ICOT Technical Report TR-208

Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard

Kazunori Ueda

Institute for New Generation Computer Technology
4-28, Mita 1-Chome, Minato-ku, Tokyo 108 Japan

October 1986
Revised: July 1987*

Abstract. This paper defines a parallel logic programming language called Guarded Horn Clauses (GHC), introducing the ideas that went into the design of the language. It also points out that GHC can be viewed as a process description language in which input and output can be described naturally by treating the outside world as a process. Relationship between GHC and logic programming in the original, strict sense is also discussed. Some familiarity with GHC and/or other parallel logic programming languages will be helpful in reading this paper, though it is not indispensable.

1. Introduction

GHC is a parallel programming language devised from examination of the basic framework and the practice of logic programming. The most important characteristic is the simplicity of the language rules and the expressive power obtained from them. Guard is the only syntactic construct added to the framework of logic programming. Although its semantics causes loss of completeness which would be important for GHC as a theorem prover, it provides the language with a control mechanism that is a prerequisite for a general programming language.

The main purpose of this paper is to describe the language GHC informally but rigorously, but we will also try to introduce the ideas that went into the design of the language for better understanding. The original document of GHC is [7], and the most detailed description of the language is found in [9]. Ueda [10] introduces GHC through program examples.

2. Design Goals of GHC

GHC was designed as a general parallel programming language based on Horn-clause logic. By a general parallel programming language we mean a language

* To appear in *Programming of Future Generation Computers*, M. Nivat and K. Fuchi (eds.), North-Holland, Amsterdam, 1987.

in which we can describe processes and interactions among them. The important point is that we must be able to handle *interacting* processes; if we have only to deal with mutually independent processes, the subtle problems of parallel programming will not appear, but the language will be less expressive.

The above requirement means that we need some notion of control, which is in general used for introducing reasonable order among primitive operations constituting a computation. The order should of course be partial for a parallel programming language. GHC introduced partial order to (atomic pieces of) unification, a primitive operation in logic programming.

Recalling that a sequential algorithm specifies a total order on primitive operations, a *parallel algorithm* can be defined as specifying a partial order on them, and this is just what we intend to express in GHC. In this sense, GHC could be viewed as a realization of Kowalski's thesis "algorithm = logic + control" [4].

There may be various ways to achieve our purposes and to obtain a parallel programming language. Concurrent Prolog [6], PARLOG [2] and GHC share the above design goals in principle, and they all have guard as a syntactic construct. The unique feature of GHC is that it uses guard as the *only* syntactic construct, as we will describe below.

3. Syntax and Semantics

3.1. Syntax

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

$$H \;\texttt{:-}\; G_1\texttt{,}\ \ldots\texttt{,}\ G_m \mid B_1\texttt{,}\ \ldots\texttt{,}\ B_n\texttt{.} \qquad (m > 0,\ n > 0).$$

where H, G_i's, and B_i's are atomic formulas. H is called a clause head, the G_i's are called guard goals, and the B_i's are called body goals. The connective '`:-`' means 'is implied by', and '`,`' means conjunction. The only difference from an ordinary Horn clause is that one of the conjunctive operators is replaced by a commitment operator '`|`'. The part of a clause before '`|`' is called a guard, and the part after '`|`' is called a body. Note that the clause head is included in the guard. Declaratively, the commitment operator denotes conjunction, and the above guarded Horn clause is read as "H is implied by $G_1, \ldots, G_m$ and $B_1, \ldots, B_n$".

We use a goal clause of the following form to start a GHC program:

$$\texttt{:-}\; B_1\texttt{,}\ \ldots\texttt{,}\ B_n\texttt{.} \qquad (n > 0).$$

This is just an ordinary Horn clause used in the original framework.

We follow the syntactic convention of DEC-10 Prolog [1] that begins constant, function and predicate symbols by lowercase letters or symbols and variables by uppercase letters.

One binary predicate, '`=`', is predefined by the language. The predicate '`=`' is used for unifying two terms. This predicate is considered as predefined, since it cannot be defined in the language. However, the reason for this is purely notational and we need not regard it as the second additional construct, as will be explained in Section 3.2.

The nullary predicate `true` is used for denoting an empty set of guard or body goals explicitly in a program. The use of `true` is for notational convenience and is not essential at all. Actually it could be defined as follows:

```
true :- 1=1 | 1=1.
```

3.2. Semantics

The semantics of GHC is described in two stages: (i) parallel input resolution as a basis, and (ii) a restriction to a single environment imposed on it. The second stage describes the differences between the original framework and GHC by introducing the semantics of the distinction between guard and body. The first stage, parallel input resolution, is not specific to GHC, so we first describe it for a set of *ordinary* Horn clauses. The description will then be slightly modified for use as the basis of GHC.

The resolution strategy normally used as the basic framework of logic programming is SLD-resolution [5]. However, it is too specific as a framework of parallel execution of logic programs and this is why we begin with defining parallel input resolution.

We first consider a *proof tree* constructed from a program P and a goal clause C defined as follows:

(1) The root represents the goal clause C, and has an outgoing arc for each body goal.

(2) Each non-root node represents a program clause in P, and has one incoming arc from its parent node and an outgoing arc for each body goal.

Henceforth we may identify a node with the clause it represents. Note that the leaves of a tree represent unit clauses.

Each arc A of a proof tree is considered as representing an equation $G = H$, where G is the body goal (of the parent node) corresponding to A and H is the head of the child node C. The goal G is called the *caller* of C, and G is said to *call* C.

The set of all the arcs in a proof tree defines a unification problem. The *answer substitution* of a proof tree is the most general unifier of the unification problem. A proof tree may not have an answer substitution; a proof tree that has an answer substitution is called a *non-ordered refutation*. Note that there may be many possible proof trees constructed from a given program and a goal clause, since any program clause can be a child of a given parent node.

Our purpose is to obtain non-ordered refutations and their answer substitutions for a given program and a goal clause, which is achieved by performing the following two tasks:

(1) production of possible proof trees, and

(2) solution of the unification problem defined by each proof tree.

The proof procedure may exploit parallelism in the above tasks. Firstly, it may exploit parallelism in constructing *a* non-ordered refutation: A proof tree may be constructed in parallel, the associated unification problem may be solved in parallel, and these two tasks may be performed in parallel. This proof method is called *parallel input resolution*, and the parallelism of this kind is often referred to as AND-parallelism. The primitive operation in the construction of a proof tree is to provide a body goal with a renamed program clause, leaving unification of the goal and the clause head as a separate operation. The primitive operations in solving a unification problem are those in the unification algorithm employed. We do not specify a particular unification algorithm, but only request that the algorithm be *correct*, that is, it calculates the most general unifier if and only if the original unification problem has it.

Secondly, the proof procedure may exploit parallelism in getting *different* non-ordered refutations. Parallelism of this kind is often referred to as OR-parallelism.

Amending the above framework for GHC requires the following modification. Firstly, each non-root node representing a guarded Horn clause must have an outgoing arc for each guard goal as well as for each body goal. Secondly, the predefined predicate '`=`' is treated as if there were a unit clause "`X = X.`" whose contents are semantically divided into two parts: The predicate name '`=`' and its arity are treated like the entities in a guard, and the identity of the two arguments are treated like the entities in a body.

Now we give the semantics of the distinction between guard and body by restricting the above parallel input resolution.

In the above framework, different proof trees constructed from a given program and a goal clause are treated independently. A variable in the goal clause may be bound to different values in different proof trees, and this is why OR-parallel Prolog requires implementation of multiple binding environments. Backtracking in sequential Prolog is also regarded as a sequential implementation of multiple environments.

On the other hand, GHC treats all proof trees in a single environment rather than independently. For this purpose, it disallows unification performed in two proof trees to instantiate *corresponding* variables to different values, where the correspondence of entities in different proof trees is defined as follows. Given two proof trees T_1 and T_2 constructed from a given program and a goal clause, we say

(i) that a non-root node in T_1 and a non-root node in T_2 correspond if they represent identical program clauses (up to renaming) and their parent nodes correspond, and

(ii) that the roots of T_1 and T_2 correspond.

Further we say that two variables correspond if they appear in the same position of the (identical) program clauses represented by corresponding nodes. If corresponding variables are never instantiated differently, it is unnecessary for corresponding nodes to represent independent program clauses. Therefore, in the following we assume that corresponding nodes *share* the program clause they represent.

The purpose of the above single environment restriction is to let any substitution be common and global throughout the execution of a program and to let it be determinate and unnecessary to revoke. This restriction forces GHC to abandon completeness as a proof procedure. However, disallowing nondeterminate bindings greatly simplifies a binding-oriented view of the execution of logic programs discussed in detail in Section 4.

The simplest way to achieve the single environment restriction is to restrict resolution so that no bindings may be generated, but it will severely limit the expressive power of the language. To obey the restriction while letting the language be still useful, we impose the following rules of suspension:

- *Rules of Suspension*

  (a) Unification invoked directly or indirectly in the guard of a clause C called by a goal G (i.e., unification of G with the head of C and any unification invoked by solving the guard goals of C) cannot instantiate the goal G.

  (b) Unification invoked directly or indirectly in the body of a clause C called by a goal G cannot instantiate the guard of C or G *until C is selected for commitment* (see below).

  A piece of unification that can succeed only by causing such instantiation is suspended until it can succeed without causing such instantiation *(end of the rules of suspension)*.

Here, we assume that any substitution (legally) generated by a piece of unification is applied to the proof trees, instantiating the clauses represented by their nodes. This means that the equations defined by the proof trees are always getting instantiated.

For the clause defining the predicate '`=`', the above rules are understood as follows: Unification between the clause "`X = X.`" and its caller G (*which does not* *necessarily call the predicate* '`=`') cannot instantiate G until this clause is selected for commitment.

A suspended piece of unification may (but cannot always) be resumed when some other unification goal running in parallel has instantiated the variable that caused suspension to some term. For example, if the guard of C tries to unify `X` appearing in G with `a`, the unification suspends due to Rule (a) above. It can be resumed and succeeds when some other goal instantiates `X` to `a`. If the guard of C tries to unify `X` and `Y` both appearing in G, then the unification can be resumed and succeeds either when `X` and `Y` are unified together and become identical or when both `X` and `Y` are bound to an identical non-variable term.

Another rule we must have is *the rule of commitment:*

- *Rule of Commitment*

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

For the clause defining the predicate '`=`', the above rule is understood as follows: When it is confirmed that a goal G calls the binary predicate '`=`', the clause "`X = X.`" tries to be selected for establishing the identity of the arguments by unifying them.

We say that a part P of a program clause C called by a goal G or of a goal clause C *succeeds* (or is *solved*) if the proof procedure succeeds in constructing the part of a proof tree consisting of

(i) the arcs corresponding to the body goals in P,

(ii) the subtrees having the child nodes of those arcs as their roots, and

(iii) if C is a program clause and P contains the head H of C, the arc corresponding to the unification of G with H

such that

(a) the set of the arcs in it has a most general unifier (under the rule of suspension) and

(b) the nodes in it all correspond to selected clauses.

As a special case, we say that a goal clause succeeds if the proof procedure succeeds in constructing a non-ordered refutation whose non-root nodes all correspond to selected clauses. We do not say that a goal clause succeeds even if the proof procedure constructs a non-ordered refutation with a non-root node representing a non-selected clause: We are interested in a proof in which only selected clauses are involved. The above definition does not introduce the notion of failure. We could introduce it, but it is not necessary for ordinary applications.

It must be stressed that under the rules stated above, anything can be done in parallel. To put it precisely, parallel execution is the basic computation strategy in GHC, whether performed by a parallel computer or simulated by a sequential computer. Conjunctive goals (i.e., goals in the same proof tree) are solved in parallel; candidate clauses called by a goal (that belong to different proof trees) compete in parallel for commitment; unification of a goal with the head of a candidate clause is done in parallel, both internally and with the execution of guard goals.

However, it must be even more stressed that we can also execute a set of operations in a predetermined order as long as it does not affect the result. There are two possible ways in which pre-ordering of operations makes difference. One is that sequential execution may suspend forever on a piece of unification which parallel execution would sometime make succeed. For example, consider the following program

```
p(ok) :- true | true.          (3–1)
q(Z)  :- true | Z=ok.          (3–2)
```

and a goal clause "`:- p(X),q(X).`"; and suppose we solve the goal clause sequentially from left to right, moving to the next goal when and only when the previous goal has succeeded. Then the goal `p(X)` will suspend (because the the head `p(ok)` of Clause (3–1) would otherwise instantiate it) and the execution will result in deadlock. However, if we solve `p(X)` and `q(X)` in parallel, sometime the goal `q(X)` will select Clause (3–2). Then the body goal of Clause (3–2) will bind the shared variable `X` to `ok`, and the goal `p(X)`, now instantiated to `p(ok)`, will select Clause (3–1). This means that sequential execution of conjunctive goals is not a legal execution strategy in general.

The second possible difference is that sequential execution may be trapped by infinite computation when parallel execution could avoid it. For example, suppose we solve candidate clauses for a goal G sequentially, moving to the next clause when the previous clause turns out to be unselectable forever or when it has to wait for instantiation of G. Then, if the guard of some clause falls into infinite computation (i.e., construction of an infinite proof tree) trying to be solved, subsequent clauses whose guards could be solved and selected will not be executed forever.

In a word, serialization of operations is allowed when and only when it does not make an execution trapped by unresumable suspension or infinite computation. For example, we can exploit sequentiality between the guard and the body of a clause, and between the head and the guard goals of a clause. Serialization may be useful for avoiding scheduling overhead of pseudo-parallel execution and computation that may not contribute to the success of a top-level goal clause.

The above rules of suspension and commitment guarantee that among program clauses called by a goal G, only one that is selected for commitment can instantiate G (by executing its body goals). Thus only one binding environment need be managed. The guard of a clause is entirely passive and never instantiates G. The guard of a clause suspends until G gets enough binding information to select that clause, and then the body generates bindings back to the caller. This suspension mechanism is very important because due to the single environment restriction, commitment can never be revoked and hence must be done deliberately based on sufficient information.

Unlike an ordinary Horn clause, a guarded Horn clause explicitly specifies the direction of computation. For instance, a program for concatenating two lists cannot be used for dividing a list into two. If a program is used against its intended direction, it simply suspends.

4. GHC as a Process Description Language

It is said that the main purpose of a logic programming system is to compute answer substitutions for a given goal [5], but in the original framework, its success or failure should be important as well. In GHC, we are more interested in bindings generated by computation. Most programs are defined so that they will successfully solve a given goal, and a program that does not succeed for a 'correct' goal (due to deadlock or unsuccessful unification) is usually erroneous.

Therefore, it is quite natural to view a GHC program in terms of binding information and the agents that observe and generate it. Consider the following program:

```
gen(N,Ns) :- true | Ns=[N|Ns1], N1:=N+1, gen(N1,Ns1).        (4–1)
```

This clause defines an eager generator of a sequence of integers. Once we generate a goal `gen(1,Xs)`, `Xs` will be gradually instantiated and will approximate an infinite list of integers beginning with `1`. This goal generates bindings autonomously, since the guard of Clause (4–1) called by the top-level and recursive goals can always succeed immediately. However, if Clause (4–1) is rewritten as follows, the behavior of the goal will be quite different:

```
gen(N,[M|Ns1]) :- true | M=N, N1:=N+1, gen(N1,Ns1).          (4–2)
```

Now the goal `gen(1,Xs)` will suspend as long as no other goal instantiates `Xs`, since unifying `Xs` with `[M|Ns1]` in the guard would instantiate `Xs` and violate the rules of suspension. However, if some conjunctive goal (say `p`) has instantiated `Xs` to `[X1|Xs1]`, then `gen(1,Xs)`, which has now become `gen(1,[X1|Xs1])`, selects

Clause (4–2) and binds `X1` to `1`. If `p` has further instantiated `Xs1` to `[X2|Xs2]`, `gen(1,Xs)` will bind `X2` to `2`. Thus, the goal `gen(1,Xs)` cannot create a list structure by itself, but it fills a given list structure with successive integers. It can be called a demand-driven (lazy) generator of a sequence of integers, where instantiating (a sublist of) `Xs` to the structure `[`*Car*`|`*Cdr*`]` can be regarded as a demand. Note that Clauses (4–1) and (4–2) are equivalent as logical formulae; the difference comes from whether the second argument of the caller is unified with the list structure in the guard or in the body.

In general, a goal can be viewed as a process that observes input bindings and generates output bindings according to them. Observation and generation of bindings are the basis of computation and communication in our model. The behavior of a process is defined by program clauses using other processes, possibly recursively. A program clause can be viewed as a process rewrite rule. Consider Clause (4–2). Its guard specifies the conditions for commitment: The predicate must be '`gen`' with two arguments and the second argument must have the list structure `[`*Car*`|`*Cdr*`]`. If both conditions are satisfied, this clause can be selected; and if selected, the original goal is replaced by the three goals `M=N`, `N1:=N+1`, and `gen(N1,Ns1)`. The goal `M=N` determines the value of the first element of the list, `N1:=N+1` computes the value for the next element, and `gen(N1,Ns1)` generates the sublist `Ns1` in a similar way on demand. Strictly speaking, the commitment does not mean that the original goal disappears from a proof tree; the point is that its function is realized by the three subgoals.

In the GHC framework, interprocess communication is done using shared variables, but it is quite different from shared-variable communication in procedural languages. The difference comes from the single-assignment (as opposed to destructive assignment) property of logical variables, which contributes much to conceptual simplicity. Since we have a natural notion of an undefined value, we can realize synchronization by letting a process wait until necessary values are known. No other means need be provided for synchronization. A shared variable between recursive processes is usually instantiated to a list of data or messages gradually as computation proceeds, and we call such a shared variable a *stream*. That a sequence of data or messages is just a data structure also contributes to the simplicity. In most procedural languages, sequences of communication messages are implicit and must be manipulated using a special set of operations.

One might feel it inconvenient that many-to-one communication requires explicit stream merging defined by the following predicate:

```
merge([A|Xs],Ys,    Zs) :- true | Zs=[A|Zs1], merge(Xs,Ys,Zs1).
merge(Xs,    [A|Ys],Zs) :- true | Zs=[A|Zs1], merge(Xs,Ys,Zs1).
merge([],    Ys,    Zs) :- true | Zs=Ys.
merge(Xs,    [],    Zs) :- true | Zs=Xs.
```

Many-to-one communication serializes messages from two or more sources, which should involve arbitration on the arrival order of the messages. This means that any programming language that allows implicit many-to-one communication does arbitration implicitly. However, since the concept of arbitration does not exist in the original framework of logic programming, we had to include it in the form of the commitment operator.

A process can also be viewed as performing data-driven computation in the sense that it generates output data when necessary input data are available. For example, the goal `N1:=N+1` does nothing until `N` is instantiated, and binds `N1` to `6` when `N` is bound to `5`. Although the predicate '`:=`' will be defined as a system predicate for reasons of efficiency and convenience, exactly the same behavior can be obtained using the following clause:

```
N:=5+1 :- true | N=6.
```

GHC has a unique feature of uniformity and flexibility as a dataflow language. It naturally contains the notion of non-strict data structures. It handles data-driven and demand-driven computation in the same framework. It separates static program structures and possibly dynamic process structures. It allows us to define mutable objects (such as arrays) as processes completely in a declarative framework.

Finally, we note that the features of GHC listed in this section apply also to other parallel logic programming languages including PARLOG, Concurrent Prolog, and Oc [3].

5. Interaction with the Outside World

We pointed out in Section 4 that the main purpose of GHC programs is to compute binding information. However, observing an answer substitution returned by the system is not the normal way of interacting with a large program. It may be good for a query language which allows only a very restricted form of input/output, but is not appropriate for more general input/output required in interactive programs. It looks like the logic programming counterpart of the post-mortem dump facilities in conventional operating systems. A method for normal input/output must be provided separately from it.

In the GHC framework, the most appropriate input/output method is to include in a goal clause a goal (i.e., a process) modeling the outside world. This is quite natural because the computer and the outside world always run concurrently. The outside world includes everything outside the proof procedure; it includes the operating system that manages and performs physical input/output, peripheral devices, and a human being in front of the terminal. It is important to note that the outside world *participates* in the proof procedure instead of *observing* it.

Treating the outside world as a process means that we treat interprocess communication and input/output in the same framework. In addition to the conceptual simplicity and uniformity, this contributes to the modularity and reusability of programs. For instance, we can easily redirect a stream of output data to other processes for further processing.

The specification of the goal modeling the outside world should be defined by each system. In general, the argument of the goal will represent request messages to the operating system and actual input/output data. The request messages will be used for specifying a device, an access method, and so on. There will be various access methods for input/output, since a method appropriate for one application may be inappropriate for another. For instance, input data from a magnetic tape will be best formulated as a stream instantiated in a data-driven manner, while input data from a terminal may be better formulated as responses to requests issued by a program. Thus, full-fledged input/output facilities for GHC may not be very simple, but still it will achieve a certain simplification in that everything including data-driven and demand-driven input is realized within the basic framework of the language.

6. GHC as a Logic Programming Language

This section examines GHC as a logic programming language. The operational semantics of GHC described in Section 3.2 is a sound proof procedure for a Horn-clause program; if a goal clause " `:-` B_1`,` ... `,` B_n`.`" succeeds with an answer substitution θ, $\forall(B_1\theta\texttt{,}\ \ldots\texttt{,}\ B_n\theta)$ is a logical consequence of the program. Unfortunately, it is not a complete proof procedure. The incompleteness is due to the single-environment restriction which means only one solution can be returned. However, not a few of logic programs we write are deterministic when used in their intended directions, i.e., each of them is expected to return a single solution which is not affected by the choice nondeterminism involved in commitment. Moreover, it is unlikely that one can write a correct and efficient program without thinking of how the values of variables are determined, whether the program is intended to return single or multiple solutions. So it is worth noting that as long as we write a deterministic logic program with a correct specification of the intended data flow (using the guard/body distinction) and as long as we use it in the intended direction it expresses, the operational semantics of GHC is complete.

There are two cases where a program cannot apparently be deterministic. One is where we want to use don't-care (choice) nondeterminism intentionally. In this case, the loss of possible solutions due to commitment is just what the programmer wants to specify, so incompleteness is not a defect. The other case is where we really want to search all possible solutions, that is, where we don't want to have guard and commitment. Ordinary Horn clauses should be a better language for this purpose, but we can relate these two languages by a translation technique.

Ueda [8] proposed a technique for compiling an exhaustive search program in ordinary Horn clauses into a deterministic GHC program. Collecting all solutions requires the primitives like `setof` and `bagof` of DEC-10 Prolog whose semantics cannot be explained only by the semantics of first-order logic programming. The above technique compiles away such primitives without introducing any extralogical features, and establishes ordinary Horn clauses as a user language on top of GHC for simple search problems.

GHC has a greater expressive power as a programming language than the original framework, but of course this means that the semantics of GHC is less simple and that mechanical handling of programs such as program transformation is less easy. Nevertheless, being based on logic programming is an advantage of GHC: A program allows declarative reading as a logical formula, and the description of the semantics is made quite simple using the terms of theorem proving.

GHC has no notion of the order of program clauses or the order of goals in a clause, except that each clause has a guard and a body. For this reason, many extralogical built-in predicates of Prolog whose behavior is order- (or time-) dependent cannot make any sense and are excluded from GHC. Among such predicates are input/output predicates à la Prolog; other examples are predicates like `var` and '`==`' used for knowing the 'current' state of instantiation. We have found that GHC encourages programmers to write better logic programs; programmers never make inadvertent use of extralogical features that often happens in Prolog programming.

References

[1] Bowen, D. L. (ed.), Byrd, L., Pereira, F. C. N., Pereira, L. M. and Warren, D. H. D., DECsystem-10 Prolog User's Manual. Dept. of Artificial Intelligence, Univ. of Edinburgh, 1983.

[2] Clark, K. L. and Gregory, S., PARLOG: Parallel Programming in Logic. *ACM Trans. Prog. Lang. Syst.*, Vol. 8, No. 1 (1986), pp. 1–49.

[3] Hirata, M., Letter to the editor. *Sigplan Notices*, Vol. 21, No. 5 (1986), pp. 16–17.

[4] Kowalski, R., Algorithm = Logic + Control. *Comm. ACM*, Vol. 22, No. 7 (1979), pp. 424–436.

[5] Lloyd, J. W. *Foundations of Logic Programming*. Springer-Verlag, Berlin Heidelberg New York Tokyo, 1984.

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

[7] Ueda, K., Guarded Horn Clauses. ICOT Tech. Report TR-103, Institute for New Generation Computer Technology, Tokyo, 1985 (revised in 1986). Also in *Proc. Logic Programming '85*, Wada, E. (ed.), Lecture Notes in Computer Science 221, Springer-Verlag, Berlin Heidelberg New York Tokyo, 1986, pp. 168–179. Also to appear in *Concurrent Prolog: Collected Papers*, Vol. 1, Shapiro, E. Y. (ed.), The MIT Press, Cambridge, Mass, 1987.

[8] Ueda, K., Making Exhaustive Search Programs Deterministic. *New Generation Computing*, Vol. 5, No. 1 (1987), pp. 29–44.

[9] Ueda, K., *Guarded Horn Clauses*. Doctoral thesis, Information Engineering Course, Faculty of Engineering, Univ. of Tokyo, 1986.

[10] Ueda, K., Introduction to Guarded Horn Clauses. ICOT Tech. Report TR-209, Institute for New Generation Computer Technology, Tokyo, 1986.