ICOT Technical Memorandum: TM-0136

TM-0136

# On the Operational Semantice of Guarded Horn Clauses (Preliminary draft)

by
Kazunori Ueda

October, 1985



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
(preliminary draft)

October 1, 1985

Kazunori Ueda
Institute for New Generation Computer Technology

Abstract: An operational semantics of Guarded Horn Clauses (abbreviated to GHC) is shown. The purpose of this operational semantics is to show what are primitive operations in GHC and to show a guideline for fully distributed implementation. The most important point is that we no longer consider a variable as a centralized entity. However, many problems remain on the formalization of full parallelism.

## 1. Introduction

The language Guarded Horn Clauses [Ueda 85] is intended to be the standard of logic programming languages which allows parallel execution. It is capable of expressing important concepts in parallel programming such as processes, communication, and synchronization. GHC shares this feature with Concurrent Prolog [Shapiro 83] and PARLOG [Clark and Gregory 84], but it has one unique feature of simplicity. Guard is the only syntactic construct added to Horn clauses, and the semantics of the guard has augmented the expressive power of the original Horn clause logic to the level of a practical programming language. GHC has no multiple environments or backtracking (i.e., multiple environments expanded in a 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 Horn clause logic. However, GHC was undoubtedly born from the investigation of logic programming. Moreover, we have to use don't care nondeterminism to write a program which interfaces with a real world, and in such a case the completeness is not only unimportant but 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 binding information available, a goal generates additional binding information, and/or reduces itself to a set of goals. GHC is a generalization of nondeterministic dataflow languages in that the data structure it handles is not limited to streams, and it can handle incomplete data structures. The capability of handling incomplete data structure enables us to naturally express demand-driven computation without introducing new primitives.

In the following chapters, we first discuss three ways to define the semantics of GHC. Then we introduce a nondeterministic unification algorithm which models a parallelism of GHC better than previous ones. Lastly, we describe the semantics of GHC by extending the unification algorithm we propose.

## 2. Three Directions towards the Semantics of GHC

### 2.1 Declarative Semantics as an Logic Programming Language

The declarative semantics of a logic program which does not deal with infinite computation is well studied in [van Emden and Kowalski76] and [Apt

and van Emden 82]. [Hagiya 83] and [Lloyd 84] tries to capture the declarative semantics of infinite computation by the greatest fixpoint on the (extended) Herbrand base.

Unfortunately, because of the don't-care nondeterminism of GHC which results from the semantics of the trust operator, the semantics along this line cannot capture all the aspects of GHC. Moreover, it fails to show the causality of 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 in which no don't-care nondeterminism is involved.

## 2.2 Functional 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 other terms, depending on, and possibly after waiting for, the bindings given by other goals. In other words, GHC goals are processes interacting with one another by means of variable instantiation. 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 very promising because the obtained semantics will capture the all aspects of GHC including causality and don't-care nondeterminism in a fully abstract manner. It should provide a theoretical foundation for every kind of mechanical and manual handling of a program such as 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. The operational semantics must be moderately abstract: If it is too specific, it can serve only for a small range of implementations and one cannot distinguish between essential and inessential matters; if it is too abstract it cannot serve as a guideline of any implementation.

However, being moderately abstract is of course difficult, because there are only subjective criterion. This 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 as a proof procedure of a given formula, and in the case of a Horn-clause language it is identified as a refutation procedure. The semantics of GHC can also 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, it must clarify what can be executed in parallel, in order to serve for fully parallel implementation.

We consider fully parallel execution as a standard and serialization of primitive operations as optimization for hardware which favors sequential computation. This view is the exact opposite of the usual view of optimization. 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 computation or storage which is useless functionally may be useful for efficiency; to disallow it would be harder and/or it would cause more serious inefficiency in a distributed computation. Therefore, we decided to show what is 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 back-up computation only as

a consequence of the semantics.

## 2. A Nondeterministic Unification Algorithm

The most important and delicate operation in GHC is unification. This chapter shows the nondeterministic unification algorithm which will be incorporated in the semantics of GHC. The algorithm gives the semantics of a GHC program which comprises only of a predefined unification predicate '=' and a predefined predicate for output. The point is that we no longer consider a complex term as an atomic entity nor we consider a variable as atomic. Thus the algorithm is more nondeterministic that the nondeterministic algorithm in [Martelli and Montanari 82]. The algorithm also tries to express parallelism, though it does not fully address the problem of deadlock and starvation, while the algorithm in [Martelli and Montanari 82] is sequential.

### 3.1 The Algorithm

Our formalization basically follows [Martelli and Montanari 82]. Function symbols, variables, and terms are defined as usual. The only difference is that a set of variables is ordered.

The unification problem is a set of equations with a set of 'observartion variables' of the form

S1=T1, ..., Sn=Tn; V1, ..., Vm.

Observation variables are used for observing the result of unification. Usually, the result of unification is given as a set of substitutions after it is terminated, and most Prolog systems follows this convention. However, this is like seeing a post-mortem dump and it is not a usual way of getting results in practical programming. An observation variable is a variable through which to dynamically observe a result of computation of interest. This models stream-oriented input/output of GHC better.

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 selected equations nor atomicity of variable rewriting (in Steps (e), (f) and (g)) is violated. Unless stated otherwise, the selected equation becomes unselected when the required transformation is complete. We may attach 'marks' to variables to prevent backward rewriting. Existence or nonexistence of a mark is insignificant unless explicitly specified. The algorithm terminates if no transformation applies.

(a) Select any equation of the form S=T where S and T are not variables. If the two principal function symbols are different, unselect this atom and stop with failure. Otherwise, the equation is of the form f(S1, ..., Sn)=f(T1, ..., Tn), and rewrite it to S1=T1, ..., Sn=Tn in any way but without erasing Si's and Ti's. The condition 'without erasing ...' means that Si's and Ti's must continue to appear in the problem during rewriting.

(b) Select any equation of the form X=f(T1, ..., Tn) where f is some n(>0)-ary function symbol, Ti's are non-marked terms and X is a variable, and rewrite it to X=f(X1*, ..., Xn*), X1=T1, ..., Xn=Tn in any way but without erasing X and Ti's, where Xi's are distinct variables which are different from the variables in the current problem. The original equation X=f(T1, ..., Tn) becomes unselected when it is rewritten to X=f(X1*, ..., Xn*). Asterisks are called marks, and they are attached to the new variables to prevent backward

rewriting toward the original term.

(c) Select any equation of the form f(T1, ..., Tn)=X where f is some n(>0)-ary function symbol, Ti's are non-marked terms and X is a variable, and rewrite it to X=f(X1*, ..., Xn*), X1=T1, ..., Xn=Tn in any way but without erasing X and Ti's, where Xi's are distinct variables which are different from the variables in the current problem. The original equation f(T1, ..., Tn)=X becomes unselected when it is rewritten to X=f(X1*, ..., Xn*).

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

(e) Select any equation of the form X=Y where X and Y are distinct variables and X > Y by the given ordering, and find one of the other occurrences of non-marked X and replace it by Y with no mark. If there are no other occurrences of non-marked X then erase the selected equation.

(f) Select any equation of the form X=Y where X and Y are distinct variables and Y < X by the given ordering, and find one of the other occurrences of non-marked Y and replace it by X. If there are no other occurrences of non-marked Y then erase the selected equation.

(g) Select any equation of the form X=f(X1*, ..., Xn*) where f is an n(>=0)-ary function symbol and Xi*'s are marked variables. Then find one of the other occurrences of unmarked Y and replace it by f(X1, ..., Xn). If there are no other occurrences of unmarked X then erase the selected equation.

3.2 Examples

The following examples illustrate some subtle points in the algorithm.

(1) X=1, X=2. The first equation can terminate only after rewriting the second X to 1, and the second equation can terminate only after rewriting the first X to 2. If the second X is rewritten first, then the X=2 equation is changed to the 1=2 equation, which causes failure. If the first X is rewritten first, the X=1 equation is changed to the 2=1 equation, which also causes failure. Therefore, the order of rewriting is independent of the result.

(2) X=Y, Y=X, X=1. This example shows why the ordering of variables is important. Without ordering, the first equation may rewrite the third equation to Y=1, and then the second equation may rewrite the third equation back to X=1, and so on. This would not be remedied by a fairness assumption on the selection of variables to be rewritten.

(3) X=1, X=1, X=2. This example shows why an equation being selected must not be rewritten by some other equation. Suppose that the first equation and the second equation are simultaneously selected and each of them rewrites the other to 1=1. Then the original problem is rewritten to '1=1, 1=1, X=2', and then to 'X=2'. This is obviously an erroneous rewriting. If a selected equation is locked, this situation never occurs. However, this locking may cause deadlock as is evident from this example--The algorithm may have to be called a semi-algorithm in this sense. To avoid it would require ordering the occurrences of equations and variables in the problem, and then reordering of them would be necessary to avoid starvation. The problem of deadlock and starvation is yet to be investigated. One research direction would be

to make some of the primitive operations a little bit larger.

(4) X=1, Y=X, Y=2. This example shows another reason why an equation being selected must not be rewritten. Suppose that the second equation Y=X is selected and that Y > X. It rewrites the third equation to X=2. However, before this rewriting is completed, the first equation may (i) be selected, (ii) rewrite the second equation to Y=1, (iii) be selected again, (iv) judge that there are no other occurrences of X, and (v) erase itself. Then the original problem is reduced to Y=1, X=2. This shows together with Example (3) that no variables on either side of the selected equation must be rewritten.

3.3 Proof

(to be supplied)

3.4 Implications and Motivations

The above algorithm differs from that of [Martelli and Montanari 82] in the following two 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(1, nil) ...(i)

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

X = cons(A#, B#), A = 1, B = nil ...(ii)

where A# and B# are new variables, then the equation X=cons(A#, B#) is used for instantiation. In general, only the equation of the form X = T where T is a most general term 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 is motivated by the observation that Equation (i) and (ii) are logically the same and Equation (ii) has smaller granularity.

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. The algorithm explicitly allows such transmission, and we can use the transmitted value before the transmission is complete.

The other important point is that we do not consider variables as centralized entities but as distributed entities. This decision is motivated by the observation that the problem

X = 1, X = 2

can be considered as shorthand of

X1 = 1, X1 = X2, X2 = 2.

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 may have a local copy or a cache of some variable. The algorithm explicitly allows it, and it also says that these copies need not have the same value at the same time, as long as they become identical finally. A local copy may be instantiated by some other processor with potential delay. Such freedom is usually considered as a consequence of the strict semantics, but we took the other way.

## 4. An Operational Semantics of GHC

(Currently this chapter is stated rather informally; the detail is yet to be clarified.)

### 4.1 Syntax of GHC [Ueda 85]

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

H :- G1, ..., Gm | B1, ..., Bn.  (m >= 0, n >=0).

where H, Gi's, and Bi's are atomic formulas. H is called a clause head, Gi's are called guard goals, and Bi's are called body goals. The operator `|' is called a trust 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:

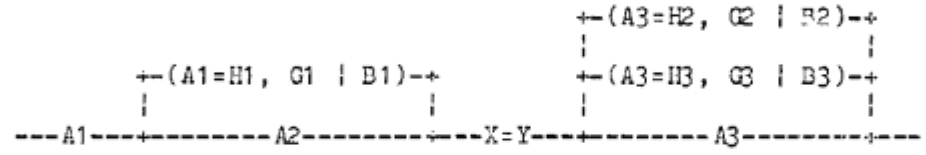
:- B1, ..., Bn.  (n >= 0).

This can be regarded as a guarded Horn clause with an empty guard. A goal clause is called an empty clause when n is equal to 0.

### 4.2 Semantics of GHC

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

(a) Select any user-defined goal (i.e., a call to some predicate other than the predefined unification goal '=') A and any program clause of the form H :- G | B. Then make a variant H' :- G' | B' of the selected clause by using variables greater (in the given ordering) than those in the current goal clause. Then superimpose on A a guarded set of goals of the form (A=H', G' | B'). This is done by first making a skeleton ( | ) on a new layer superimposed on A and then by incrementally filling its left hand side with 'A=H, G' and its right hand side with B. Superimposing makes the original goal partly multi-layered, as depicted by the following diagram:




Each layer shares the other part of the goal. Some layer of the multi-layered part may further become partly multi-layered.

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

(1) One restriction is added: The restriction is that a unification goal (i.e., an equation) appearing in the left hand side L of the form (L | R) cannot rewrite a variable outside this form, and a

unification goal appearing in the right hand side R of the form (L | R) cannot rewrite a variable in L or outside this form.

(2) The judgment of whether there are any other occurrences of a variable X is now done as follows. For a user-defined goal with superimposed layers none of which has been trusted (see below), we say that a variable X does not appear if X does not appear in any of the layers including the layer of the original goal; for a user-defined goal with a superimposed layer which has been trusted, we say that a variable X does not appear if X does not appear in that layer.

(3) An additional rule exists for the equation E of the form V=f(X1#, ..., Xn#) where Xi#'s are all marked variables. If it appears in the left hand side L of the construct (L | R) and there are no occurrences of V except in R and E itself, then move E to the right hand side R. This is for making (L | R) trusted if the only remaining task is to instantiate R.

(c) Select any layer of the form (L | R) where L is empty. Then confirm if no other layers has been trusted. If confirmed, this layer is trusted indivisibly. Then rewrite this layer to R by removing ( | ) possibly incrementally; that is, restriction of instantiation due to the construct ( | ) is removed possibly incrementally (This need further formalization).

This transformation terminates when the original goal clause is reduced to the empty clause. Here, a trusted layer is assumed to represent the multi-layered part it belongs, that is, a non-trusted layer is ignored for the judgment of termination if and only if some sibling layer has been trusted.

The above semantics clarifies that a resolution operation can be separated into two parts: goal rewriting and head unification. And the latter can be executed in parallel with the corresponding guard goals. A goal can be executed (i.e., rewritten) even before it is known what predicate it is calling. A set of goals can be said to terminate even if some untrusted clause is still being executed. Stopping unnecessary computation is considered an optimization.

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, each candidate clause can share its outer world.

The above algorithm has at least the following problems to be resolved:

(1) It requires that the variables in a renamed clause created in Step (a) be greater than those in the current goal clause. The purpose of this requirement is to ensure that the unification of a new variable and an outer variable executed in some guard may not suspend: If the new variable is greater, then it need not replace the outer variable. However, this requirement may cause serialization in the allocation of new variables. A more sophisticated rule may be necessary.
(2) The current description of Step (c) is awkward. Protection against instantiating outer variables can be abolished incrementally, but this is hard to express. One solution may be to introduce the system for managing the nesting of guards.

## 5. Conclusions

We have described an operational semantics of Guarded Horn Clauses which tries to preserve parallelism inherent in GHC as much as possible. The semantics is still preliminary in that it does not satisfactorily deal with the problem of deadlock and starvation, as well as in that a global notion such as 'a greatest variable' remains. Notational problems remain also. It is very important to give a solution to these problems: It may show the limitation of exploiting parallelism and freedom in a GHC program, which should be of interest both theoretically and practically.

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 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.