ICOT Technical Report TR-209

Introduction to Guarded Horn Clauses

By Kazunori UEDA*

October 1986
Last modification: November 1986

ABSTRACT This paper informally introduces a programming language Guarded Horn Clauses (GHC) through program examples. GHC is a parallel programming language devised from investigation of the basic framework and practice of logic programming. It has introduced the guard construct with simple semantics into logic programming to express interacting processes. A GHC program naturally expresses parallelism inherent in the original problem. The simple and uniform framework of GHC should be far easier to understand than the constructs of conventional parallel programming languages. We explain GHC in comparison with Prolog, the best-known logic programming language. The readers are assumed to be familiar with programming in Prolog.

KEYWORDS **Parallel programming language, Logic programming, Guard, Unification, Processes, Communication, Synchronization**

1. INTRODUCTION

It is often claimed that parallel programming is difficult. There seem to be two reasons why people think so; one is that we are so inexperienced in grasping and describing parallel activities with interaction. The other reason is that the existing parallel programming languages are generally complex and awkward in describing parallelism. There have been many proposals of language constructs for describing processes, communication and synchronization [1]. However, most of those constructs were designed as additional features for conventional sequential languages. Although the resulting languages might be advantageous for implementation on parallel computers in the near future, we are more interested in designing independently of conventional languages a higher-level language that is suitable for programming parallelism.

Logic programming (or more specifically, programming in a subset of first-order logic called *Horn-clause logic*) is said to be a good framework for parallelism [2]. Execution of a logic program means finding the instances of a given goal that are logical consequences of the program. Since a logic program is a logical

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

formula and expresses nothing on control, the proof procedure can take any strategy in finding those instances. Prolog [3][4], the first and the best-known logic programming language, chose sequential execution, but this choice is inessential unless extralogical features such as side effects are involved. There has been much research on parallel execution of logic programs.

However, parallelism in implementation and parallelism in a programming language are different and even independent. The purpose of the former is to enhance performance, and an implementation may exploit parallelism even from a sequential program for this purpose. Our concern, on the other hand, is to design language constructs in which to describe parallel processes and interaction among them. How to implement them is a separate issue; even without parallel computers, we should support language constructs for parallel programming and their implementation on sequential computers as long as they are useful for natural description of the problem. The original framework of logic programming lacks the concept of control, so it is not as it is suitable for a general parallel programming language, though it may be suitable for a database query language in which control issues are not up to the programmers. We need appropriate notations for expressing control.

Guarded Horn Clauses (GHC) has introduced the concept of guard into the original framework of logic programming to fulfill the above requirements. Introduction of guard into the logic programming framework itself is not an original idea; GHC is unique in that guard is essentially the only additional syntactic construct and that the semantics associated with the guard is quite simple. While the sequentiality of Prolog introduces total orders on the goals in a clause and on the clauses constituting each predicate, the guard mechanism of GHC introduces a partial order on the events of binding variables to terms. The execution mechanism of GHC is best understood in terms of bindings observed and generated by goals.

Programming in GHC is easy at least for simple programs; actually, many simple GHC programs are almost the same as their Prolog counterparts. Furthermore, GHC has the important advantage of allowing declarative reading of a program containing input and output, while Prolog handles them by side effects. On the other hand, GHC is less advantageous than Prolog for programming search problems, since GHC has no automatic backtracking or its parallel counterpart. However, it is quite possible to program search problems in GHC, and furthermore, a pure Prolog program for exhaustive search can be automatically compiled into an *efficient* GHC program [6]. This indicates that GHC is at a lower level than pure Prolog and that pure Prolog can be regarded as a higher-level user language for GHC.

In the subsequent sections, we introduce GHC through program examples. The programs have been tested on the GHC system on top of DEC-10 Prolog

[7]. We omit the systematic description of GHC which can be found in other documents: [8] is the original, [9] is the most detailed, and [10] is the latest and the most rigorous. Comparison between GHC and other parallel/logic programming languages is made in [8] and [9]. The readers are assumed to be familiar with Prolog and its standard syntax first adopted in DEC-10 Prolog [5]. GHC follows that syntax since it is well established as the de facto standard of the logic programming community.

2. A SIMPLE EXAMPLE—CONCATENATING LISTS

Let us begin with a program for concatenating two lists:

```
concat([A|X1],Y,Z) :- true | Z=[A|Z1], concat(X1,Y,Z1).     (2–1)
concat([],    Y,Z) :- true | Z=Y.                           (2–2)
```

This program concatenates two input lists given to the first and the second arguments and returns the result through the third argument. Its Prolog counterpart should look like the following:

```
concat([A|X1],Y,[A|Z1]) :- concat(X1,Y,Z1).                 (2–3)
concat([],    Y,Y     ).                                    (2–4)
```

The major syntactic differences are that each clause of the GHC program has a commitment operator '`|`' and that unification for constructing the result is specified in the right-hand side of '`|`' using the predicate '`=`'.

The commitment operator denotes conjunction in declarative reading. The left-hand side of the commitment operator is called a guard, and the right-hand side is called a body. Note that the guard includes the head of the clause.

We have used two predefined predicates. The goal `true` is used for denoting an empty set of goals and always succeeds immediately. The goal t_1=t_2 denotes equality and is used for unifying two terms. Now it should be clear that the above two programs are equivalent as logical formulae.

In GHC, the order of program clauses and the order of goals in a clause are both insignificant, except that we cannot move a goal across the commitment operator. Unlike in Prolog, conjunctive goals are solved in (pseudo-) parallel, and candidate clauses for a goal are tried in (pseudo-) parallel. However, these activities must obey the following two semantic rules associated with the new syntactic constructs. Firstly, the guard of a clause must be solved passively, that is, without instantiating its caller. We will explain this in more detail below.

To solve the guard of a clause C, we must unify the head of C with the goal G that called C and solve the guard goals in C. This may generally instantiate G; that is, it may bind a variable in G with another variable in G or a non-variable term. However, the above rule states that the guard of C must be solved

without instantiating G. Instantiation of G would be caused by some unification invoked directly or indirectly in the guard, and such unification is suspended until G is sufficiently instantiated (by some other goal running in conjunction) to let it succeed without instantiating G. Consider the following goal clauses:

```
:- concat([1,2,3],[4,5],W).          (2–5)
:- concat([1,2,3],V,    W).          (2–6)
:- concat(U,      [4,5],W).          (2–7)
```

The guard of Clause (2–1) succeeds for the goal `concat([1,2,3], [4,5], W)`; that is, the clause head `concat([A|X1], Y, Z)` can be unified with the goal without instantiating the goal. Likewise, the guard of Clause (2–1) succeeds for `concat([1,2,3], V, W)`. However, it does not succeed for `concat(U, [4,5], W)` because it would instantiate the variable `U` to `[A|X1]`. The guard of Clause (2–2) does not succeed for that goal either. Therefore, Clause (2–7) is suspended forever. Consider the following goal clause next:

```
:- concat(U,[4,5],W), U=[1,2,3].     (2–8)
```

The first goal `concat(U, [4,5], W)` is suspended until the second goal instantiates `U`. However, `concat(U, [4,5], W)` is finally instantiated to `concat([1,2,3], [4,5], W)`, and the guard of Clause (2–1) succeeds for it. Thus, the guard of a clause is used for making the computation suspended until sufficient binding information is supplied by the caller.

The second semantic rule is on the body of a clause: A clause can instantiate its caller by solving its body goals, but only one of the program clauses for each goal is allowed to do so. The *commitment operation* is used for selecting the clause: Each goal commits the subsequent execution of itself to arbitrary one of those clauses whose guards have succeeded. Once committed, the execution never considers other clauses. The body of an *unselected* clause can be solved in principle, but it must not instantiate the caller nor the guard of the clause.

For instance, the execution of the goal `concat([1,2,3], [4,5], W)` performs four commitment operations, three to Clause (2–1) and one to Clause (2–2), for the top-level and the three recursive goals, and instantiates `W` to `[1,2,3,4,5]`.

Any unification for generating bindings back to the caller must be specified in the body of a clause. This is why we need the predicate '`=`'. Consider what happens if we move the unification in the bodies of Clauses (2–1) and (2–2) back to the clause heads:

```
concat([A|X1],Y,[A|Z1]) :- true | concat(X1,Y,Z1).   (2–9)
concat([],    Y,Y     ) :- true | true.               (2–10)
```

Clause (2–9) states that the term of the form $[t_1|t_2]$ must be given to the third argument by the caller. Clause (2–10) states that the second and the third arguments must be made identical by the caller; it never makes them identical by

itself. Hence, the above program can be used for checking if a given list is the concatenation of two other lists, but cannot be used for generating a concatenated list.

To sum up, each program clause waits for input bindings in the guard, and after it is selected for commitment, it generates output bindings in the body. Each clause thus specifies the direction of computation as well as logical contents. For this reason, a GHC program cannot be used in more than one direction; for instance, we cannot use Clauses (2–1) and (2–2) to divide a list into two. They simply suspend the goal `concat(U, V, [1,2,3,4,5])`. Even if Clauses (2–1) and (2–2) are rewritten as follows, they return only one of the six possible solutions that is arbitrarily chosen by the commitment mechanism:

```
concat(X,Y,[A|Z1]) :- true | X=[A|X1], concat(X1,Y,Z1).    (2–11)
concat(X,Y,Z     ) :- true | X=[], Y=Z.                     (2–12)
```

To compute all the possible ways to divide a list into two, we must write a program (say `decompose`) that returns a list of all solutions as a first-order relation. The predicate `decompose` should return a list `[p([], [1,2,3,4,5]), p([1], [2,3,4,5]), ..., p([1,2,3,4,5], [])]` for the input list `[1,2,3,4,5]`. A method for deriving `decompose` from `concat` written in pure Prolog (Clauses (2–3) and (2–4)) is described in [6], but the readers are encouraged to try to define `decompose` by hand for better understanding of exhaustive search.

3. DIFFERENCE LISTS—QUICKSORT

The next example is a `quicksort` program. Here we introduce the concept of a difference list and also we discuss parallelism.

```
quicksort(Xs,Ys) :- true | qsort(Xs,Ys-[]).                  (3–1)

qsort([X|Xs],Ys0-Ys3) :- true |
    part(Xs,X,S,L),
    qsort(S,Ys0-Ys1), Ys1=[X|Ys2], qsort(L,Ys2-Ys3).         (3–2)
qsort([],    Ys0-Ys1) :- true | Ys0=Ys1.                     (3–3)

part([X|Xs],A,S, L0) :- A <  X |
    L0=[X|L1], part(Xs,A,S, L1).                             (3–4)
part([X|Xs],A,S0,L ) :- A >= X |
    S0=[X|S1], part(Xs,A,S1,L ).                             (3–5)
part([],    _,S, L ) :- true   | S=[], L=[].                 (3–6)
```

The goal `quicksort(`u`, `s`)` sorts the list u of integers and return the result to s. This program makes effective use of difference lists. A difference list is a difference $h-t$ of two lists such that t (for tail) appears as a sublist of h (for head). If $h = t$, $h - t$ denotes an empty difference list. When we handle a difference list

$h - t$, we are interested in the part of h before t but not usually in the value of t. Conveniently enough, logic programming enables us to handle a difference list with an undefined tail. Determining the value of t can be left to other goals, and when t is instantiated to a complete list, so is h. Thus the difference list $h - t$ can be considered as a part of a complete list. This technique enables us to exploit parallelism in constructing a complete list.

In the above program, difference lists are used for the results returned by the predicate `qsort`. The top-level clause (3–1) just converts the difference list returned by `qsort` into a complete list by terminating its tail by `[]`. Clause (3–2) is the most important clause in the program: It partitions an input list `[X|Xs]` into a list `S` of small integers and a list `L` of large integers, using `X` as the threshold. The lists `S` and `L` are then sorted recursively, and here we can use parallelism. Since neither of the goals `qsort(S, Ys0-Ys1)` and `qsort(L, Ys2-Ys3)` needs to wait for the data returned from the other, they can run in parallel and generate two difference lists `Ys0-Ys1` and `Ys2-Ys3` simultaneously. The goal `Ys1=[X|Ys2]` relates `Ys1` and `Ys2` by sandwiching `X` between them. It effectively creates a difference list `Ys1-Ys2` with a single element `X`. These three difference lists finally make up a longer difference list `Ys0-Ys3`, the result of Clause (3–2).

The fact that there is no order on the four body goals of Clause (3–2) indicates another source of parallelism. The goal `part(Xs, X, S, L)` can generate `S` and `L` incrementally from their heads, so the two recursive goals can start their jobs before the partition has completed. However, parallelism between `part` and the two `qsort`'s is somewhat different from parallelism between two `qsort`'s. The two `qsort`'s must be suspended if they hit upon the yet undetermined parts of `S` and `L` while inspecting them. This kind of parallelism is called *pipelining*; it is more complex than the parallel execution of independent tasks because we have to care about synchronization.

Clauses (3–4) and (3–5) perform arithmetic comparison in their guards. The goal `A<X` is like its Prolog counterpart, but it never causes an error when `A` or `X` is uninstantiated. Instead, it waits until both `A` and `X` are instantiated and performs comparison. In other words, the goal `A<X` succeeds if and when `A` is *known* to be less than `X`.

Note that one cannot replace the guard goal of Clause (3–5) by `true`. The order of Clauses (3–4) and (3–5) is insignificant, so Clause (3–5) must not assume that Clause (3–4) has failed to be selected. Any condition for commitment must always be specified explicitly.

4. PROCESSES AND STREAMS—GENERATING PRIMES

This section introduces process interpretation of a GHC program and interprocess communication using the concept of streams. The method for output is

also introduced. The example program is the generator of a sequence of prime numbers. This is the first complete program in this paper in that it specifies output operations explicitly. Most implementations of logic programming languages display binding information after solving each top-level goal, but we cannot rely on these facilities when we must have full control over what to output and how.

We first show the top-level predicate:

```
go(Max) :- true |
    primes(Max,Ps), outterms(Ps,Os), outstream(Os).          (4–1)
```

The predicate `go` receives an integer `Max`, and generates and prints all the primes up to `Max`. The goal `primes(Max, Ps)` instantiates `Ps` to a list of primes `[2, 3, 5, ... ]`, and the remaining two goals inspect and display the primes in `Ps`. The method of input/output may vary from implementation to implementation, but here we follow the method of the GHC system on top of Prolog [7]. The predicate `outstream` is a system predicate for output; it accepts a list of commands and executes them in order. Each command has the same meaning as the goal of the same form of the underlying Prolog system. For example, the display

```
2
3
5
...
```

can be obtained by instantiating `Os` to `[write(2), nl, write(3), nl, write(5), nl, ... ]`. In the above program, the goal `outterms(Ps, Os)` generates the value of `Os`. It generates two elements `write(`p`)` and `nl` for each p in `Ps`.

To sum up, the goal `primes(Max, Ps)` generates `Ps`, `outterms(Ps, Os)` transforms `Ps` to `Os`, and `outstream(Os)` consumes `Os` to perform output operations. We may execute these goals sequentially from left to right, but alternatively, we can exploit pipeline parallelism. That is, the goal `outterms(Ps, Os)` can start determining the initial elements of `Os` as soon as `primes(Max, Ps)` determines the initial elements of `Ps`. Similarly, the goal `outstream(Os)` can start output operations as soon as `outterms(Ps, Os)` determines the initial elements of `Os`.

We may view these three goals as parallel processes communicating via shared variables `Ps` and `Os`. In general, a GHC goal can be regarded as a process that observes input bindings and generates appropriate output bindings depending on them*. Two goals communicate by instantiating and observing a shared variable appearing in common as their arguments. Such a shared variable is usually instantiated to a list of data or commands gradually from the head as computation goes on, and a list used in this manner is often called a *stream*.

* Although every goal can be called a process in principle, our pragmatics usually regards only long-lived (and possibly perpetual) ones as processes. The three body goals in Clause (4–1) are all long-lived, and this is why we introduce process interpretation in this section.

Parallel execution of conjunctive goals improves space efficiency also. Suppose we are to compute primes up to 1,000,000. Sequential computation would create a huge list of primes, convert it to a list of output commands, and then display the result. On the other hand, (pseudo-) parallel computation lets `outterms` consume the elements of `Ps` incrementally. The part of `Ps` once inspected by `outterms` to instantiate `Os` is no longer accessed in the program and becomes garbage. At any moment, only those elements of `Ps` that have been created by `primes` and not yet consumed by `outterms` need be maintained in the system. A similar argument applies to `Os` also. Therefore, (pseudo-) parallel execution will require less storage than sequential execution.

The goal `outstream(Os)` can be viewed as a process modeling the output device and its interface with the GHC system. It is quite natural to view peripheral devices as processes; they generally work in parallel with a program and form integral parts of the computational system. It is an important feature of GHC that interprocess communication and communication with peripheral devices are described using the same mechanism.

Now we will show the implementation of `primes` and `outterms`.

```
primes(Max,Ps) :- true | gen(2,Max,Ns), sift(Ns,Ps).          (4–2)

gen(N0,Max,Ns0) :- N0=<Max |
    Ns0=[N0|Ns1], N1:=N0+1, gen(N1,Max,Ns1).                  (4–3)
gen(N0,Max,Ns0) :- N0 >Max | Ns0=[].                          (4–4)

sift([P|Xs1],Zs0) :- true |
    Zs0=[P|Zs1], filter(P,Xs1,Ys), sift(Ys,Zs1).              (4–5)
sift([],     Zs0) :- true | Zs0=[].                           (4–6)

filter(P,[X|Xs1],Ys0) :- X mod P=\=0 |
    Ys0=[X|Ys1], filter(P,Xs1,Ys1).                           (4–7)
filter(P,[X|Xs1],Ys0) :- X mod P=:=0 |
    filter(P,Xs1,Ys0).                                        (4–8)
filter(P,[],     Ys0) :- true | Ys0=[].                       (4–9)

outterms([X|Xs1],Os0) :- true |
    Os0=[write(X),nl|Os1], outterms(Xs1,Os1).                 (4–10)
outterms([],     Os0) :- true | Os0=[].                       (4–11)
```

The stream `Ps` of primes is generated by removing multiples of primes from a stream of successive integers beginning with 2. The goal `gen(2, Max, Ns)` instantiates `Ns` to a stream of integers, and `sift(Ns, Ps)` creates `Ps` from `Ns`. For the predicate `gen`, we only note that the goal `N1:=N0+1` waits until `N0` is instantiated, computes `N0+1`, and unifies the result with `N1`.

The predicate `sift` will require much more detailed explanation. It assumes that the first argument receives a null list or an ascending list `[`n_1`, ... ]` of those

integers which are not multiples of any prime p less than n_1. Note that a list of successive integers beginning with 2 satisfies the above assumption. The predicate `sift` then returns a list of primes beginning with n_1 through the second argument. Clause (4–5) handles the recursive case: Since the first element `P`($= n_1$) of the input list is a prime by the assumption and the definition of a prime number, we let it be the first element of `Zs0`. We want to process the rest `Xs1` of the input list recursively, but it cannot be fed to the predicate `sift` as it is, because it may contain the multiples of n_1. So we filter out multiples of n_1 from `Xs1` and let `Ys` be the result. Then `Ys` is a null list or an ascending list `[`n_1'`, ... ]` of those integers which are not multiples of any prime p less than or equal to n_1. We want to rewrite the above condition $p \leq n_1$ on p to $p < n_1'$ to establish that `Ys` satisfies the input assumption of `sift`, and this rewriting is justified by the fact that any prime larger than n_1 should remain in the list `[`n_1'`, ... ]`, that is, n_1 is the largest prime less than n_1'. Therefore, we can call `sift(Ys, Zs1)` to generate the rest of the list `Zs0` of primes.

We conclude this section with remarks on the characteristics of GHC as a process description language. Firstly, unlike in conventional parallel programming languages, a GHC process is not a sequential process. A process is defined using other processes and these subprocesses may run in parallel. Unification is considered as a system-defined process that observes and generates bindings. Secondly, a GHC process allows declarative reading. It states (the relationship among) the values of its arguments like a Prolog goal. Thirdly, GHC allows dynamic process creation. For example, the above program generates a filtering process for each prime it has found.

5. DEMAND-DRIVEN COMPUTATION—FIBONACCI NUMBERS

The prime generator program in the last section performs *data-driven computation*; that is, the process `gen(2, Max, Ns)` autonomously generates a stream of integers, and the other processes such as `sift(Ns, Ps)` and `outterms(Ps, Os)` can consume input data as soon as the previous stage has generated them.

However, sometimes we may want to generate a stream on demand rather than autonomously. Consider the following program for generating a stream of Fibonacci numbers:

```
go :- true |
    fibonacci(Fs), outterms(Fs,Os), outstream(Os).          (5–1)

fibonacci(Ns) :- true | fib(1,0,Ns).                        (5–2)
fib(N1,N2,Ns0) :- true |
    N3:=N1+N2, Ns0=[N3|Ns1], fib(N2,N3,Ns1).                (5–3)
```

The goal `fibonacci(Fs)` autonomously generates a stream `Fs` of Fibonacci numbers `[1,1,2,3,5,8,13, ... ]`. The auxiliary predicate `fib` is used for maintaining

the last two numbers, and they are initialized to `1` and `0` in Clause (5–2) to start the stream with two `1`'s.

The problem with this goal is that there is no way to stop the generation of `Fs`. A termination mechanism can be easily incorporated if either the upper bound of the Fibonacci numbers to be generated or the number of elements is given beforehand. However, the termination condition may not always be given beforehand in such a simple form. In such cases, it is best to let the consumer of the stream have full control over its generation. Let us examine the following predicate:

```
fibonaccilazy(Ns) :- true | fiblazy(1,0,Ns).                    (5–4)
fiblazy(N1,N2,Ns0) :- Ns0=[N3|Ns1] |
    N3:=N1+N2, fiblazy(N2,N3,Ns1).                              (5–5)
fiblazy(_, _, [] ) :- true | true.                              (5–6)
```

Unlike `fibonacci(Fs)`, the goal `fibonaccilazy(Fs)` generates the stream `Fs` on demand. The difference comes from the difference between Clause (5–3) and Clause (5–5): Clause (5–3) unifies the third argument `Ns0` with the list structure `[N3|Ns1]` in the body, while Clause (5–5) unifies them in the guard. Clause (5–5) does not create a list structure by itself but waits until it comes from outside. The list structure must be created by a goal (say `g(Fs)`) that runs in conjunction with `fibonaccilazy(Fs)`, consuming the Fibonacci numbers. The structure created by `g(Fs)` should be a list of uninstantiated variables, which are to be unified with Fibonacci numbers by `fibonaccilazy(Fs)`.

For instance, when `g(Fs)` needs the first Fibonacci number, it instantiates `Fs` to a structure `[F0|Fs1]`. Then `fibonaccilazy(Fs)` compute the first number and unifies the result with `F0` using Clause (5–5). The second number is computed when `g(Fs)` further instantiates the rest of the structure `Fs1` to, say, `[F1|Fs2]`. If `g(Fs)` does not need the second or the subsequent Fibonacci numbers, it should instantiate `Fs1` to `[]`. In this case, Clause (5–6) is selected and the goal `fibonaccilazy(Fs)` terminates. Instantiating `Fs` or its sublist to a structure `[`*Variable*`|`*Rest*`]` can be interpreted as a demand for a new element; and instantiating it to `[]` can be interpreted as a command for terminating `fibonaccilazy`.

Note that Clause (5–5) is equivalent to the following, more usual form:

```
fiblazy(N1,N2,[N3|Ns1]) :- true |
    N3:=N1+N2, fiblazy(N2,N3,Ns1).                              (5–5′)
```

Now we show a complete program using `fibonaccilazy`. It accepts the commands `more` and `done` from the terminal, and displays the next Fibonacci number for `more` and terminates for `done`.

```
golazy :- true |
    fibonaccilazy(Fs), driver(Fs,IOs), instream(IOs).           (5–7)
```

```
driver(Fs,IOs0) :- true |
    IOs0=[read(X)|IOs1], checkinput(Fs,IOs1,X).                (5–8)
checkinput(Fs0,IOs0,more) :- true |
    Fs0=[N|Fs1], IOs0=[write(N),nl|IOs1],
    driver(Fs1,IOs1).                                          (5–9)
checkinput(Fs, IOs, done) :- true | Fs=[], IOs=[].             (5–10)
```

The predicate `golazy` creates three processes `fibonaccilazy(Fs)`, `driver(Fs, IOs)`, and `instream(IOs)`. The predicate `instream` is a system predicate for input operations. The goal `instream(IOs)` accepts a list of commands which in fact are Prolog input goals such as `read(X)`. Suppose `IOs` is instantiated to `[read(X)|IOs1]` and after a while the command `read(X)` is recognized by `instream`. Then `instream` reads a term from the terminal, and unifies it with `X`. Notice that a variable is sent with the command to get back a result. Such bi-directional use of `IOs` is just like the use of `Fs` as a demand-driven stream; actually, `instream(IOs)` can be regarded as a process that reads data on demand.

Another point to note on `instream` is that the goal `instream(IOs)` accepts output commands as well as input commands for the following reason. In non-interactive programs, input and output operations may be performed asynchronously; so we may get input data by `instream` and write output data by `outstream`. In interactive programs, however, we often need to guarantee the precise order of input operations from the keyboard and output operations to the display. For example, a prompting message must appear before the corresponding input command is executed. We cannot use separate streams for input and output in such a case, because two commands in different streams are not ordered and may be executed in any order. Therefore, we decided to let `instream(IOs)` accept output commands also. Now the goal `instream(IOs)` can be regarded as modeling the terminal with a keyboard.

One may doubt if `outstream` is necessary when `instream` accepts output commands. Certainly it could be dispensed with, but the existence of `outstream` may be convenient when we write non-interactive programs in which input and output devices are regarded as separate.

The goal `driver(Fs, IOs)` is a driver of the other two goals. It first sends a command `read(X)` to `instream(IOs)` (Clause (5–8)). If '`more`' is typed in, it sends a variable `N` to `fibonaccilazy(Fs)` to compute the next Fibonacci number, writes it, and then accepts the next command (Clause (5–9)). If '`done`' is typed in, it closes the streams `Fs` and `IOs` to terminate `fibonaccilazy(Fs)` and `instream(IOs)`, and terminates itself (Clause (5–10)).

An important feature of GHC is that data-driven and demand-driven computation can be expressed in the uniform framework of the language. However, we must be careful not to use demand-driven computation except when it is really necessary. Demand-driven computation involves bi-directional communication, and

this means that the constituent goals should be more tightly coupled to cope with more complex information flow. For this reason, programs using demand-driven computation are much harder to optimize than those using data-driven computation. For example, while the prime generator program in Section 4 runs almost without process switching on the implementation on top of Prolog, the program in Section 5 causes intensive process switching, which can be costly in most implementations.

6. CONCLUSION

We have described Guarded Horn Clauses by sample programs. GHC is quite simple compared with other parallel programming languages, but of course, further work is necessary to make it a truly practical language. Firstly, we must develop (very) efficient implementations and sophisticated optimization techniques to acquire a wide range of users. It is already shown that GHC can be implemented as efficiently as Prolog, but further improvement will be called for. Secondly, we must prepare a good environment for parallel programming. In particular, debugging facilities seem to be quite important. Thirdly, when programs and programming techniques are accumulated, we will have to analyze them and provide appropriate syntactic supports for terse description of typical programs, for developing large programs, and for assisting sophisticated optimization.

REFERENCES

[1] Andrews, G. R. and Schneider, F. B., "Concepts and Notations for Concurrent Programming", *Computing Surveys*, Vol. 15, No. 1, pp. 3–43, 1983.

[2] Kowalski, R., "Predicate Logic as Programming Language", *Proc. IFIP '74*, North-Holland, Amsterdam London, pp. 569-574, 1974.

[3] Clocksin, W. F. and Mellish, C. S., *Programming in Prolog*, 2nd ed., Springer-Verlag, Berlin Heidelberg New York Tokyo, 1984.

[4] Sterling, L. and Shapiro, E., *The Art of Prolog*, MIT Press, Cambridge, 1986.

[5] 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.

[6] Ueda, K., *Making Exhaustive Search Programs Deterministic*, ICOT Tech. Report TR-145, Institute for New Generation Computer Technology, Tokyo, 1985. Also in *Proc. Third Int. Conf. on Logic Programming*, Shapiro, E. (ed.), Lecture Notes in Computer Science 225, Springer-Verlag, Berlin Heidelberg, pp. 270–282, 1986.

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

[8] Ueda, K., *Guarded Horn Clauses*, ICOT Tech. Report TR-103, Institute for New Generation Computer Technology, Tokyo, 1985. Also in *Proc. Logic*

Programming '85, Wada, E. (ed.), Lecture Notes in Computer Science 221, Springer-Verlag, Berlin Heidelberg, pp. 168–179, 1986.

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

[10] Ueda, K., *Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard*, ICOT Tech. Report TR-208, Institute for New Generation Computer Technology, Tokyo, 1986.