TR-581

Designing a Concurrent Programming Language

by
K. Ueda

August, 1990

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

(To be presented at InfoJapan'90, Tokyo, October 1990)

Designing a Concurrent Programming Language

Kazunori Ueda

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

Abstract

This paper reviews the design and the evolution of a concurrent programming language Guarded Horn Clauses (GHC). GHC was born from a study of parallelism in logic programming, but turned out to be a simple and flexible concurrent programming language with a number of nice properties. We give both an abstract view of GHC computation based the notion of transactions and a concrete view, namely an operational semantics, based on reductions. Also, we discuss in detail the properties of GHC such as its atomic operations, which have much to do with the design of GHC.

1. Introduction

It seems that many people still regard concurrent programming as something special and difficult to learn. Indeed, concurrent programming may have inherent difficulties not in sequential programming, but the situation could be improved by developing better formalisms and programming languages. Many concurrent languages designed so far were built by adding a number of special constructs to existing sequential languages, adding certain complexity as well. So it is worth trying to build a concurrent language in a totally different way.

Guarded Horn Clauses (abbreviated to GHC) [21] [22] was designed to be a simple concurrent programming language with a very small number of primitive constructs. As its name suggests, GHC borrowed a lot of concepts from (ordinary) logic programming, which, together with the concept of a guard, were tailored into a concurrent language. Hence it is usually called a *concurrent logic programming language*.

The simplest account of GHC is the slogan:

$$\text{GHC} = \text{Horn clauses} + \text{Guards}.$$

This can be correlated with another slogan by Kowalski [8]:

$$\text{Algorithm} = \text{Logic} + \text{Control}.$$

Actually, the logical reading of a program expresses its static aspects, namely the relationship between input and output information; and the guards are used for expressing dynamic aspects or control, namely the causality between pieces of input and output information. The correlation thus suggests that GHC is a language for describing *concurrent algorithms*.

In logic programming, the execution result of a goal ":- G" under a program P has two aspects, namely

(i) the unsatisfiability of $P \cup \{\neg G\}$ (returned as a yes/no answer), and
(ii) substitutions that make G logical consequences of P.

From a programming language point of view, however, more interest is in the second aspect [9].

GHC as a concurrent language fully exploits the second aspect of computing substitutions. A building block of a GHC program, called a *process*, is an entity that observes and generates substitutions. For instance, a process `factorial(X,Y)` will generate a substitution `{Y=120}` when it observes a substitution `{X=5}`. Unlike ordinary logic programs, GHC programs specify the direction of computation. That is, GHC programs impose partial causal order on substitutions observed and generated by processes. This is done by restricting dataflow caused by unification, as will be described in Section 3.

2. Processes in GHC

GHC is a *reactive* (as opposed to transformational) language. In reactive languages, we are interested in the communication between a program and the

rest of the world performed in the course of computation, rather than the final result of computation that is the primary concern in transformational languages (the most typical of which are functional languages). Another interesting characterization of reactive or interactive programs is that the *input* to a program can depend on the *output* from the program. This property can be mimicked in functional languages with lazy evaluation, but GHC allows us to describe various forms of communication more naturally.

GHC is an *asynchronous* reactive language. Each piece of communication engaged in by a process is either from the process to the rest of the world or the other way around, and the sender of information is never blocked by the receiver. A history of communication between a process and the rest of the world can be divided into *transactions*, each transaction being an act of providing the process with a possibly empty *input substitution* and getting an observable, non-empty *output substitution*. The input can be empty because an autonomous process may not require any input. The output should be observable because that is usually what the outside world is waiting for. A process is considered erroneous if it fails to generate any output substitution when the outside world expects it.

Each transaction is quite similar to the whole computation of a transformational program. The difference is that the input from the outside world may depend on the outputs of *previous* transactions. This suggests that the behavior of a process can be formulated as a sequence of transactions, which is an external, abstract view of a process in our setting. Note that the above view of a process becomes very similar to the view of Theoretical CSP (TCSP) [7], a synchronous model of concurrency, by regarding each transaction as a single event.

As we have seen, GHC uses *substitutions* (sets of bindings between variables and their values) to model information communicated between processes. Substitutions are generated by *unification* and observed using *matching* (Section 3.1), a restricted form of unification. A nice thing is that these notions have a *logical* as well as an *algebraic* characterization, as pointed out by Maher [10] and studied extensively by Saraswat [16]. A substitution can be viewed as an equality constraint on the possible values of variables. A (binding) environment, which is the product of all the substitutions generated so far, can be viewed as the multiset (interpreted as the conjunction) of the constraints corresponding to the substitutions. In the GHC context, unification can be viewed as the *publication* of a constraint into the current environment, and matching can be viewed as the *checking* of whether the current environment implies a given constraint under the equality theory of GHC. GHC adopts Clark's equality theory [2] that models syntactic equality over finite terms. We could adopt equality theories other than Clark's without changing the essence of the language [16]. Moreover, we could allow constraints other than conjunctions of equalities. However, the current choice has the advantage that the generation and the observation of constraints can be easily computed.

3. GHC

How can we define the intended behavior of a process? The basic idea is to describe it in terms of *other* processes. This is attractive since the behavior can then be realized by the reduction of processes.

There are two possible styles for the description: One is to use process constructors [11], namely operators on processes, to compose a more complex process expression from simpler ones. The other style is to use rewrite rules of processes as in term rewriting systems. GHC has taken the rewrite-rule approach following the tradition of logic programming, though this choice is for historical reasons and is not essential. Saraswat [16] shows how concurrent logic programming can be reformulated using the constructor approach.

3.1 Syntax of GHC

Now let us introduce GHC. GHC has borrowed many notions from logic and logic programming, which enables a terse introduction of the language. We assume here that the following notions are defined as usual [9]:

> variables, function(symbol)s, constants (regarded as 0-ary functions), predicate(symbol)s, terms, atom(ic formula)s, substitutions, renaming, unification.

We say that a term t_1 *matches* a term t_2 if there is a substitution such that $t_1\theta = t_2$ ('$=$' denoting syntactic equality). Matching is called *one-way unification* also.

A *program* is a set of guarded clauses. A *guarded clause* is of the form

$$h :- G \mid B,$$

where h is an atom, and G and B are multisets of atoms. h is called the *head* of the clause; atoms in G are called *guard goals*; and atoms in B are called *body goals*. The part before the *commitment operator* '|' is called the *guard*, and the part after '|' is called the *body*.

A clause with an empty body is called a *unit clause*. The set of all clauses in a program whose heads have the predicate symbol p is called the *procedure* for p. A goal with the predicate symbol p is said to *call* p.

Informally, each guarded clause is a conditional rewrite rule of goals, where

- h is the template that should match a goal (say g) to be rewritten,
- G is the auxiliary condition for the rewriting (G must be executed without instantiating g), and
- B is the multiset of (sub)goals to replace g.

That a program is a *set* means that the duplication (up to renaming of variables) of guarded clauses is insignificant, not to mention their ordering. On the other hand, G and B are *multisets* because two syntactically identical goals may behave differently due to indeterminacy.

To run a program, we use a goal clause of the form

$$:- B.$$

which specifies the initial multiset of body goals.

A goal is either a unification goal of the form $t_1 = t_2$ or a non-unification goal. A unification goal, whose behavior is predefined in the language, may generate a substitution and constrain the possible values of variables. A non-unification goal is rewritten to other goals using guarded clauses, possibly after observing a substitution. The guard of a guarded clause specifies what substitution should be observed before rewriting, and provides the language with a synchronization mechanism.

3.2 Flat GHC

The above definition of GHC allows any atom to occur as a guard goal. However, this proved to be unnecessarily expressive as a concurrent language (Section 6), which motivated us to move to a subset of GHC called *Flat GHC*.

Since guard goals are used as conditions, we first define a class of predicates, called *test predicates*, that are appropriate for the purpose. A predicate p is called a *test predicate* if the procedure for p is defined by a set of *unit* clauses. Calls to a test predicate have a property that they do not generate observable substitutions; the only thing that matters is whether they succeed or not.

A Flat GHC program is a set of *flat guarded clauses*, clauses in which guard goals are restricted to unification goals and calls to *test* predicates.

3.3 Operational Semantics of Flat GHC

Now we formalize the operational semantics of Flat GHC. We follow the structural approach of Plotkin [14], which is now a standard way of describing operational semantics formally. The structural operational semantics of *full* GHC is found in [15].

- $\forall . (\neg(f(\mathtt{X}_1, \ldots, \mathtt{X}_m) = g(\mathtt{Y}_1, \ldots, \mathtt{Y}_n)))$, for all pairs f, g of distinct functions (including constants).
- $\forall . (\neg(t = \mathtt{X}))$, for each term t other than and containing $\mathtt{X}$.
- $\forall . (\mathtt{X} = \mathtt{X})$.
- $\forall . (f(\mathtt{X}_1, \ldots, \mathtt{X}_m) = f(\mathtt{Y}_1, \ldots, \mathtt{Y}_m) \supset \bigwedge_{i=1}^{m} (\mathtt{X}_i = \mathtt{Y}_i))$, for each function f.
- $\forall . (\bigwedge_{i=1}^{m} (\mathtt{X}_i = \mathtt{Y}_i) \supset f(\mathtt{X}_1, \ldots, \mathtt{X}_m) = f(\mathtt{Y}_1, \ldots, \mathtt{Y}_m))$, for each function f.
- $\forall . (\mathtt{X} = \mathtt{Y} \supset \mathtt{Y} = \mathtt{X})$
- $\forall . (\mathtt{X} = \mathtt{Y} \wedge \mathtt{Y} = \mathtt{Z} \supset \mathtt{X} = \mathtt{Z})$

Figure 1. Clark's equality theory $\mathcal{E}$

Let B be a multiset of goals, and C a multiset of equations that represents a (binding) environment of B. The current *configuration* is a pair $\langle B, C \rangle$, which records the goals to be reduced and the current environment. A computation starts with the initial configuration $\langle B_0, \emptyset \rangle$, where B_0 is the body of the given goal clause.

What we are going to define is a transition relation $c_1 \longrightarrow c_2$, which reads "the configuration c_1 can be reduced to the configuration c_2." When we need to explicitly mention the program P being used, we use the form $P \vdash c_1 \longrightarrow c_2$, which reads "under the program P, c_1 can be reduced to c_2." By $\stackrel{*}{\longrightarrow}$ we denote the reflexive, transitive closure of $\longrightarrow$. The natural deduction form

$$\frac{P_1 \vdash t_1}{P_2 \vdash t_2} \quad (\text{if } \mathit{Cond})$$

says that if the transition t_1 can happen under P_1 and the condition Cond holds, the transition t_2 can happen under P_2. The numerator and the condition are omitted if they are empty.

We have three rules. In the following rules, $F \models G$ means that G is a logical consequence of F, and $\mathcal{V}_F$ means the set of all variables occurring in F. $\forall \mathcal{V}_F . F$ and $\exists \mathcal{V}_F . F$ are abbreviated to $\forall . F$ and $\exists . F$, respectively. Also, following [18], we denote $\exists (\mathcal{V}_F \backslash V) . F$ by $\delta V . F$, where V is a finite set of variables. We assume that there is an injection, denoted '$\overline{\ }$', from the set of predicates to the set of functions, which is naturally extended to an injection from the set of atoms to the set of terms. $\mathcal{E}$ denotes Clark's equality theory (Figure 1).

$$\frac{P \vdash \langle B_1, C_1 \rangle \longrightarrow \langle B_1', C_1' \rangle}{P \vdash \langle B_1 \cup B_2, C_1 \rangle \longrightarrow \langle B_1' \cup B_2, C_1' \rangle} \quad \text{(i)}$$

$$\frac{P \vdash \langle (\overline{b} = \overline{h_i}) \cup G_i, C \rangle \stackrel{*}{\longrightarrow} \langle \emptyset, C \cup C_g \rangle}{(h_i :- G_i \mid B_i) \cup P \vdash \langle b, C \rangle \longrightarrow \langle B_i, C \cup C_g \rangle} \left(\begin{array}{l} \text{if } \mathcal{E} \models \forall . (C \supset \delta \mathcal{V}_b . C_g) \\ \text{and } \mathcal{V}_{(h_i, G_i, B_i)} \cap \mathcal{V}_C = \emptyset \end{array} \right) \quad \text{(ii)}$$

$$P \vdash \langle t_1 = t_2, C \rangle \longrightarrow \langle \emptyset, C \cup (t_1 = t_2) \rangle \qquad \text{(iii)}$$

Rule (i) expresses concurrent reduction of a multiset of goals. Rule (ii) says that a goal b can be reduced using a guarded clause "h_i :- G_i | B_i" if the head unification $b = \overline{h_i}$ and the guard goals G_i can be reduced out without affecting the variables in b. This means that the head unification is restricted to matching effectively. The condition $\mathcal{V}_{(h_i, G_i, B_i)} \cap \mathcal{V}_C = \emptyset$ guarantees that the guarded clause has been renamed using fresh variables. Rule (iii) says that a unification goal publishes (or posts) a constraint to the current environment.

3.4 Interacting with a Flat GHC Process

How does the above transition relation relate to our external view of a process stated in Section 2? Roughly speaking, a multiset of goals implements a process, and a sequence of reductions realizes a transaction. Recall that a transaction is an act by an observer process of providing an observee process with a possibly empty input substitution and getting an observable (and hence non-empty) output substitution. In the following, we consider in more detail how a transaction is realized by reductions.

Consider the initial configuration $\langle P \cup O, \emptyset \rangle$, where the process O is assumed to be observing the process P, and assume the transition $\langle P \cup O, \emptyset \rangle \xrightarrow{*} \langle P' \cup O', C'' \rangle$ has been made so far. Then each element of C'' is either the one posted by O or the one posted by P (note that C'' is a multiset). Let C''_O be the set of constraints posted by O (including the constraints on local variables generated during the execution of guards). C''_O is regarded as the current ***knowledge*** of O. C''_P is defined similarly. Henceforth, to denote $\langle P' \cup O', C'_P \cup C''_O \rangle$, we use a more *modular* notation $\langle P', C'_P \rangle \cup \langle O', C''_O \rangle$; also, we use abbreviations such as $c_1 \cup (c_2 \longrightarrow c'_2)$ which means $c_1 \cup c_2 \longrightarrow c_1 \cup c'_2$.

Now assume

(i) a (possibly empty) transition by the observer

$$\langle P', C'_P \rangle \cup (\langle O', C''_O \rangle \xrightarrow{*} \langle O'', C''_O \cup \alpha \rangle)$$

is made without reference to C'_P (i.e, the transition must be such that it can be made without $\langle P', C'_P \rangle$), where $\mathcal{E} \models \exists.(C''_O \cup \alpha)$ (i.e., O's knowledge is still consistent), and then

(ii) a (possibly empty) transition by the observee

$$(\langle P', C'_P \rangle \xrightarrow{*} \langle P'', C''_P \rangle) \cup \langle O'', C''_O \cup \alpha \rangle$$

is made (possibly with reference to $C''_O \cup \alpha$), and then

(iii) a (non-empty) transition by the observer

$$\langle P'', C''_P \rangle \cup (\langle O'', C''_O \cup \alpha \rangle \longrightarrow \langle O''', C''_O \cup \alpha \cup \beta_1 \rangle \longrightarrow \cdots \longrightarrow \langle O'''', C''_O \cup \alpha \cup \beta_1 \cup \cdots \cup \beta_n \rangle)$$

is made, where $\beta \stackrel{\text{def}}{=} \beta_1 \cup \cdots \cup \beta_n$ is such that

(a) for each β_i posted from a clause *body*, $\mathcal{E} \models \forall.(C''_O \cup \alpha \cup \beta_1 \cup \cdots \cup \beta_{i-1} \supset \delta\mathcal{V}_O.(C''_O \cup \alpha \cup \beta_1 \cup \cdots \cup \beta_i))$ (i.e., non-local constraints are not posted from clause bodies) and

(b) $\mathcal{E} \not\models \forall.(C''_O \cup \alpha \supset \delta\mathcal{V}_O.(C''_O \cup \alpha \cup \beta))$ (i.e., a new constraint is observed).

Then, we say that O has engaged in a (normal) transaction $\langle\!\langle \alpha, \beta \rangle\!\rangle$ with P. The above transitions need not happen strictly in that order; the point is that α is first generated with reference to C''_O only, and then β is generated without constraining non-local variables. The reductions of P' can be interleaved with these two phases. Note that following this transaction, O may engage in the next transaction with P.

As well as normal transactions, we must be able to model various abnormal phenomena. This is because we want to distinguish between a process that always behaves normally and a process that only sometimes behaves normally. First, the observee may post a constraint inconsistent with the existing ones; or in algebraic terms, a unification body goal may *fail*. In that event, *any* constraint and its negation becomes observable, and from then on each goal in the observee *can and cannot* be reduced using *any* clause. In a word, the observee has fallen into *chaos*, a totally unpredictable condition. Interestingly, chaos in GHC is very similar to chaos in TCSP introduced in order to model a totally undefined indeterminate process.

Second, the observee may fail to generate an observable output constraint in response to a given input constraint for various reasons, which is called *inactivity*. The reason will be one of the following:

(1) the observee has been reduced out (i.e., succeeds) with no observable output.

(2) the observee has been reduced, with no observable output, to a multiset of goals that does not allow further reduction in the current environment,

(3) the observee has fallen into infinite computation.

We call the first *success*, the second *deadlock*, and the third *divergence* [23]. Of these, divergence consumes unbounded computation resource beyond the observer's control, while success and deadlock do not. Unless the scheduling of goals is fair, a divergent process may monopolize the computational resource, blocking the execution of other processes running concurrently. Hence divergence is worse than, and should be distinguished from, non-divergent inactivity. It is mathematically attractive to regard divergence as chaos, as in TCSP. This treatment equates the two apparently different but most undesirable phenomena in Flat GHC, divergence and the failure of unification.

Success and deadlock cannot be distinguished by observable output constraints. However, sometimes

it is useful to treat them separately. The observer of a process usually gives an input constraint to observe an output constraint, but may sometimes do so to terminate the observee. Then, success is not an abnormal phenomenon any more, and should be distinguished from deadlock. Thus the abstract view of a process depends on what phenomena the observer is interested in.

Note that a normal transaction is of a finite nature: it records the observation of finite output information made in finite time. A meaningful Flat GHC process can be non-terminating and can engage in an infinite number of transactions, but it should be non-divergent and controllable in the sense that it should not run indefinitely without observing an infinite number of non-empty input constraints.

We imposed the restriction that input constraints should be consistent with the observer's knowledge, because otherwise the observer itself would go chaotic. Our assumption is that the observer must be well-behaved, while it is unreasonable to assume anything about the observee. An observer is said to be *faithful* if it eventually observes some of the observable output constraints generated by the observee.

Now we claim that a Flat GHC process is adequately characterized by the set of all possible sequences of transactions made by all possible faithful observers. This view of processes gives a sufficiently weak, but still reasonable, equivalence relation for processes, which abstracts away the notion of reductions. Whether infinite sequences should be included or are approximated by sets of finite sequences depends on whether fairness is considered or not. Our current position is to say nothing about fairness in the definition of the language. However, the notion of the knowledge of a process we have given above can be used to discuss whether information sent by a sender process is eventually delivered to a receiver or not.

4. Some Properties of Flat GHC

4.1 Atomic Operations

One of the motivations that lead us to design GHC was the examination of atomic operations in Concurrent Prolog [19]. Concurrent Prolog (including its offsprings) and the language cc(↓, —) [16] have the notion of *atomic publication*, in which the publication of a constraint by a process is done *upon* reduction and only when it does not cause inconsistency. Atomic publication may have to 'test-and-set' a number of variables at the same time, which can be costly in a distributed implementation. In GHC and PARLOG [4], on the other hand, the publication of a constraint is separated from the reduction of a non unification goal and is done by an independent unification goal. This alternative, called *eventual publication*, is advantageous for implementation, though some programming techniques can be used only in atomic publication languages. Interestingly, our choice of eventual publication recently lead to the idea of the message-oriented scheduling of goals [25], a scheduling that contrasts sharply with the ordinary one.

Moreover, GHC enjoys anti-substitutivity [21], a property which allows the delay of interprocess communication between two occurrences of a shared variable. Anti-substitutivity allows two occurrences of the same variable to have even inconsistent values. (Such a variable is referred to as a non-atomic variable in [20].) Fortunately, Maher's logical characterization of the communication mechanism of GHC-like languages [10] later assured that anti-substitutivity is quite a natural notion.

4.2 Binding Environments

In GHC, constraints obtained by executing the guards of clauses cannot affect the caller side. This means that a single binding environment is sufficient for managing the values of variables, while in OR-parallel Prolog and full Concurrent Prolog, multiple environments need to be maintained.

The binding environment of GHC is *monotonic*; the publication of a new constraint does not invalidate any previous observations done by clause guards. In other words, if a clause C can reduce a goal g in some environment, it can reduce g in an environment with more constraints. Thanks to this property, GHC can allow eventual publication and anti substitutivity.

4.3 Treatment of Failure

The original definition of GHC did not state much about failure. In Prolog, a goal is considered to have failed if no clause can resolve it, and many other concurrent logic languages followed this tradition. However, we had felt that this was inappropriate for GHC. GHC separated unification from reduction, so it is quite reasonable to distinguish between the failure of reduction and the failure of unification which have quite different behavioral consequences.

It is worth noting that the transition relation of Flat GHC does not rely on any notion of failure. Many other concurrent logic languages and Prolog allow us to write a clause that is tried only when all the preceding clauses (assuming a program is a *sequence* of clauses) turn out to be inapplicable forever. However, it is not so easy to correctly check if a clause cannot reduce a given goal forever. First, the check requires that guard goals and head unification be executed

concurrently in general. Second, head unification becomes more difficult because we must detect that the head `p(a,b)` cannot unify with the goal `p(X,c)` or `p(X,X)` forever. Flat GHC, on the other hand, allows a clause guard to be executed sequentially in a pre-determined order; the matching of `p(a,b)` with `p(X,c)` can be left suspended at the first argument. Although a Flat GHC program represents minimum sequentiality, the only places that require concurrent execution are between body goals reduced from the top-level goals and between guarded clauses trying to reduce a goal.

Failure of a unification body goal in GHC is an exceptional situation which is essentially the same as division-by-zero in any programming language. Consider the constraint `X = 5/0` over real numbers. This is equivalent to `X*0 = 5`, namely `0 = 5`, whose publication would cause inconsistency. How to handle such an exception is discussed in Section 6.

5. Advantages of Flat GHC as a Concurrent Language

Now let us summarize advantages of Flat GHC as a concurrent language.

(1) A process is defined using other processes, unlike many concurrent languages in which processes are defined using iteration. This is consistent with the use of streams, which are a recursive data structure, for interprocess communication.

(2) As we have seen, the monotonic property of binding environments realizes a natural synchronization mechanism of waiting until sufficient information is observed.

(3) The mechanism for interprocess communication is expressive enough to naturally describe data-driven and demand-driven computation and dynamically evolving process structures. TCSP and CCS [11] allowed recursive definitions of processes, but could not deal with dynamically evolving process structures because they lacked the ability to create and pass new communication channels. It is only recently that CCS was extended to deal with evolving process structures [12].

(4) A sequence of messages (i.e., a stream) is represented using an explicit data structure, namely a list. This is unlike most languages, in which message sequences are implicit and a set of dedicated operations is provided for them. GHC uses operations like Lisp's *car* and *cons* for interprocess communication. No specific communication protocols (e.g., FIFO communication using streams) are built-in because they are programmable.

(5) GHC allows various views. It can be viewed as a process description language, a dataflow language, and a concurrent assembly language. It can be viewed also as a logic programming language in the sense that the result of a computation allows declarative interpretation. A GHC program is better amenable to declarative reading than Prolog programs with extralogical operations such as I/O.

6. Evolution of GHC

GHC was born at the very end of 1984 from close examination of parallelism in logic programming, the direct trigger being the study of the atomic operations and the binding environment mechanism of (full) Concurrent Prolog. No essential change has been made since then, but the proposed language has been studied from various aspects.

One good result of the study is that now we understand the language much better than when it was born, which means that the language is more robust than before. The study of atomic operations, of unusual behavior such as failure, and of relationship with other logic languages, concurrent languages and models of concurrency helped explain the language better. The study of formal semantics of concurrent logic languages by many people (e.g., [3], [10], [13], [16]) also helped our understanding.

6.1 Subsetting

Another important result is the identification of subsets which can be more efficiently implemented but are still useful.

Full GHC allowed any atom to occur as a guard goal, trying to retain the expressive power of full Concurrent Prolog as much as possible. However, then, a unification body goal may have to be suspended when it is executed as a subgoal of some guard goal. More importantly, our programming experience showed that guard goals are used only for the simple testing of conditions. Since guard goals are given limited communication capability, they are not very powerful anyway. We had been unwilling to implement the guard mechanism of full GHC for these reasons, and finally decided to allow only predefined predicates to be called from a guard. This was our first approximation to Flat GHC, which was clearly influenced by the subsetting of Concurrent Prolog to Flat Concurrent Prolog [19].

However, the above-mentioned way of subsetting was not quite satisfactory for a rather idealized programming language like GHC, because it depends on the arbitrary choice of predefined predicates. We felt that it would be much better to state what properties are sufficient for a predicate to be called a test predicate. The definition of Flat GHC in Section 3.2 is one solution.

Flatness as defined in Section 3.2 guarantees that no body goals are spawned by the execution of a guard. All the synchronization conditions of unification can then be analyzed statically and without global analysis. Flat GHC does, however, allow nested guard goals. Calls to test predicates in Flat GHC have a desirable property that they are *deterministic*, that is, the current environment uniquely determines whether the calls succeed in it or not.

Once we have defined within the framework of GHC what are test predicates, an actual implementation of Flat GHC could reasonably restrict guard goals to calls to predefined predicates. A wonderful discovery of the language Oc [6] was that guard goals in (Flat) GHC are not essential and can be disallowed theoretically.

A problem with Flat GHC is that it is left to programmers to guarantee that the binding environment never becomes inconsistent; a program goes chaotic once its binding environment becomes inconsistent. Of course, it is most desirable that such insecurity be detected at compile time.

The main reason for the insecurity is that two or more processes sharing a variable may try to instantiate it non-cooperatively. Doc [5] and Janus [17] introduced annotations (attached to occurrences of variables) to syntactically guarantee that only one process can instantiate a variable. On the other hand, Ueda and Morita [25] showed that simple mode analysis can be used to guarantee the same property. The mode system provides a unified framework for mode declaration (of which annotation is one possible way), mode inference and mode checking. Restriction to one producer per variable disallows a variable to be used as a shared resource with 'multiple-writers'. However, such a shared resource does not have to be implemented using a variable, because it can be implemented using a process. The mode system has been designed so that it can be incorporated into Flat GHC as a new language construct; in effect, we have proposed a further subset that could be called *Moded Flat GHC*.

Unfortunately, the above restriction is still insufficient for guaranteeing the consistency of the binding environment because of the occur-check problem. One solution is to adopt *rational* terms instead of *finite* terms, as in some Prolog systems and Janus. This makes it possible to create infinite terms in a finite time, while in GHC, infinite terms can be created only using infinite recursion. The consequences of this is yet to be studied.

6.2 Flat GHC and KL1

Although Flat GHC has a number of good properties, it is not quite appropriate for programming parallel computers.

First, GHC is a reactive language in which processes are assumed to be cooperative rather than competitive. In actual applications, however, not all processes may be cooperative with others. An example is a user process running concurrently with an operating system.

Second, although a GHC program fully expresses the possibility of parallel execution, it does *not* specify at all how it *should* be executed. It may be a good platform for parallel processing because no unnecessary sequentiality is imposed. However, it is a *concurrent* language, not a *parallel* language in which one can specify how processes should be executed on a parallel computer.

The separation of concurrency and parallelism is not a design flaw but a deliberate decision. Since language constructs for specifying parallel execution may depend on the computation models that reflect underlying implementations, they should be defined separately.

The language called KL1 [1] takes these two issues into account. It is based on Flat GHC, but has included the '*shōen*' (manor) construct so that a process may have full control over another process that may not be cooperative. The shōen construct enables a process to control the execution of another process executed within a shōen and the resource it consumes. The shōen construct also handles exceptional situations of a process such as failure and deadlock. For parallel execution, KL1 provides a construct for specifying which goal should be executed on which processor and with what priority.

7. Conclusion

We have reviewed the design and the evolution of GHC. Macroscopically, GHC should be regarded as a concurrent programming language rather than a logic programming language. However, when we look into the language more microscopically, we find that each transaction is similar to partial refutation in logic programming and that the communication mechanism allows an elegant logical characterization.

It seems that concurrent logic programming is often misunderstood because it stemmed from logic programming. However, it is not just an incomplete variant of logic programming. We believe that concurrent logic programming is interesting in its own right and deserves much more attention and study.

The research on GHC has been focused on the semantical aspects of the language. The current syntax of GHC is not essential at all; software engineering aspects such as the modularization of large pro-

grams are important but separate issues to be considered. However, some software engineering issues are already addressed in GHC: it provides an abstraction and encapsulation mechanism based on processes, and we can put object-based concurrent programming into practice.

Finally, we note that the current status of GHC has been influenced by many works in the fields of logic programming and concurrent programming and also by many discussions with a number of researchers.

Acknowledgments

We are indebted to Kenji Horiuchi, Masahiro Hirata and Keiji Hirata for valuable comments on earlier versions of this paper.

References

1. Chikayama, T., Sato, H. and Miyazaki, T., Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. Int. Conf. on FGCS'88*, ICOT, Tokyo, 1988, pp. 230–251.

2. Clark, K. L., Negation as Failure. In *Logic and Data Bases*, Gallaire, H. and Minker, J. (eds.), Plenum Press, New York, 1978, pp. 293–322.

3. Gerth, R., Codish, M., Lichtenstein, Y. and Shapiro, E., Fully Abstract Denotational Semantics for Flat Concurrent Prolog. In *Proc. Third Annual Conf. on Logic in Computer Science*, IEEE, 1988, pp. 320–335.

4. Gregory, S., *Parallel Logic Programming in PARLOG: The Language and its Implementation*. Addison-Wesley, Wokingham, England, 1987.

5. Hirata, M., Programming Language Doc and Its Self-Description or, $X = X$ is Considered Harmful. In *Proc. 3rd Conf. of Japan Society of Software Science and Technology*, 1986, pp. 69–72.

6. Hirata, M., Parallel List Processing Language Oc and Its Self-Description. *Computer Software*, Vol. 4, No. 3 (1987), pp. 41–64 (in Japanese).

7. Hoare, C. A. R., *Communicating Sequential Processes*. Prentice-Hall International, UK, London, 1985.

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

9. Lloyd, J. W., *Foundations of Logic Programming* (Second ed.), Springer-Verlag, Berlin, 1987.

10. Maher, M. J., Logic Semantics for a Class of Committed-Choice Programs. In *Proc. Fourth Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 858–876.

11. Milner, R., Process Constructors and Interpretations. In *Information Processing 86*, Kugler, H. -J. (ed.), North-Holland, 1986, pp. 507–514.

12. Milner, R., Parrow, J. and Walker, D., A Calculus of Mobile Processes, Parts I and II. ECS-LFCS-89-86, Dept. of Computer Science, Univ. of Edinburgh, 1989.

13. Murakami, M., A Declarative Semantics of Parallel Logic Programs with Perpetual Processes. In *Proc. Int. Conf. on FGCS'88*, ICOT, Tokyo, 1988, pp. 374–381.

14. Plotkin, G. D., A Structural Approach to Operational Semantics. DAIMI FN-19, Computer Science Dept., Aarhus Univ., Denmark, 1981.

15. Saraswat, V. A., GHC: Operational Semantics, Problems and Relationship with CP(|,&). In *Proc. 1987 Symp. on Logic Programming*, IEEE, 1987, pp. 347–358.

16. Saraswat, V. A., Concurrent Constraint Programming Languages. Ph. D. Thesis, CMU, 1989.

17. Saraswat, V., Kahn K. and Levy J., Janus: A Step Towards Distributed Constraint Programming. SSL 89-108, System Sciences Lab., Xerox PARC, 1989.

18. Saraswat, V. A. and Rinard, M., Concurrent Constraint Programming (Extended Abstract). In *Conf. Record of the Seventeenth Annual ACM Symp. on Principles of Programming Languages*, ACM, 1990, pp. 232–245.

19. Shapiro, E. Y., Concurrent Prolog: A Progress Report. *Computer*, Vol. 19, No. 8 (1986), pp. 44–58.

20. Shapiro, E., The Family of Concurrent Logic Programming Languages. *Computing Surveys*, Vol. 21, No. 3 (1989), pp. 413–510.

21. Ueda, K., Guarded Horn Clauses. Doctoral thesis, Information Engineering Course, Faculty of Engineering, Univ. of Tokyo, 1986.

22. Ueda, K., Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. ICOT Tech. Report TR-208, ICOT, Tokyo, 1986. Also in *Programming of Future Generation Computers*, Nivat, M. and Fuchi, K. (eds.), North-Holland, 1988, pp. 441–456.

23. Ueda, K. and Furukawa, K., Transformation Rules for GHC Programs. In *Proc. Int. Conf. on FGCS'88*, ICOT, Tokyo, 1988, pp. 582–591.

24. Ueda, K., Parallelism in Logic Programming. In *Information Processing 89*, Ritter, G. X. (ed.), North-Holland, 1989, pp. 957–964.

25. Ueda, K. and Morita, M., A New Implementation Technique for Flat GHC. In *Proc. Seventh Int. Conf. on Logic Programming*, MIT Press, 1990, pp. 3–17.