ICOT Technical Report TR-103

GUARDED HORN CLAUSES

Kazunori Ueda

C&C Systems Research Laboratories, NEC Corporation*
1-1, Miyazaki 4-chome, Miyamae-ku, Kawasaki, Kanagawa 213 Japan

June 1985
Revised: December 1985**
Modified: November 1986***

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

Kowalski [1974] showed that a Horn clause is amenable to procedural interpretation. Prolog was developed as a sequential programming language based on the procedural interpretation of Horn clauses [Roussel 1975], and it has proved to be a simple, powerful, and efficient sequential programming language [Warren, Pereira and Pereira 1977].

As Kowalski 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 capable of describing important concepts such as communication and synchronization. We need some additional mechanism to express these concepts, and this paper shows that this can be effected with only one construct, the *guard*.

We introduce guarded Horn clauses in the following sections. Guarded Horn Clauses (GHC) will be used as the name of our language. We compare GHC with other logic/parallel programming languages. GHC is intended to be the machine-independent core of the Kernel Language for ICOT's Parallel Inference Machine.

---

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

** Major revision was made to Section 6 and slight revision to the other sections. This version appeared 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.

*** Slightly reworded. This version appeared in *Concurrent Prolog: Collected Papers*, Shapiro, E. Y. (ed.), The MIT Press, Cambridge, 1987, pp. 140–156.

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 among primitive operations as possible in order to preserve parallelism inherent in a Horn clause program. This would 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 the foundations of parallel programming with 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 comparable programs written in existing parallel programming languages.

Concurrent Prolog [Shapiro 1983] and PARLOG [Clark and Gregory 1984a] seem to lie near the solution. Both realize processes by goals and communication by streams implemented as lists. Synchronization is realized by read-only variables in Concurrent Prolog and by one-way unification in PARLOG.

GHC inherits the *guard* construct and the programming paradigm established by these languages. The most characteristic feature of GHC is that the guard is the only syntactic construct added to Horn clauses. Synchronization in GHC is realized by the semantic rules of guards.

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 GHC-to-Prolog compilers in Prolog by modifying different versions of Concurrent Prolog compilers on top of Prolog [Ueda and Chikayama 1985].

3. SYNTAX AND SEMANTICS

3.1. Syntax

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

$$H \text{ :- } G_1, \ldots, G_m \mid B_1, \ldots, B_n. \quad (m \geq 0, n \geq 0).$$

where H, G_i's, and B_i's are atomic formulas that are defined as usual. H is called a clause head, G_i's are called guard goals, and B_i's are called body goals. The operator ' | ' is called a commitment operator. The part of a clause before ' | ' is called the guard, and the part after ' | ' is called the body. Note that *the clause head is included in the guard.* The 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 is read as "H is implied by G_1, ..., and G_m and B_1, ..., and B_n".

A goal clause has the following form:

$$:\!- B_1, \ldots, B_n. \quad (n \geq 0).$$

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.

In this paper, we use symbols beginning with uppercase letters for variables and symbols beginning with lowercase letters for function and predicate symbols, following DECsystem-10 Prolog [Bowen et al. 1983]. 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 an 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:

- *Rules of Suspension*

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

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

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

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

  until that clause is selected for commitment (see below).

  A piece of unification which can succeed only by making such bindings is suspended until it can succeed without making such bindings *(end of the rules of suspension).*

Note that a set of variables whose instantiation is inhibited by the above rules can vary as computation proceeds. When a variable `X` in the set S is bound to a non-variable term `T` (in a way not disallowed above), we include all the variables in `T` in S and remove `X` itself from S.

Another rule we have to add is the *commitment* rule. When some clause succeeds in solving (see below) its guard for a given goal, that clause tries to be selected exclusively for subsequent execution of the goal. To be selected, it must first confirm that no other clauses belonging to the same procedure have been selected for the same goal. If confirmed, that clause is selected indivisibly; we say that the goal is committed to that clause and also that that clause is selected for commitment.

We say that a set of goals *succeeds* (or is *solved*) if it is reduced to an empty set of goals by using a selected clause for each initial or intermediate goal: We are interested in a reduction path in which only selected clauses are involved. The notion of failure is not introduced here, but it will be discussed in Section 6.1.

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; head unification involved in resolution can be done in parallel; head unification and the execution of guard goals can be done in parallel. However, what is even more important is that we can also execute a set of tasks in a predetermined order as long as this does not change the meaning of the program.

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

(a) The guard of a clause cannot export any bindings to (or, make any bindings observable from) the caller of that clause, and

(b) the body of a clause cannot export any bindings to (or, make any bindings observable from) the guard of that clause before commitment.

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 clause not yet selected for commitment. 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 performed in a guard (including a head) and which would export bindings is recorded locally. In GHC, such unification simply suspends. Suspension of unification due to some guard may be released when some goal running in parallel with the goal for which the guard is being executed has instantiated the variable that caused suspension.

An example may be helpful in 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)
```

The predicate '=' is a predefined predicate which unifies its two arguments. This predicate must be considered as predefined, because it cannot be defined in the language.

Clause (ii) cannot instantiate the argument `X` of its caller to the constant '`ok`', since this unification is executed in the guard. 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 selected for commitment, and this clause can be selected almost immediately. Therefore, no matter which of the two goals of Clause (i) starts first, the head unification of Clause (ii) can succeed only after the '`Z=ok`' goal in Clause (iii) is executed.

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 guard of Clause (ii′), we have to do two things in parallel: unify `X` and `Y` (i.e., parameter passing), and solve `q(Y)`. Let us assume that parameter passing occurs first. Then the goal `q(Y)` tries to unify `Y` (now identical to `X`) with '`ok`'. However, this unification cannot instantiate `X` because it is indirectly invoked by the guard 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 a binding to the caller of Clause (ii′), namely `p(X)`. However, this binding causes the subsequent parameter passing to suspend because it would export a binding. Hence, no matter which case actually arises, Clause (ii′) behaves exactly like Clause (ii) with respect to bindings given to the variable `X`.

Some important consequences of the above rules follow:

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

(2) 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.

(3) The unification of head arguments and the execution of guard goals can be executed in parallel. That is, the execution of guard goals can start before the unification of head arguments has completed. However, the usual execution method that solves guard goals only after head unification is also allowed; it does not change the meaning of a program.

(4) The execution of the body of a clause may, but need not, start before that clause is selected. Bindings made by the body are unobservable from the guard before commitment, so the meaning of the program is independent of whether the body starts before or only after commitment.

(5) We need not implement a multiple environment mechanism, i.e., a mechanism for binding a 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 selected clause, can export bindings, thus eliminating the need of a multiple environment mechanism.

Unfortunately, properties (2) and (3) do not hold if we introduce the concept of failure. For example, the following goal

```
Goal:                       :- and(X, false).
Clause:  and(true, true) :- true | true.
```

fails if the arguments are unified in parallel, but suspends if they are unified from left to right [Gregory 1985].

4. PROGRAM EXAMPLES

4.1. Binary Merge

```
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.
```

The goal '`merge(Xs, Ys, Zs)`' merges two streams `Xs` and `Ys` (implemented as lists) into one stream `Zs`. This is an example of a nondeterministic program. 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` are expected to appear on `Zs` almost in the order of arrival.

Note that no bindings can be exported from the guards; bindings to `Zs` must be made within the bodies. This programming style, however, serves to clarify causality. 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,Ps) :- true | gen(2,Max,Ns), sift(Ns,Ps).

gen(N,Max,Ns) :- N=<Max | Ns=[N|Ns1], N1:=N+1, gen(N1,Max,Ns1).
gen(N,Max,Ns) :- N >Max | Ns=[].

sift([P|Xs],Zs) :- true | Zs=[P|Zs1], filter(P,Xs,Ys), sift(Ys,Zs1).
sift([],    Zs) :- true | Zs=[].

filter(P,[X|Xs],Ys) :- X mod P=:=0 |             filter(P,Xs,Ys).
filter(P,[X|Xs],Ys) :- X mod P=\=0 | Ys=[X|Ys1], filter(P,Xs,Ys1).
filter(P,[],    Ys) :- true        | Ys=[].
```

The call '`primes(Max, Ps)`' returns through `Ps` 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, Xs, Ys)`' is generated which filters out the multiples of `P` from the stream `Xs`, yielding `Ys`.

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 '`=:=`'.

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

4.3. Bounded Buffer Stream Communication

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

buffer(N,Hs,Ts) :- N > 0 | Hs=[_|Hs1], N1:=N-1, buffer(N1,Hs1,Ts).
buffer(N,Hs,Ts) :- N=:=0 | Ts=Hs.

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

consume([H|Hs],Ts) :- H\='EOS' | Ts=[_|Ts1], consume(Hs,Ts1).
consume([H|Hs],Ts) :- H ='EOS' | Ts=[].
```

This program illustrates the general statement that demand-driven computation can be implemented by means of data-driven computation. It uses the

bounded-buffer concept first introduced by Takeuchi and Furukawa [1983] in a logic programming framework. 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 '`Ts=[_|Ts1]`' every time it has confirmed the top element `H` of the stream. The top and the tail of the stream are initially related by the goal '`buffer(N, Hs, Ts)`'.

The binary predicate '`\=`' is the negation of the predicate '`=`'. It succeeds when its two arguments are proved to be ununifiable; it suspends until then.

4.4. Meta-Interpreter of GHC

```
call(true ) :- true | true.
call((A,B)) :- true | call(A), call(B).
call(A=B  ) :- true | A=B.
call(A    ) :- A\=true, A\=(_,_), 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 basically a GHC version of the Concurrent Prolog meta-interpreter by Shapiro [1984]. The predicate '`clauses`' is a system predicate which returns in a *frozen* form [Nakashima, Ueda and Tomura 1984] a list of all clauses whose heads are potentially unifiable with the given goal. 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 body of an arbitrary clause whose guard has been successfully solved. This many-to-one arbitration is realized by the combination of binary clause selection performed in the predicate '`resolve`'.

It is essential that each candidate clause is melted after it has been brought into the guard of the first clause of '`resolve`'. 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 with a non-variable term whose arguments are all new distinct variables, and

(3) commitment.

Operation (1) is effectively resolution without unification. From the viewpoint of parallel execution, resolution in the original sense [Robinson 1965] need not be considered as an indivisible operation. Resolution can be decomposed into goal rewriting and unification, and the latter can be executed in parallel with the newly created goals, as stated in Section 3.2.

Operation (2) shows that the unification of a variable and a non-variable term is not necessarily a primitive operation. For example, the unification '`X=f(a)`' can be decomposed into the two operations '`X=f(Y)`' and '`Y=a`', where `Y` is a new variable. This was also suggested by Hagiya [1983].

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

(1) conditional branching,

(2) nondeterministic choice, and

(3) synchronization.

This feature is much like CSP [Hoare 1978], but CSP provides additional constructs '`?`' (input command) and '`!`' (output command) for synchronization. The Relational Language [Clark and Gregory 1981] was the first to introduce the guard concept to logic programming for reasons similar to ours****. However, GHC has removed the restrictions on the guard of the Relational Language together with mode declarations and annotations.

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 (Section 4.3), and a meta-interpreter of GHC itself (Section 4.4).

**** IC-Prolog [Clark and McCabe 1980] was the first to introduce the guard concept to logic programming, but for rather different reasons.

5.3. Efficiency

It cannot be immediately concluded that GHC can be efficiently implemented on parallel computers. The efficiency of GHC will depend very much on future research on the language itself and its implementation. However, GHC is more amenable than Concurrent Prolog to efficient implementation: It needs no mechanism for multiple environments; and it provides more information on synchronization statically. We made a compiler of GHC subset which compiles a GHC program into Prolog [Ueda and Chikayama 1985], and an '`append`' program ran at more than 12kLIPS on DEC2065. The current restriction is that user-defined goals are not allowed in guards. Another GHC-to-Prolog compiler was made by Miyazaki. Although less efficient than ours, his compiler is capable of handling nested guards.

For applications in which efficiency is the primary issue but little flexibility is needed, we could design a restricted version of GHC which allows only a subclass of GHC and/or introduces declarations which help optimization. Such a variant should have the properties that additional constructs such as declarations are used only for efficiency purposes and that a program in that variant is readable as a GHC program once the additional constructs are removed from the source text.

6. POSSIBLE EXTENSIONS

This section suggests some possible extensions, which are currently not part of GHC. Issues such as their necessity, implementability, and compatibility with other language features should be carefully examined before they are actually introduced.

6.1. Finite Failure and the Predicate '`otherwise`'

The semantics of GHC as described in Section 3.2 does not include the concept of failure. However, failure of unification can be readily introduced into the language. We can say that a set of goals fails if it contains or derives some unification goal and its two arguments are instantiated to different principal functors. Then, in general, a suspended unification may turn out later either to fail or to succeed.

Another kind of failure is caused by a goal for which there prove to be no selectable clauses. Calling a non-existent predicate also falls under this category. This kind of failure must be detected as failure only under the *closed world assumption*; otherwise, that goal would have to suspend until somebody adds a selectable clause to the program.

The predicate '`otherwise`' proposed by Shapiro and Takeuchi [1983] can be introduced to express 'negation as failure'. The predicate '`otherwise`' can appear

only as a guard goal. A goal '`otherwise`' succeeds when the guards of all the other candidate clauses for a 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 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.

Let us consider a program tracer next. 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 guards, 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 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 were used, the solver of the initial goal could not know when that goal had been fully instantiated, nor could we know when all bindings had been made. The binding delay is not guaranteed to be bounded.

We are considering language facilities which support all of these applications. However, we have not reached a satisfactory solution yet. The metacall facility proposed by Clark and Gregory [1984b] was a candidate solution, but it proved to have some semantical problems. Their two-argument metacall '`call(Goal, Result)`' tries to solve `Goal` possibly generating output bindings, and it unifies `Result` with '`succeeded`' upon success and with '`failed`' upon failure. However, consider the following example [Sato and Sakurai 1984]:

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

If the first goal is executed first, `X` becomes 0. 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 choice of selectable clauses.

Let us consider another example:

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

The semantics of a GHC variable is intended to allow the above goal to be rewritten as follows [Ueda 1985],

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

because they are logically equivalent. However, this rewriting shows that the failure of unification cannot be confined in either '`call`'. The failure can creep out and topple the whole goal. This means that the metacall facility as proposed by Clark and Gregory cannot protect a system program from unpredictable behavior by a user program. Further investigation is necessary to find a better solution.

7. IMPLEMENTATION OUTLINE

The purpose of this section 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 bodies. The body of a clause is assumed to start after the clause has been selected.

When a term in a goal and a variable in the guard 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 selected, colored pointers created in its guard are uncolored. For this purpose, the guard of a clause must record all pointers colored for that guard. Uncoloring can be done in parallel with the other operations in the body.

Care must be taken when the term in a goal to be unified with the variable in the guard 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 selection of Clause (iii), the variable `A` would be erroneously instantiated to the constant '`b`'. There are a couple of possible remedies:

(1) Disallow a pointer which goes directly out of nested guards and use a chain of pointers instead.

(2) Let each pointer know how many levels of guards it goes through.

(3) [Miyazaki 1985] Allow a pointer to go directly through nested guards. However, let each colored pointer know for what guard it is colored. When directly pointing a term dereferenced using colored pointers, that new pointer must be recorded in the guard 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 the corresponding goal argument must have been instantiated to '`true`' for this clause to be selected. We can statically generate the code for this check, and need not use colored pointers in this case.

In general, if a guard calls only 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 it calls a user-defined predicate, global analysis is necessary to determine which unification may suspend and which unification cannot. There will be no general method for static analysis, but in many useful cases, static analysis like PARLOG's compile-time mode analysis [Clark and Gregory 1984c] 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. In GHC, the semantics of guards enables process synchronization.

Secondly, Concurrent Prolog needs a multiple environment mechanism while GHC and PARLOG do not. In Concurrent Prolog, bindings generated in each guard are recorded locally until commitment and are exported into the global environment upon commitment. However, this mechanism contains semantical problems whose solution would require an additional set of language rules, as Ueda [1985] pointed out. More importantly, we have not obtained any evidence that we need multiple environments in stream-AND-parallel programming.

Thirdly, unlike PARLOG, we require no mode declaration for each predicate. PARLOG's mode declaration is nothing but a guide for translating PARLOG programs into Kernel PARLOG [Clark and Gregory 1984c], so we can do without modes. In fact, GHC is closer 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 1984c]. 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

Qute [Sato and Sakurai 1984] is a functional language based on unification. Qute allows parallel evaluation which corresponds to AND-parallelism in logic programming languages, but the result of evaluation is guaranteed to be the same irrespective of the particular order of evaluation. That is, there is no observable nondeterminism.

Although Qute and GHC were developed independently and may look different, 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 which 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 [1984] suggest 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. One may think that sequential AND could be used for the specification of scheduling and for synchronization. However, the primitives for scheduling should be introduced at a different level from that of GHC, and sequential AND as a synchronization primitive is of no use in the intended computation model of GHC which allows delay for communication by shared variables.

8.3. Comparison with CSP

GHC is similar to CSP (Communicating Sequential Processes)[Hoare 1978] 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 a concept of sequential processes while GHC does not. CSP is at a level nearer to the current computer architecture. GHC is more abstract and has a smaller set of primitives: it uses unification instead of input, output, and assignment commands, and it uses a recursive call instead of a repetitive command.

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 Horn clause logic on 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 a semantics to the guard within the framework of first-order logic. However, Prolog also suffers from the same problem because of the notorious, but useful, cut operator. The commitment operator of GHC corresponds to the cut operator. However, since the commitment operator has been introduced in a more controlled 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 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. Large data structures such as mutable arrays and databases can also be logically and efficiently handled using transaction streams as the interface [Ueda and Chikayama 1984].

8.5.Comparison with Delta Prolog

Delta-Prolog [Pereira and Nasr 1984] is an extension of Prolog which allows multiple processes. Communication and synchronization are realized using the notion of an *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 seem to be peculiarities of Prolog, so GHC avoided them. A parallel program in Delta-Prolog may look quite different from 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 commitment 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 described the 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 were outlined and discussed.

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-like languages. 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 ICOT Working Groups, for useful discussions on GHC and its implementation. Thanks are also due to Ehud Shapiro, Steve Gregory, Anthony Kusalik and Vijay Saraswat 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, D. L. (ed.), Byrd, L., Pereira, F. C. N., Pereira, L. M. and Warren, D. H. D. [1983] DECsystem-10 Prolog User's Manual. Dept. of Artificial Intelligence, Univ. of Edinburgh.

Clark, K. L. and Gregory, S. [1981] A Relational Language for Parallel Programming. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, ACM, pp. 171–178.

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

Clark, K. L. and Gregory, S. [1984b] Notes on Systems Programming in PARLOG. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, Institute for New Generation Computer Technology, Tokyo, pp. 299–306.

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

Clark, K. L. and McCabe, F. [1980] IC-PROLOG—Language Features. In *Proc. Logic Programming Workshop*, Tärnlund, S.-Å. (ed.), Debrecen, Hungary.

Gregory, S. [1985] *Private communication.*

Hagiya, M. [1983] On Lazy Unification and Infinite Trees. In *Proc. Logic Programming Conference '83*, Institute for New Generation Computer Technology, Tokyo (in Japanese).

Hoare, C. A. R. [1978] Communicating Sequential Processes. *Comm. ACM*, Vol. 21, No. 8, pp. 666–677.

Kowalski, R. [1974] Predicate Logic as Programming Language. In *Proc. IFIP 74*, North-Holland, Amsterdam New York Oxford, pp. 569–574.

Miyazaki, T. [1985] Unpublished manuscript. Institute for New Generation Computer Technology, Tokyo.

Nakashima, H., Ueda, K. and Tomura, S. [1984] What Is a Variable in Prolog? In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, Institute for New Generation Computer Technology, Tokyo, pp. 327–332.

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

Robinson, J. A. [1965] A Machine-Oriented Logic Based on Resolution Principle. *J. ACM*, Vol. 12, No. 1, pp. 23–41.

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

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

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

Shapiro, E. Y. [1984] Systems Programming in Concurrent Prolog. In *Conf. Record of the 11th Annual ACM Symp. on Principles of Programming Languages*, ACM, pp. 93–105.

Shapiro, E. Y. and Takeuchi, A. [1983] Object Oriented Programming in Concurrent Prolog. *New Generation Computing*, Vol. 1, No. 1, pp. 25–48.

Takeuchi, A. and Furukawa, K. [1983] Interprocess Communication in Concurrent Prolog. In *Proc. Logic Programming Workshop '83*, Universidade Nova de Lisboa, Portugal.

Ueda, K. and Chikayama, T. [1984] Efficient Stream/Array Processing in Logic Programming Languages. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, Institute for New Generation Computer Technology, Tokyo, pp. 317–326.

Ueda, K. and Chikayama, T. [1985] Concurrent Prolog Compiler on Top of Prolog. In *Proc. 1985 Symposium on Logic Programming*, IEEE Computer Society Press, pp. 119–126.

Ueda, K. [1985] Concurrent Prolog Re-Examined. ICOT Tech Report TR-102, Institute for New Generation Computer Technology, Tokyo.

Warren, D. H. D., Pereira, L. M. and Pereira, F. [1977] PROLOG—The Language and Its Implementation Compared with Lisp. *Sigplan Notices*, Vol. 12, No. 8, pp. 109–115.