=\ldots$ for some n which does not exceed k+1, and g^n is the most general unifier of the obtained equation (end of Theorem 4).

Theorems 1 to 3 together state that when our algorithm terminates, the original set of equations is unifiable with the most general unifier almost evident from the final set of equations, and that when it stops with failure, the original set of equations is ununifiable. In fact, by adding some conditions we can guarantee termination in the case where the original problem has a unifier, as will be discussed in Section 3.5.

3.4. Motivations and Implications

We make some remarks on the above algorithm to clarify the underlying motivations and implications.

Our algorithm differs from that of [Martelli and Montanari 82] in the following three points. One is that a non-variable term with arguments which are not guaranteed to be new variables is not treated as atomic. For example, the equation

$$X=cons(a,nil) \qquad \ldots(i)$$

is not directly used for substitution of X appearing in the problem. It is first rewritten to

$$X=cons(A^*,B^*),\ A=a,\ B=nil \qquad \ldots(ii)$$

where A^* and B^* are new variables, and then the equation $X=cons(A^*, B^*)$ is

used for instantiation. In general, only an equation of the form X=T where X is a variable and T is the most general term with some principal function symbol whose arguments, if any, are all distinct marked variables can be used for instantiation. This means that the primitive operation for the instantiation of some occurrence of a variable is to determine its principal function symbol. This decision was motivated by the observation that Equations (i) and (ii) are logically the same while Equation (ii) has smaller granularity. We regard Equation (i) as an abbreviation of Equation (ii).

The practical meaning of this is as follows. When we transmit a large data structure from one processor to another, we often transmit it block by block. Our algorithm explicitly states that a large data structure is not an atomic entity but can be transmitted little by little, possibly on demand, with untransmitted parts indicated by uninstantiated variables. Moreover, the consumer can use the transmitted value before the transmission is complete.

The second point is that we do not consider a variable as a centralized entity but as a distributed entity. This decision was motivated by the observation that the problem

$$\underline{X}=\underline{a},\ \underline{X}=\underline{b}$$

can be regarded as an abbreviation of

$$\underline{X1}=\underline{a},\ \underline{X1}=\underline{X2},\ \underline{X2}=\underline{b}.$$

The practical meaning of this is that a variable need not be implemented by a single memory cell. It is quite likely that each processor has a local cache for variables. The algorithm explicitly allows it, and it also says that local copies of a variable need not have the same value at the same time (assuming that we have some notion of global time), as long as they become identical finally. To put it differently, communication by shared variables may have a potential delay.

The third point in which our algorithm is different from [Martelli and Montanari 82] is that we have omitted the so-called 'occur check'; see Section 3.5.3 for more detail.

3.5. Termination

The algorithm shown in Section 3.1 may fail to terminate for the following reasons:

(1) lack of a mechanism for controlling mutual exclusion,

(2) lack of the fairness assumption, and

(3) lack of the occur check.

The first two points are particularly important because they may cause nontermination in unifiable cases. In the following, we discuss each of the above problems.

3.5.1. Controlling Mutual Exclusion

The algorithm states that the entities currently chosen by some transformation should not be rewritten by other transformations. This is the rule of mutual exclusion. However, as a result of regarding a variable as a non-atomic, distributed entity, we must lock two resources, an equation and an occurrence of a variable, when we apply Rules (d) and (e). As seen in Section 3.2, when we rewrite some occurrence of a variable X, there should be a moment when both that occurrence and the equation being chosen are locked or protected against rewriting by other equations.

The problem is that locking two resources may generally cause deadlock if we wait until one of them is available, lock it, and wait until the other resource is available. One way to avoid deadlock in such an incremental locking scheme is to introduce an ordering to the occurrences of equations and variables and to lock the entity lower in that ordering first. The only significant fact about the ordering is that two entities to be locked are ordered; it does not matter how they are ordered and a newly created entity may have an arbitrary order.

Ordering avoids deadlock. That is, if we can guarantee that the algorithm terminates when each transformation is performed nondeterministically but sequentially, we can guarantee termination also when we allow parallelism.

3.5.2. Fairness

However, ordering of entities alone does not guarantee termination even in unifiable cases. Consider the following example:

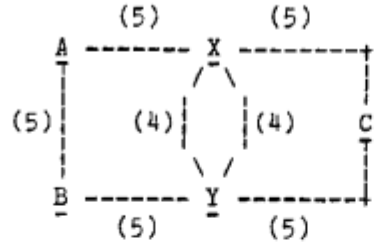
X=Y, Y=X, X=a.

The first equation X=Y may rewrite the third equation to Y=a, and then the second equation may rewrite the third equation back to X=a. We must disallow such a sequence of rewriting to be exclusively performed infinitely many times by introducing some notion of fairness.

One possible definition of fairness is as follows. We first consider a non-directed graph G formed by a set S of equations whose left- and right-hand sides are both variables. G has a node V corresponding to each variable V in S, and an arc (V1, V2) corresponding to each occurrence of an equation V1=V2. The graph may have loops, and we assign to each arc the size of the largest loop it forms. For example, the set of equations

X=Y, X=Y, X=A, A=B, B=Y, X=C, C=Y

forms the following graph:



The number assigned to each arc shows the size of the largest loop it forms.

Now we can define fairness. We first consider nondeterministic but serialized execution to keep the above notion of loops meaningful. Let L be the sum of the values assigned to the arcs of the graph G formed by S. A fair execution must not repeatedly choose the transformation rule (d) without reducing L. Speaking more precisely, when Rule (d) is repeatedly chosen, L must be reduced to L' after finitely many transformations and it must never exceed L' thereafter, or else other rules must be chosen after

finitely many transformations. Nontermination illustrated in the beginning of this subsection is now disallowed, since L is initially 4 and each transformation never reduces it.

Fairness of parallel execution is easily defined in terms of fair serialized execution: Every fair parallel execution is obtained from some fair serialized execution by allowing overlapping of transformations under the rule of mutual exclusion.

3.5.3. Occur Check

For a set of equations for which usual unification algorithms detect a cycle and stops with failure, our algorithm indefinitely computes the values of infinite terms. For example, an equation

$$X=f(X)$$

is rewritten as follows:

$X=f(X1^*)$, $X1=X$ by Rule (b)
$X=f(X1^*)$, $X1=f(X1)$ by Rule (e)
$X=f(X1^*)$, $X1=f(X2^*)$, $X2=X1$ by Rule (b)
...

However, since the execution does not terminate, we cannot get any result in the above framework. One way to observe the value of X is to introduce the notion of 'observation variables'. We specify observation variables $V_1, \ldots, V_k$ in the unification problem as follows:

$$S_1=T_1, \ldots, S_n=T_n; V_1, \ldots, V_k.$$

These variables are rewritten according to Rules (d) and (e). Then if we give the above equation as

$$X=f(X); X$$

the observation variable X will be indefinitely instantiated to $f(X1)$, $f(f(X2))$, and so on. The notion of observation variables well models stream-oriented output of GHC. Observation variables can be used also for observing the result of terminating unification; we can prove that when the

algorithm terminates, an observation variable X is instantiated to mgu(X) where mgu is the most general unifier.

When we take infinite computation into account, however, we need a rather different notion of fairness from that introduced in Section 3.5.2. For example, it seems undesirable to continue to compute the value of X without computing the value of Y in the following problem:

X=f(X), Y=g(Y); X, Y.

However, this problem is out of the scope of this paper and we do not discuss it any further.

4. THE OPERATIONAL SEMANTICS OF GHC

In this chapter, we define the operational semantics of GHC by extending the operational semantics of unification defined in Chapter 3.

4.1. Syntax of GHC

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

$H :- G_1, \ldots, G_m \mid B_1, \ldots, B_n.$ $(m >= 0, n >= 0).$

where H, G_i's, and B_i's are atomic formulas. H is called a clause head, G_i's are called guard goals, and B_i's are called body goals. The operator '|' is called a commitment operator. The part of a clause before '|' is called a guard, and the part after '|' is called a body. Note that a clause head is included in a guard. A goal is a call either to the predefined unification predicate '=' or to some other predicate which should be user-defined.

A goal clause has the following form:

$:- B_1, \ldots, B_n.$ $(n >= 0).$

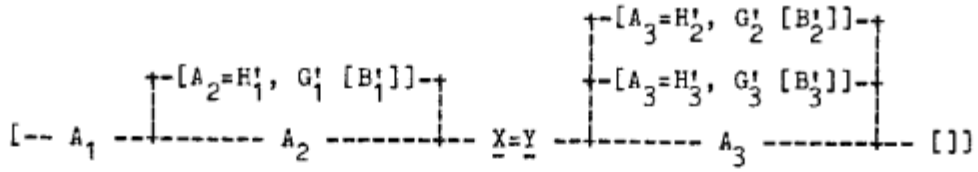
The predefined predicate '=' is used for unifying two terms. A call to the predicate '=' was called an equation in Chapter 3. This predicate should be considered as predefined, since it cannot be defined in the

language merely for syntactical reasons.

4.2. Semantics of GHC

To solve a goal clause, we repeatedly perform any of the following transformations.

(1) Choose any user-defined goal (i.e., a call to some predicate other than the predefined predicate '=') A and any program clause of the form H :- G | B where G and B are sets of goals. Then make a variant H' :- G' | B' of the chosen clause, and superimpose on A a guarded set of goals of the boxed form [A=H', G' [B']]. These two operations can be done in any manner, as long as each goal is not placed before the box directly surrounding the goal has been created. Here, a box [ ] has a semantical role of restricting dataflow, as will be stated below. We say that each goal in "A=H', G'" belongs to the outer box, and each goal in B' belongs to the inner box. Superimposing makes the original goal clause partly multi-layered, as depicted by the following diagram:



Each layer shares the other parts of the goal clause. Some layer of the multi-layered part may further become partly multi-layered. Note that for uniformity, we assume two nested boxes between which the original set of goals is initially placed.

(2) Choose any unification goal of the form S=T, and perform an appropriate transformation stated in Chapter 3 according to the forms of S and T. The algorithm of Chapter 3 must be modified as follows:

(2-1) Two general restrictions are added. Firstly, a unification goal (i.e., an equation) belonging to some box (say O) cannot rewrite a variable outside O. Such a goal, however, can rewrite variables within O (including the variables appearing in some other boxes within O), and also it can be rewritten by other goals except those belonging to some boxes within O. Secondly, when a unification

goal belonging to a box O is rewritten to a set of other unification goals, they are also placed in, and belong to, O.

(2-2) Rule (a) of Chapter 3 is modified. Even when the both sides of the equation have different principal function symbols, we do not stop the computation with failure but merely leave the equation as it is.

(2-3) An additional rule exists. Choose any unification goal (equation) E

(i) which appears in the guard G of the construct [G [B]], and

(ii) which is

- of the form $X=f(X_1^*, \ldots, X_n^*)$ where X is a variable not occurring anywhere except in B and E itself, f is some n(>=0)-ary function symbol, and X_i^*'s are all marked variables, or
- of the form X=Y where X and Y are distinct variables and X does not occur anywhere except in B and E itself.

Then move E to the body B.

(2-4) There is another additional rule. Choose any equation which belongs to a box O and which is of the form X=Y where X and Y are distinct variables, and one of the other occurrences of non-marked X outside O. Then replace X=Y by Y=X.

(2-5) The judgment of whether there are any occurrences of a variable X in some multi-layered part (say P) of a goal clause is now made as follows. Let G be the goal on the ground layer of P for which new layers have been superimposed. G must be a user-defined goal because a unification goal never creates superimposed layers. If G has not yet been committed (see below) to any of the layers, we say that a variable X appears in P iff X appears in any of the layers including the layer of the original goal G; if G has already been committed to some superimposed layer, we say that a variable X appears in P iff X appears in that layer.

(3) (The rule of commitment) Choose any layer (say L) of the form [[B]] of

some multi-layered part, that is, a layer whose head unification and guard have been reduced to an 'empty' set of goals (see below for emptiness). Then confirm that the original goal (say G) on the ground layer has been committed to no other layers. If confirmed, G is indivisibly committed to the layer L. Then, the outer box is simply deleted since now it does not impose semantical restriction on any goals, and the inner box is enlarged and amalgamated to the innermost box containing it by taking in all the symbols between these two boxes in any manner. If the original goal has already been committed to any other layer, do nothing.

An 'empty' set of goals is a set of goals consisting only of 'empty' multi-layered parts. A multi-layered part is said to be 'empty' iff the original goal G in the ground layer has been committed to some other layer L and L denotes an 'empty' set of goals.

The original goal clause between the initial two boxes is regarded as solved when it is reduced to the 'empty' clause or an 'empty' set of goals, whether the above transformation procedure terminates or not. Here, the inner box of the initial nested boxes need not be empty but may contain a set of unification goals (moved by Rule (2-3)) of the form

$$X_1=T_1, \ldots, X_k=T_k$$

where $X_1, \ldots, X_k$ are distinct variables. This set of goals has no cycles (see Section 3.3), and the most general answer substitution is given by g^k, where g is defined as follows:

$$g(X) = T_i \qquad \text{if } X=X_i \text{ for some } i=1, \ldots, k$$
$$g(X) = X \qquad \text{otherwise}$$

The final set of equations in the inner box is slightly different from the final form obtained by the unification algorithm of Chapter 3. This is because we used the concept of being solved which is different from being terminated.

4.3. Motivations and Implications

Some explanation will be necessary to clarify the motivations and the

implications of the above semantics.

First of all, the above semantics clarifies that a resolution operation can be separated into two parts: goal rewriting and head unification. Moreover, the latter can be executed in parallel with the corresponding guard goals. Spawning new layers also can be done incrementally and in parallel with the execution of already generated goals. A set of goals is regarded as solved even when some clause not selected for commitment is still being executed. Stopping unnecessary computation is considered an optimization in our framework.

The above semantics is based on parallel term rewriting. However, the new notion, superimposing, has been introduced to express OR-parallel execution of candidate clauses. Since GHC is a single-environment language like PARLOG, candidate clauses can share its outer world. On the other hand, the operational semantics of the OR-parallel execution of a usual Horn-clause program would have to express some mechanisms for multiple environments.

Boxes are used for restricting information flow caused by unification. The restriction is stated in Rule (2-1). Although the main purpose of the introduction of boxes is to express the restriction imposed on guards, it is used also for stating what can be done with the clause body before commitment. That is, the semantics allows the body B of a clause C to be executed before C is selected for commitment, as long as the execution of B never affects the execution of any goals outside B. This is a kind of backup computation, since the execution of B need not start before C is selected for commitment.

Rule (2-2) has been added because the failure detected in some guard does not mean the failure of the whole system. This rule makes the treatment of the failure of unification similar to suspension due to the dataflow restriction.

The purpose of Rule (2-3) is to make a candidate clause selected for commitment when the goals remaining in the guard have nothing to do other than to instantiate the body. Such goals no longer rewrite other goals except those in the body. Moreover, such goals themselves are never rewritten to other goals that may cause failure, since the conditions of

Rule (2-3) guarantees that the variables on the left-hand sides of those goals (of the form $X=f(X_1^*, \ldots, X_n^*)$ or $X=Y$) are never rewritten. Therefore, they can be moved to the clause body.

The purpose of Rule (2-4) is to make the unification of two variables X and Y proceed unless both X and Y appear outside the box to which the unification goal belongs. Suppose that X appears outside the box and Y does not. Then the goal X=Y cannot rewrite all the other occurrences of X, while the logically equivalent goal Y=X can rewrite all the other occurrences of Y. So in order to let the unification proceed, we have to exchange the left- and the right-hand sides. However, we do so only when the left-hand side variable appear outside the box, because arbitrary exchange would make the termination condition (not discussed in this paper) unnecessarily complex. Even under this restriction, Rule (2-4) may cause 'busy waiting' when both X and Y appears outside the box,

Rule (3) says that the dataflow restriction imposed on the body of a clause is removed when that clause is selected for commitment, but that it need not be removed instantaneously. Moreover, the removal of dataflow restriction is not an indivisible part of the commitment operation. It can be done in parallel with the execution of the clause body.

Note that although the answer substitutions are put in the inner box of the initial nested boxes, we can also observe the result by means of observation variables.

5. CONCLUSIONS

We have described the operational semantics of Guarded Horn Clauses which tries to preserve parallelism inherent in GHC as much as possible. The given semantics is anarchical and allows anything which is harmless. However, we have also discussed how to guarantee termination when the goal clause consists only of unification goals and it has a solution. It should be interesting to consider whether we can further refine the given semantics without changing the intended informal semantics behind it.

ACKNOWLEDGMENTS

The author would like to thank Akikazu Takeuchi and Satoru Tomura for helpful discussions on the earlier versions of this paper.

REFERENCES

[Apt and van Emden 82] Apt, K.R. and van Emden, M.H., Contributions to the Theory of Logic Programming, J. ACM, Vol.29, No.3, pp.841-862, 1982.

[Brock and Ackermann 81] Brock, J.D. and Ackermann, W.B., Scenarios: A Model of Nondeterminate Computation, In Formalization of Programming Concepts, J. Diaz and I. Ramos (Eds.), Lecture Notes in Computer Science, Vol.107, Springer-Verlag, New York, pp.252-259, 1981.

[Clark and Gregory 84] Clark, K.L. and Gregory, S., PARLOG: Parallel Programming in Logic, Research Report DOC 84/4, Dept of Computing, Imperial College, London, 1984.

[van Emden and de Lucena 82] van Emden, M. H. and de Lucena, G. J., Predicate Logic as a Language for Parallel Programming, In: K. L. Clark and S. A. Tarnlund (eds.), Logic Programming, Academic Press, New York, pp.189-198, 1982.

[van Emden and Kowalski 76] van Emden, M.H. and Kowalski, R.A., The Semantics of Predicate Logic as a Programming Language, J.ACM, Vol.23, No.4, pp.733-742, 1976.

[Hagiya 83] Hagiya, M., On Lazy Unification and Infinite Trees, Proc. Logic Programming Conference '83, 1983 (in Japanese).

[Martelli and Montanari 82] Martelli, A. and Montanari, U., An Efficient Unification Algorithm, ACM Trans. Prog. Lang. Syst., Vol.4, No.2, pp. 258-282, 1982.

[Shapiro 83] Shapiro, E.Y., A Subset of Concurrent Prolog and Its Interpreter, ICOT Tech. Report TR-003, Institute for New Generation Computer Technology, 1983.

[Staples and Nguyen 85] Staples, J. and Nguyen, V.L., A Fixpoint Semantics for Nondeterministic Data Flow, J. ACM, Vol.32, No.2, pp.411-444, 1985.

[Ueda 85] Ueda, K., Guarded Horn Clauses, ICOT Tech Report TR-103, Institute for New Generation Computer Technology, 1985. Also in Proc. Logic Programming Conference '85, ICOT, pp.225-236. Also to appear in Lecture Notes in Computer Science, Springer-Verlag.