ICOT Technical Report: TR-103

TR-103

GUARDED HORN CLAUSES

by

Kazunori Ueda

July, 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

GUARDED HORN CLAUSES

Kazunori Ueda
C&C Systems Research Laboratories, NEC Corporation *

ABSTRACT

A set of Horn clauses, augmented with a `guard' mechanism, is shown to be a simple and yet powerful parallel logic programming language.

1. INTRODUCTION

A set of Horn clauses allows procedural interpretation [Kowalski 74]. It was given a semantics as a sequential programming language by Prolog [Roussel 75], and Prolog has proved to be a simple, powerful, and efficient sequential programming language [Warren et al. 79].

As [Kowalski 74] points out, a Horn clause program allows parallel or concurrent execution as well as sequential execution. However, although a set of Horn clauses may be useful for uncontrolled search as it is, it is inadequate for a parallel programming language which is capable of describing important concepts such as communication and synchronization. We need some additional mechanism to express these concepts. This paper shows that only one construct, `guard', is adequate for our purposes.

In the following chapters, we introduce guarded Horn clauses. The name Guarded Horn Clauses (abbreviated to GHC) will be used also as the name of our language. Comparisons of GHC with other logic/parallel programming languages are included.

The language GHC is intended to be the machine-independent core of the Kernel Language for ICOT's Parallel Inference Machine.

2. DESIGN GOALS AND OVERVIEW

Our goal is to obtain a logic programming language that allows parallel execution. It is expected to fulfill the following requirements.

(1) It must be a parallel programming language `by nature'. It must not be a sequential language augmented with primitives for parallelism. That is, the language must assume as little sequentiality between primitive operations as possible, in order to preserve parallelism inherent in a Horn clause program. This might lead to a clearer formal semantics, as well as to an efficient implementation on a novel architecture in the future.

(2) It must be an expressive, general-purpose parallel programming language. In particular, it must be able to express important concepts in parallel programming--processes, communication, and synchronization.

(3) It must be a simple parallel programming language. We do not have much experience with either theoretical or pragmatic aspects of parallel programming. Therefore, we must first establish a foundation of parallel programming on a simple language.

(4) It must be an efficient parallel programming language. We have a lot of simple, typical problems to be described in the language as well as complex ones. It is very important that such programs run as efficiently as the comparable ones in existing parallel programming languages.

Concurrent Prolog [Shapiro 83] and PARLOG [Clark and Gregory 84a] seem to lie near the solution. Both realize processes by goals, and communication by streams. Synchronization is realized by read-only variables in Concurrent

* Author's current address: Institute for New Generation Computer Technology, 1-4-28, Mita, Minato-ku, Tokyo, 108 Japan.

Prolog and by one-way unification in PARLOG.

GHC inherits the `guard' construct and the programming paradigm founded by these languages. What is the most characteristic with GHC is that the guard is the only syntactic construct added to Horn clauses. In GHC, synchronization is realized by the semantic rules of a guard.

GHC is expected to fulfill all the above requirements. We have succeeded in rewriting most of our Concurrent Prolog programs. Miyazaki and Ueda have independently written a GHC-to-Prolog compiler in Prolog by modifying the different versions of Concurrent Prolog compilers on top of Prolog [Ueda and Chikayama 84][Ueda and Chikayama 85].

3. SYNTAX AND SEMANTICS

3.1 Syntax

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 that are defined as usual. 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 `|' (including the head) is called a passive part or a guard, and the part after `|' is called an active part or a body. A set of all clauses whose heads have the same predicate symbol with the same arity is called a procedure. Declaratively, the above guarded Horn clause should be read as "H is implied by G1, ..., and Gm and B1, ..., and Bn".

A goal clause has the following form:

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

This can be regarded as a guarded Horn clause with no passive part. A goal clause is called an empty clause when n is equal to 0.

In this paper, we use symbols beginning with uppercase letters for variables and ones beginning with lowercase letters for function and predicate symbols, following DECsystem-10 Prolog [Bowen et al. 83]. The nullary predicate `true' is used for denoting an empty set of guard or body goals.

3.2 Semantics

The semantics of GHC is quite simple. Informally, to execute a program is to reduce a given goal clause to the empty clause by means of input resolution using the clauses constituting the program. This can be done in a fully parallel manner under the following rules of suspension:

o Rules of Suspension

(a) Any piece of unification invoked directly or indirectly in the passive part of a clause cannot bind a variable appearing in the caller of that clause with
(i) a non-variable term or
(ii) another variable appearing in the caller
until that clause is trusted (see below).

(b) Any piece of unification invoked directly or indirectly in the active part of a clause cannot bind a variable appearing in the passive part of that clause with
(i) a non-variable term or
(ii) another variable appearing in the passive part
until that clause is trusted.

A piece of unification which can succeed only by making such bindings is suspended until it can succeed without making such bindings or until it turns out to fail.

Another rule we have to add is the trust rule. When some clause succeeds in solving its guard goals, that clause tries to be trusted. To do so, it must first confirm that no other clauses belonging to the same procedure have been trusted for the same call. If confirmed, that clause is trusted indivisibly.

We say that a set of goals succeeds (or is solved) if it is reduced to the empty set of goals by using only trusted clauses.

It must be stressed that under the rules stated above, anything can be done in parallel: Conjunctive goals can be executed in parallel; candidate clauses for a goal can be tested in parallel; unification used for resolution can be done in parallel; head unification and the execution of guard goals can be done in parallel. However, it would have to be even more stressed that we can also execute a set of tasks in a predetermined order as long as it does not change the meaning of the program.

The rules of suspension could be more informally restated as follows:

(a) The passive part of a clause cannot export any bindings to (or, make any bindings which is observable from) the caller of that clause before trust, and
(b) the active part of a clause cannot export any bindings to (or, make any bindings which is observable from) the passive part of that clause before trust.

Rule (a) is used for synchronization, so it could be called the rule of synchronization. Rule (b) is rather tricky; it states that we can solve the body of a non-trusted clause. However, the above restrictions guarantee that this never affects the selection of candidate clauses nor the other goals running in parallel with the caller of the clause. So Rule (b) is effectively the rule of sequencing.

In Concurrent Prolog, the result of unification which is performed in a passive part before trust and which would export a binding is recorded locally. In GHC, such unification simply suspends instead. Suspension of unification due to some passive part may be released when some goal running in parallel with the one for which the passive part is executed has instantiated the variable that caused suspension.

An example may help understanding the rules of suspension. Let us consider the following program:

```
Goal:     :- p(X), q(X).                (i)
Clauses:  p(ok) :- true | ... .         (ii)
          q(Z) :- true | Z=ok.          (iii)
```

Clause (ii) cannot instantiate the argument X of its caller to the constant \`ok', since this unification is executed in the passive part. This clause has to wait until X is instantiated to \`ok' by some other goal. On the other hand, Clause (iii) can instantiate X to \`ok' after it is trusted, and this clause can be trusted almost immediately. Therefore, no matter which of the two goals in Clause (i) starts first, the head unification of Clause (ii) can succeed only after the \`Z=ok' goal in Clause (iii) has completed.

The semantics of the following program should be more carefully understood:

```
Goal:     :- p(X), q(X).                (i)
Clauses:  p(Y) :- q(Y) | ... .          (ii')
          q(Z) :- true | Z=ok.          (iii)
```

To solve the passive part of Clause (ii'), we have to do two things in parallel: unifying X and Y (i.e., parameter passing), and solving q(Y). Let us first assume that parameter passing occurs first. Then the goal q(Y) tries to unify Y (which is now identical to X) with \`ok'. However, this unification cannot instantiate X because it is indirectly invoked by the passive part of Clause (ii').

Let us then consider the other case where the goal q(Y) is executed prior to parameter passing. The variable Y is bound to `ok' because this itself does not export binding to the caller, p(X), of Clause (ii'). However, this binding causes the subsequent parameter passing to suspend because it would export binding. Hence, no matter which case actually happens, Clause (ii') behaves equivalently to Clause (ii) as for bindings given to the variable X.

Some important consequences of the above rules follow:

(1) Any unification operation which is intended to `export' bindings through head arguments to the caller of the clause must be specified in the active part. Such unification can never be specified only by the unification of a clause head; it must be done by calling directly or indirectly the predefined predicate `=' which unifies its two arguments. The predicate `=' cannot be defined in the language and should be considered as a predefined predicate.
(2) The unification of head arguments and the execution of guard goals can be executed in parallel. That is, the execution of the guard goals can start before the unification of the head arguments has completed. However, the usual way of execution that solves the guard goals only after the head unification is also allowed; it does not change the meaning of a program.
(3) The execution of the active part of a clause may, but need not, start before that clause has been trusted. The bindings made by the active part is unobservable from the passive part before trust, so the meaning of the program remains the same whether the active part starts before or only after trust.
(4) The unification of the head arguments of a clause may, but need not, be executed in parallel. It can be executed sequentially in any predetermined order.
(5) We need not implement a multiple environment mechanism, a mechanism for binding each variable with more than one value. This mechanism is in general necessary when more than one candidate clause for a goal is tried in parallel. In GHC, however, at most one clause, a trusted clause, can export bindings, thus eliminating the need of a multiple environment mechanism.

4. PROGRAM EXAMPLES

4.1. Binary Merge

```
merge([A|X], Y,     Z) :- true | Z=[A|W], merge(X, Y, W).
merge(X,     [A|Y], Z) :- true | Z=[A|W], merge(X, Y, W).
merge([],    Y,     Z) :- true | Z=Y.
merge(X,     [],    Z) :- true | Z=X.
```

The call `merge(Xs, Ys, Zs)' merges two streams Xs and Ys (implemented as lists) into one stream Zs. This is an example of nondeterministic programs. The language rules of GHC do not state that the selection of clauses should be fair. In a good implementation, however, the elements of Xs and Ys is expected to appear on Zs almost in the order of arrival.

Note that no binding can be exported from the passive part; the binding to Z must be done in the active part. This programming style, however, serves to show causality clearly. In most cases, the bi- (or multi-) directionality of a logic program is only an illusion; it seems far better to specify the data flow which we have in mind and to enable us to read it from a given program.

Note that the declarative reading of the above program gives the usual, logical specification of the nondeterministic merge--arbitrary interleaving of the two input streams makes the output stream.

4.2. Generating Primes

```
primes(Max, S) :- true | gen(2, Max, N), sift(N, S).

gen(N, Max, S) :- N <  Max | S=[N|S1], M := N+1, gen(M, Max, S1).
gen(N, Max, S) :- N >= Max | S=[].

sift([P|L], S) :- true | S=[P|S1], filter(P, L, K), sift(K, S1).
sift([],    S) :- true | S=[].

filter(P, [Q|L], K) :- Q mod P=:=0 |            filter(P, L, K).
filter(P, [Q|L], K) :- Q mod P=\=0 | K=[Q|K1], filter(P, L, K1).
filter(P, [],    K) :- true        | K=[].
```

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

The binary predicate `:=' evaluates its right-hand side operand as an integer expression and unifies the result with the left-hand side operand. The binary predicate `=:=' evaluates its two operands as integer expressions and succeeds iff the results are the same. These predicates cannot be replaced by the `=' predicate because `=' never evaluates its arguments. The predicate `=\=' is the negation of `=:='.

The readers may wish to improve the above program by eliminating unnecessary filtering.

4.3. Bounded Buffer Stream Communication

```
test(N) :- true | buffer(N, H, T), ints(0, 100, H), consume(H, T).

buffer(N, H, T) :- N > 0 | H=[_|H1], N1:=N-1, buffer(N1, H1, T).
buffer(N, H, T) :- N=:=0 | T=H.

ints(M, Max, [N|L]) :- M <  Max | N=M, M1:=M+1, ints(M1, Max, L).
ints(M, Max, [N|_]) :- M >= Max | N=`EOS'.

consume([H|Hs], P) :- H\=`EOS' | P=[_|Ts], consume(Hs, Ts).
consume([H|Hs], P) :- H =`EOS' | P=[].
```

The above program uses the bounded-buffer concept first shown in [Takeuchi and Furukawa 83]. The predicate `ints' returns a stream of integers through the third argument in a lazy manner; it never generates a new box by itself. It only fills a given box created elsewhere with a new value. In the above program, the goal `consume' creates a new box by the goal `P=[_|Ts]' every time it has consumed the top element H of the stream. The top and the tail of the stream are initially related by the goal `buffer'.

4.4. Meta-Interpreter of GHC

```
call(true   ) :- true | true.
call((A, B)) :- true | call(A), call(B).
call(A      ) :- clauses(A, Clauses) |
    resolve(A, Clauses, Body), call(Body).

resolve(A, [C|Cs], B) :- melt_new(C, (A :- G|B2)), call(G) | B=B2.
resolve(A, [C|Cs], B) :- resolve(A, Cs, B2) | B=B2.
```

This program is a GHC version of the Concurrent Prolog meta-interpreter in [Shapiro 84]. The predicate `clauses' is a system predicate which returns in a `frozen' form [Nakashima et al. 1984] a list of all clauses whose heads are potentially unifiable with the goal A. Each frozen clause is a ground term in which original variables are indicated by special constant symbols, and it is melted in the guard of the first clause of `resolve' by `melt_new'. The goal

```
    melt_new(C, (A :- G|B2))
```

creates a new term (say T) from a frozen term C by giving a new variable for each `frozen' variable in C, and tries to unify T with (A :- G|B2).

The predicate `resolve' tests the candidate clauses and returns the active part of arbitrary one of the clauses whose passive parts have been successfully solved. This many-to-one arbitration is realized by the nest of binary clause selection performed in the predicate `resolve'.

It is essential that each candidate clause is melted after it has been brought into a guard. If it were melted before passed into the guard, all variables in it would be protected against instantiation from the guard.

5. IMPORTANT FEATURES OF GHC

5.1 Simplicity

GHC has only a small number of primitive operations all of which are considered small:

(1) calling a predicate leaving all its arguments unspecified--i.e., after making sure only that they are new distinct variables,
(2) unifying a variable with another variable or a non-variable term whose arguments are all distinct variables, and
(3) trust.

Furthermore, the semantics of guard and trust is powerful enough to express the following notions:

(1) conditional branching,
(2) nondeterministic choice, and
(3) synchronization.

5.2 Descriptive Power

We have succeeded in rewriting most of the Concurrent Prolog programs we have. In particular, we have written a GHC program which performs bounded buffer communication, and a meta-interpreter of GHC itself (see Chapter 4).

5.3 Efficiency

It cannot be immediately concluded that GHC can be efficiently implemented on parallel computers. The efficiency of GHC owes very much to the future research on the language itself and its implementation.

However, GHC is more favorable than Concurrent Prolog for implementation: It needs no mechanism for multiple environments; it provides more information on synchronization statically.

6. POSSIBLE EXTENSIONS

This chapter suggests some possible extensions. The extensions shown below are currently not part of GHC. Their necessity, implementability, compatibility with other language features, and so on are yet to be examined before they are actually introduced.

6.1. Otherwise

The predicate `otherwise' proposed in [Shapiro and Takeuchi 83] can be introduced to express `negation as failure'. The predicate `otherwise' can appear only as a guard goal. A goal `otherwise' succeeds when the passive part of all the other candidate clauses for the given goal have failed; until then it suspends.

This predicate could be conveniently used for describing a `default' clause.

6.2. Metacall Facilities

We sometimes want to see whether a given goal succeeds or fails without making the test itself fail. Consider, for example, a monitor program. A monitor program may create several processes, some of which are user programs and others are service programs. In this case, the user programs must be executed in a fail-safe manner because if one of them should fail, so does the whole system. Furthermore, a monitor program must have some means to abort its subordinate user programs.

Another example is a program tracer. A program tracer must execute a given program, generating trace information every moment. Even if the program fails, the tracer should generate appropriate diagnostic information without failing. The tracer may even have to trace the execution of passive parts, which is really an impure feature since information should be extracted from the place from where no bindings must otherwise be exported.

A partial evaluator is another example. A partial evaluator rewrites a program clause by executing the goals in the clause. For example, the first clause in the following program

```
p(Y) :- q(Y) | ... .
q(Z) :- true | Z=ok.
```

in Section 3.2 can be partially evaluated to the following clause.

```
p(ok) :- true | ... .
```

To do such rewriting, it must be possible to execute a given goal to obtain a finite set of substitutions and, in the case of suspension, a finite set of remaining (suspended) goals. In this case, the initial goal and the result must be represented in a frozen form. For if ordinary variables are used, the solver of the initial goal cannot know when that goal has been fully instantiated, nor can we know when all bindings have been made; the delay of binding is not guaranteed to be bounded.

In the following subsections, we propose two kinds of metacall predicates which are expected to support the above applications: one which solves a given goal possibly by communicating with other goals, and one which solves a given goal without communication.

6.3.1. Metacall with communication

The metacall predicates whose call may communicate with other goals will have the following formats:

```
call(Goal, Write_enabled_term, Result, Interrupt)
program_call(Program, Goal, Write_enabled_term, Result, Interrupt)
```

Both versions solve Goal and if successful, try to unify Result with the constant `success' and otherwise with the constant `failure'. The argument Interrupt is used for aborting, interrupting, or resuming the execution of Goal. The above features are essentially the same as proposed in [Clark and Gregory 84b].

The Write_enabled_term argument is important for communication with other goals: Of the variables in Goal, only those appearing in Write_enabled_term may be instantiated by the metacall predicates. When some unification invoked by the metacall predicates is to instantiate other variables in Goal, it suspends. The `program_call' predicate uses Program (given in compiled form) to solve Goal. The `call' predicate uses the current program given by the nearest parent `program_call' goal or given at the top level.

If we let Write_enabled_term be some ground term or a variable appearing nowhere else, Goal is completely passive. If we let Write_enabled_term be identical to Goal, Goal can freely communicate with other goals running in parallel. A Write_enabled_term is used for specifying the communication streams used while solving Goal.

Note that this metacall facility is impure, as stated before, in that it introduces a new kind of nondeterminism. Consider the following example (idea taken from [Sato and Sakurai 84]).

```
:- call(X=0, X, _, _), X=1.
```

If the first goal is executed first, X becomes 0 since it is write_enabled. Then the unification X=1 fails and so does the whole clause. If the second goal is executed first, X becomes 1. But since the first goal never fails, the whole clause succeeds. This is a new kind of nondeterminism resulting from the order of unification; without this facility, all nondeterminism would result from the arbitrary selection of trustable clauses.

Note also that a program which uses the predicate `otherwise' can be rewritten to the program which uses `call' with ground Write_enabled_term but not `otherwise', and vice versa.

6.3.2. Frozen metacall

The purpose of frozen metacall is to enable its user to know when all bindings have been supplied. For this purpose, we have to let the metacall predicate know where variables appear in the initial goal, and the metacall predicate has to tell where variables appear when the goal has been solved. In such a case, each variable must be indicated as a constant symbol (or some other ground term) which identifies that variable; that is, the initial and the final goals must be represented as frozen terms.

The predicates for frozen metacall may look like the following.

```
frozen_call(Goal, Final_goal, Remaining_goals)
frozen_program_call(Program, Goal, Final_goal, Remaining_goals)
```

Goal is the initial goal to be solved. Final_goal is the final, instantiated goal. This is returned as a different argument since Goal is frozen and hence cannot be instantiated. Remaining_goals is a conjunction of goals which has remained irreducible. If Goal is reduced out, Remaining_goals will be bound to `true'. If Goal turns out to fail, Remaining_goals will be bound to `fail'.

For example, given the following one-clause program,

```
p(X) :- true | X=5, q(X, Y).  % '$VAR(_)' stands for frozen variables
```

the goal

```
:- frozen_call(p('$VAR'(1)), A, B).
```

may bind A to p(5) and B to q(5, '$VAR'(2)).

7. IMPLEMENTATION OUTLINE

The purpose of this chapter is to demonstrate that the suspension mechanism of GHC can be implemented. We will first show an easy-to-understand but possibly inefficient method: pointer coloring. Here we do not consider the suspension of active parts. The active part of a clause is assumed to start after the clause has been trusted.

When a term in a goal and a variable in the passive part of a clause are unified, we color the pointer which indicates the binding. A term dereferenced using one or more colored pointers cannot be instantiated. When the clause is trusted, colored pointers created in its passive part are uncolored. For this purpose, the passive part of a clause must record all pointers colored for that passive part. Uncoloring can be done in parallel with the other operations in the active part.

Care must be taken when the term in a goal to be unified with the variable in the passive part is itself dereferenced using colored pointers. Consider the following example.

```
    :- p(f(A)).                   --- (i)
    p(X) :- q(X) | ... .          --- (ii)
    q(Y) :- true | Y=f(b).        --- (iii)
```

If the variable Y should directly point to the term f(A) by a colored pointer and uncolor it upon trust of Clause (iii), the variable A would be erroneously instantiated to the constant `b'. There are a couple of ways to get around:

(1) Disallow pointers which go directly out of nested guards.
(2) Let each pointer know how many levels of guards it goes through.
(3) [Miyazaki 85] Allow pointers to go directly through nested guards. However, let each colored pointer know for what passive part it is colored. When directly pointing a term dereferenced using colored pointers, that new pointer must be recorded in the passive part which records the last colored pointer in the dereferencing chain.

The pointer-coloring method explained above is general. In many cases, however, we can analyze suspension statically. The simplest case is the following clause.

```
    p(true) :- ... | ... .
```

The head argument claims that when `p' is called, its argument must have been instantiated to `true' for this clause to be trusted. We can statically generate the code for this check, so we need not use colored pointers in this case.

In general, if guard goals consist only of system predicates for simple checking (e.g., integer comparison), compile-time analysis is easy because no consideration is needed on other clauses. On the other hand, if a user-defined predicate is called in some guard, global analysis is necessary to determine which unification may suspend and which unification cannot. There may be no general method for static analysis. However, in many useful cases, static analysis like PARLOG's compile-time mode analysis [Clark and Gregory 84c] will be effective.

8. COMPARISON WITH OTHER LANGUAGES

8.1. Comparison with Concurrent Prolog and PARLOG

GHC is like Concurrent Prolog and PARLOG in that it is a parallel logic programming language which supports committed-choice nondeterminism and stream communication. However, GHC is simpler than both Concurrent Prolog and PARLOG.

Firstly, unlike Concurrent Prolog, GHC has no read-only annotations. A read-only annotation is not very suited for fully parallel execution of a program [Ueda 85]. In GHC, the use of passive parts enables process synchronization.

Secondly, Concurrent Prolog needs a multiple environment mechanism while GHC does not. In Concurrent Prolog, bindings generated in each guard must be recorded in the local environment until trust. These bindings must be exported into the global environment upon trust, but this exportation cannot be done sequentially. Furthermore, we have to do multiple waits for suspended variables. In GHC, we are free of all these burdens. More importantly, we have never obtained any evidence that we need multiple environments in stream-AND-parallel programming.

There is another problem with multiple environments: We have to define precisely when inconsistency (or ununifiability) between the local and the global information must be detected if any, which we found is very difficult [Ueda 85].

Thirdly, unlike PARLOG, we require no mode declaration for each predicate. PARLOG's mode declaration is nothing but a guide for translating PARLOG program into Kernel PARLOG [Clark and Gregory 84c]. Therefore, we can do without modes. In fact, GHC is more similar to Kernel PARLOG than to PARLOG. However, unlike Kernel PARLOG, we have only one kind of unification. Although each unification operation occurring in a GHC program might be compiled into one of several specialized unification procedures, GHC itself needs (and has) only one.

Another difference from (Kernel) PARLOG is that a (Kernel) PARLOG program requires compile-time analysis in order to guarantee that it is legal, i.e., it contains no unsafe guard which may bind variables in the caller of the guard [Clark and Gregory 84c]. On the other hand, a GHC program is legal if and only if it is syntactically legal; it can be executed without any semantic analysis.

8.2. Comparison with Qute [Sato and Sakurai 84]

Qute is a functional language based on unification. Qute allows parallel evaluation which corresponds to AND-parallelism in logic programming languages, but the result obtained is guaranteed to be the same irrespective of the particular order of evaluation. That is, there is no observable nondeterminism.

Although Qute and GHC are independently developed and look differently at a glance, their suspension mechanisms are essentially the same. The Qute counterpart of GHC's guard is the condition part of the if-then-else construct, from where no bindings can be exported.

The major difference between Qute and GHC is that Qute has no committed-choice nondeterminism while GHC has one. Qute does not have committed-choice nondeterminism (though [Sato and Sakurai 84] suggests it could) because it pursues the Church-Rosser property of the evaluation algorithm. GHC has one because our applications include a system which interfaces with the real world (e.g., peripheral devices).

Another difference is that Qute has sequential AND while GHC does not. We deliberately excluded sequential AND, because our programming experience with Concurrent Prolog has never called for this construct. Sequential AND could be used for the specification of scheduling and for synchronization. However, the primitives for scheduling should be considered at a different level from that of GHC, and sequential AND as a synchronization primitive has proved to be hard to define so that it may fit well in the computation model of GHC.

8.3. Comparison with CSP [Hoare 78]

GHC is similar to CSP (communicating sequential processes) in the following points.

(1) Both encourage programming based on the concept of communicating processes.
(2) The guard mechanism plays an important role for conditional branching, nondeterminism and synchronization.
(3) Both pursue simplicity.

The major difference is that CSP tries to rule out any dynamic constructs --dynamic process creation, dynamic memory allocation, recursive call, etc.-- while GHC does not. Another major difference is that CSP has the concept of sequential processes while GHC does not. To put them differently, CSP is at the level nearer to the current computer architecture.

8.4. Comparison with (sequential) Prolog

Comparison with sequential Prolog must be made from the viewpoint of logic programming languages, not of parallel programming languages.

First of all, GHC has no concepts of the order of clauses or the order of goals in a clause. GHC is undoubtedly nearer to the Horn clause logic in this point. The semantics of Prolog must explain its sequentiality; without it, we cannot discuss some properties of a program such as termination.

GHC deviates from first-order logic in that it introduces the guard construct. It will be hard to give the semantics of the guard within the framework of first-order logic. However, Prolog also suffers from the same situation because of the notorious but useful cut operator. The trust operator of GHC is the parallel of the cut operator. However, since the trust operator has been introduced in a more disciplined way, it should be easier to give a formal semantics to it.

One problem with Prolog is that the use of 'read' and 'write' predicates prevents the declarative reading of a program. In GHC, we no longer need imperative predicates because the concept of streams can well be adapted to input and output.

8.5. Comparison with Delta Prolog [Pereira and Nasr 84]

Delta-Prolog is an extension of Prolog which allows multiple processes. Communication and synchronization are realized using the notion of 'event'. The underlying logic which explains the meaning of events is called Distributed Logic.

One of the differences between Delta-Prolog and GHC is that Delta-Prolog retains the sequentiality concept and the cut operator of Prolog. Both of them seemed to be a peculiarity of Prolog, so GHC did not stick to them. A parallel program in Delta-Prolog may look quite different from the comparable sequential programs in Delta-Prolog itself and in Prolog. On the other hand, a class of GHC programs which have only unidirectional information flow (like pipelining) is easily rewritable to Prolog by replacing trust operators by cuts, and a class of Prolog programs which use no deep backtracking and each of whose predicates has only one intended input/output mode is also easily rewritable to GHC.

9. CONCLUSIONS

We have proposed a parallel logic programming language, Guarded Horn Clauses. Its syntax, informal semantics, programming examples, important features, possible extensions, implementation technique of synchronization mechanism, and comparison with other languages have been described.

We hope the simplicity of GHC will make it suitable for a parallel computation model as well as a programming language. The flexibility of GHC makes its efficient implementation difficult compared with CSP. However, a flexible language could be appropriately restricted in order to make simple programs run efficiently. On the other hand, it would be very difficult to extend a fast but inflexible language naturally.

ACKNOWLEDGMENTS

The author would like to thank Akikazu Takeuchi, Toshihiko Miyazaki, Jiro Tanaka, Koichi Furukawa, Rikio Onai and other ICOT members, as well as the members of ICOT Working Groups, for useful discussions on GHC and its implementation. Thanks are also due to Ehud Shapiro, Steve Gregory, and Anthony Kusalik for their comments on the earlier versions of this paper. Katsuya Hakozaki, Masahiro Yamamoto and Kazuhiro Fuchi provided very stimulating research environments.

This research was done as part of the R&D activities of the Fifth Generation Computer Systems Project of Japan.

REFERENCES

[Bowen et al. 83] 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 (1982).

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

[Clark and Gregory 84b] Clark, K.L. and Gregory, S., Notes on Systems Programming in PARLOG, Proc. Int. Conf. on Fifth Generation Computer Systems 1984, Institute for New Generation Computer Technology, pp.299-306 (1984).

[Clark and Gregory 84c] Clark, K.L. and Gregory, S., Notes on the Implementation of PARLOG, Research Report DOC 84/16, Dept. of Computing, Imperial College, London, 1984.

[Hoare 78] Hoare, C.A.R., Communicating Sequential Processes, Comm. ACM, Vol. 21, No.8, pp.666-677 (1978).

[Kowalski 74] Kowalski, R., Predicate Logic as Programming Language, Proc., IFIP 74, North-Holland, pp.569-574 (1974).

[Miyazaki 85] Miyazaki, T., unpublished manuscript, Institute for New Generation Computer Technology (1985).

[Nakashima et al. 84] Nakashima, H., Ueda, K. and Tomura, S., What Is a Variable in Prolog? Proc. Int. Conf. on Fifth Generation Computer Systems 1984, Institute for New Generation Computer Technology, pp.327-332 (1984).

[Pereira and Nasr 84] Pereira, L.M. and Nasr, R., Delta-Prolog: A Distributed Logic Programming Language, Proc. Int. Conf. on Fifth Generation Computer Systems 1984, Institute for New Generation Computer Technology, pp.283-291 (1984).

[Roussel 75] Roussel, P., Prolog: Manual de Reference et d'Utilisation, Groupe d'Intelligence Artificielle, Marseille-Luminy (1975).

[Sato and Sakurai 84] Sato, M. and Sakurai, T., Qute: A Functional Language Based on Unification, Proc. Int. Conf. on Fifth Generation Computer Systems 1984, Institute for New Generation Computer Technology, pp.157-165 (1984).

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

[Shapiro 84] Shapiro E., Systems Programming in Concurrent Prolog, Conf. Record of the 11th Annual ACM Symp. on Principles of Programming Languages, pp.93-105 (1984).

[Shapiro and Takeuchi 83] Shapiro, E.Y. and Takeuchi, A., Object Oriented Programming in Concurrent Prolog, New Generation computing, Vol.1, No.1, pp. 25-48 (1983).

[Takeuchi and Furukawa 83] Takeuchi, A. and Furukawa, K, Interprocess Communication in Concurrent Prolog, Proc. Logic Programming Workshop '83, Universidade nova de Lisboa (1983).

[Ueda and Chikayama 84] Ueda, K. and Chikayama T., Practical Implementation of a Parallel Logic Programming Language, First Conference of Japan Society of Software Science and Technology, pp.307-310 (1984) (in Japanese).

[Ueda and Chikayama 85] Ueda, K. and Chikayama T., Concurrent Prolog Compiler on Top of Prolog, to appear as 1985 Symposium on Logic Programming (1985).

[Ueda 85] Ueda, K., Concurrent Prolog Re-Examined, to appear as ICOT Tech. Report TR-102 (1985).

[Warren et al. 77] Warren, D.H.D., Pereira, L.M. and Pereira, F., PROLOG--The Language and Its Implementation Compared with Lisp, Sigplan Notices, Vol. 12, No.8, pp.109-115 (1977).