ICOT Technical Report: TR-481

TR-481

GHC-A Language for a New Age of Parallel Programming

by
K. Furukawa & K. Ueda

June, 1989

ICOT

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

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

GHC — A Language for a New Age of Parallel Programming

Koichi Furukawa and Kazunori Ueda

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

Abstract

A parallel logic programming language GHC, proposed by Ueda (1985), is now playing a very important role in the Fifth Generation Computer Project. It is a successor of Relational Language (Clark and Gregory 1981), Concurrent Prolog (Shapiro 1983) and Parlog (Clark and Gregory 1984). Since GHC is totally based on parallelism, it provides a genuine tool for parallel programming. It encourages programmers to write parallel algorithms and therefore gives a foundation of parallel programming. We have also developed a program transformation technique for GHC programs which preserves the external behaviour of the original programs. To show the validity of the transformation technique, we have developed a formal semantics of possibly non-terminating GHC programs. The highly parallel prototype hardware of our project is now being developed to support the efficient execution of GHC programs.

1. Introduction

The Fifth Generation Computer Project started in 1982 to develop an entirely new computer system for knowledge processing. There are two significant technical characteristics of the project: the adoption of logic programming as the central concept of the system and the pursuit of highly parallel computer architecture for the very fast execution of logic programs. At the beginning, we provisionally chose Prolog as the project's kernel language, since there were no other realistic logic programming languages. We appreciated the potential ability of Prolog as a very high level user language for developing knowledge processing application programs. However, we noticed a major defect of the language in its expressiveness for parallel situations: It is very hard to write an operating system in Prolog because it has no concurrency concept.

In 1981, an entirely new logic programming language, called Relational Language, was proposed by Clark and Gregory (1981). It is a logic programming language with the concept of concurrency, but without the concept of backtracking. To introduce concurrency, they adopted the notion of "guarded commands" proposed by Dijkstra (1975). Each clause has a guard which must be satisfied in order to be selected as the subsequent computation branch. They also introduced the notion of suspension for synchronisation.

After that, there appeared two successors of the language: Concurrent Prolog by Shapiro (1983) and Parlog by Clark and Gregory (1984). We selected these two languages as candidates for the kernel language of our project, and started very careful studies on both of these languages from various viewpoints including expressive power, semantics and ease of implementation. As a result, Ueda (1985) developed another parallel logic language, called Guarded Horn Clauses (GHC). GHC turned to be a good compromise of Concurrent Prolog and Parlog. For ordinary programs, it is as expressive as Concurrent Prolog and as efficient as Parlog. Moreover, GHC is both syntactically and semantically the simplest of them.

Flat GHC (FGHC), which is a simplified version of GHC, has been selected as the core of the FGCS kernel language, KL1, which interfaces parallel software and the highly parallel prototype hardware, the Parallel Inference Machine (PIM).

2. GHC — A Brief Introduction

GHC is a general-purpose parallel language for programming with communicating processes.

Although both Prolog and GHC are based on input resolution and unification, the purposes of the languages are quite different. Prolog is a (restricted) theorem prover for Horn-clause logic, while GHC is not directly aimed at theorem proving that involves searching. The primary design goal of GHC is to provide a simple way to describe a process that may interact with other processes and the outside world. This has been achieved by regarding a goal as a process.

A process is defined in terms of other processes. Interprocess communication is realised by the information transfer caused by unification. The result of a GHC computation is the history of its interaction (i.e., the observation and the generation of substitutions) with the outside world, while the result of a Prolog computation is an answer substitution returned upon success.

A GHC program is a set of *guarded Horn clauses* (also called *(program) clauses*) of the form

$$h \text{ :- } G \mid B$$

where h is an atomic formula called the *head* and G and B are multisets of atomic formulae. Each element of the multisets is called a *goal*. A non-empty multiset with n atomic formulae is written as $g_1, g_2, \ldots, g_n$, and an empty multiset is written as `true`.

The *commitment operator* '|' divides the clause into two parts: the left-hand side is called the *guard* and the right-hand side is called the *body*. The head h is part of the guard. Roughly speaking, each clause describes a conditional rewrite (or reduction) rule for goals. The head is the template of a goal to be rewritten; the rest of the guard specifies the additional conditions for rewriting; and the body specifies the multiset of new goals that replaces the old goal.

The execution of a program begins with the initial multiset of goals specified by a goal clause of the following form:

$$\text{:- } B$$

Each goal (say g) in B rewrites itself using one of the program clauses unless it is a predefined unification goal of the form $t_1 = t_2$. A unification goal $t_1 = t_2$ unifies t_1 and t_2, and the generated substitution, if any, is applied to all the goals running. Goals run in parallel.

The clause used for rewriting a goal g is determined by executing the guards of the program clauses in parallel. For g to execute the guard h :- G of a program clause C means to execute $g = h$ and G in parallel. The important rule is that *the execution of* $g = h$ *and* G *cannot instantiate* g. The fragment of computation that would instantiate g is suspended. The suspended fragment of computation can be resumed when g gets more instantiated by other goals running in parallel with g. This rule is called *the rule of synchronisation*, because it is used for the synchronisation of goals running in parallel.

If the goal g succeeds in solving the guard of C, it can *commit to* C and replace itself by the body goals of C. When g can commit to two or more program clauses, g selects one of them and commits to it. This rule is called the *rule of commitment* and the mechanism is called *committed-choice nondeterminism*. A goal g is said to *succeed* if it becomes an empty multiset of goals by repeated rewriting.

Let us consider a ticket reservation counter with two windows. Two queues will be formed, one for each window. We assume that the requests from the two queues should be serialised behind the counter to gain access to a single shared resource. The serialiser can be defined in GHC as a process `merge(`Xs`,`Ys`,`Zs`)` which merges two queues Xs and Ys into a single queue Zs:

M_1: `merge([X|Xs],Ys,Zs) :- true | Zs=[X|Us], merge(Xs,Ys,Us).`
M_2: `merge(Xs,[Y|Ys],Zs) :- true | Zs=[Y|Us], merge(Xs,Ys,Us).`
M_3: `merge([],Ys,Zs) :- true | Zs=Ys.`
M_4: `merge(Xs,[],Zs) :- true | Zs=Xs.`

The first argument of M_1, `[X|Xs]`, means that M_1 is waiting for a request from the first window. Similarly, M_2 is waiting for a request from the second window. M_3 and M_4 handle the cases where no more requests will arrive at the first and the second windows, respectively.

The following is a simple example using the `merge` program:

```
:- queue1(As), queue2(Bs), merge(As,Bs,Cs), serve(Cs).
```

The goals `queue1(As)` and `queue2(Bs)` create two queues `As` and `Bs`, which are merged into `Cs` and served by `serve(Cs)`.

Suppose neither `queue1(As)` nor `queue2(Bs)` has generated a queue of requests, or, both `As` and `Bs` are uninstantiated. The process `merge(As,Bs,Cs)` will attempt to unify `As` with the first argument `[X|Xs]` of M_1, but this attempt is suspended because it would instantiate `As`. Suppose now `queue1(As)` has instantiated `As` to `[john|Rest]`. Then the suspended unification becomes `[john|Rest]=[X|Xs]`, which can now succeed without instantiating `As`. Thus, the guard of M_1 will succeed, and `merge(As,Bs,Cs)` can commit to it. After commitment, the goal `Zs=[X|Us]`, which has now become `Cs=[john|Us]`, will run and the first element of `Cs` will be determined. The remaining body goal of M_1, merge(Rest,Ys,Us), merges the rest of the first queue and the second queue.

If merge(As,Bs,Cs) finds that both As and Bs have been instantiated to non-empty queues, it will commit either to M_1 or to M_2, but not to both.

3. Programming in GHC

Processes play a very important role in GHC programs. "Process" is a synonym of "goal" in GHC. A process, defined using subprocesses, reduces itself into the subprocesses and terminates when all the subprocesses terminate.

For example, four processes, queue1, queue2, merge and serve, are created at the beginning of the execution of the last example:

```
:- queue1(As), queue2(Bs), merge(As,Bs,Cs), serve(Cs).
```

The merge process uses either M_1 or M_2 for each reduction while neither the rest of As nor the rest of Bs is known to be empty. A merge subprocess is created in each reduction, and thus the original merge process will continue to be alive. The original merge process will terminate and be deleted when either M_3 or M_4 is selected.

Using the process creation and deletion capability, it is possible to realise a flexible assembly line which dynamically changes its structure during the execution of the program. We explain a list compaction program which removes duplications as an example showing such behaviour. Let compact(Xs, Ys) be a process which eliminates duplications from the list Xs and returns the result through Ys. The compact process is defined in GHC as follows:

```
C1:  compact([],Ys) :- true | Ys=[].
C2:  compact([X|Xs],Ys) :- true |
          Ys=[X|Ys1], remove(X,Xs,Xs1), compact(Xs1,Ys1).

R1:  remove(X,[],Us) :- true | Us=[].
R2:  remove(X,[X|Xs],Us) :- true | remove(X,Xs,Us).
R3:  remove(X,[X1|Xs],Us) :- X=\=X1 | Us=[X1|Vs], remove(X,Xs,Vs).
```

The execution of a goal clause

```
:- compact([1,3,2,1,3,1,4,2],Ys).
```

first creates a single compact process. Since this process successfully solves the guard of C_2, C_2 is selected and the three processes appearing in the body of C_2 are created. Since the remove process is defined recursively, it will continue to be alive as long as the second argument is not empty. Each time the compact process is reduced using C_2, a new remove process is created, resulting in process proliferation as shown in Fig. 1. Ease of process creation is very important for parallel programs, because processes are

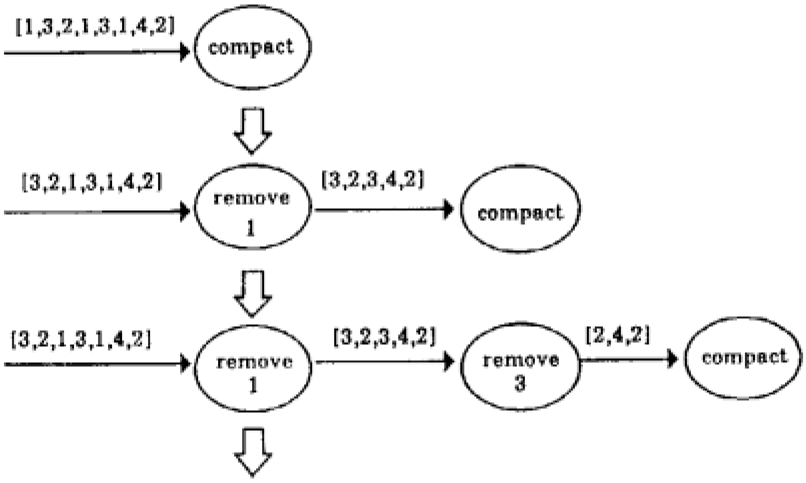

Fig. 1 Process proliferation in the compact program

```
search((Key,Value),nt_node(Key,Value,Left,Right)) :- !.
search((Key,Value),nt_node(Key1,Value1,Left,Right)) :-
        Key<Key1, !, search((Key,Value),Left).
search((Key,Value),nt_node(Key1,Value1,Left,Right)) :-
        Key>Key1, !, search((Key,Value),Right).
search((Key,Value),t_node) :- Value=undefined.

update((Key,Value),nt_node(Key,Value1,Left,Right),
        nt_node(Key,Value,Left,Right)) :- !.
update((Key,Value),nt_node(Key1,Value1,Left,Right),
        nt_node(Key1,Value1,Left1,Right)) :-
        Key<Key1, !, update((Key,Value),Left,Left1).
update((Key,Value),nt_node(Key1,Value1,Left,Right),
        nt_node(Key1,Value1,Left,Right1)) :-
        Key>Key1, !, update((Key,Value),Right,Right1).
update((Key,Value),t_node,nt_node(Key,Value,t_node,t_node)).
```

Fig. 2 Ordered binary tree search program in Prolog

the units of parallel execution and easy process creation facilitates the extraction of parallelism that may vary in the course of computation.

The role of processes in parallel programs corresponds to that of data in sequential programs and process structures to data structures (Shapiro 1984). Let us consider binary tree search programs to compare Prolog and GHC. Fig. 2 shows a Prolog program for searching and updating an ordered binary tree. In the program, ordered binary trees are represented as terms that are passed through the second and third arguments of `search` and `update`. Each tree is either a constant `t_node` representing an empty tree or of the form `nt_node(`*Key*`,` *Value*`,`*Left*`,`*Right*`)`, a structure representing a non-empty tree whose root has a pair of *Key* and *Value* and two subtrees, *Left* and *Right*.

In the GHC program shown in Fig. 3, on the other hand, each node of an ordered binary tree is represented by a process and each link is represented by a variable shared

```
nt_node([],_,_,Left,Right) :- true | Left=[], Right=[].
nt_node([search(Key,Value)|Cs],Key,Value1,Left,Right) :-
        true | Value=Value1,nt_node(Cs,Key,Value1,Left,Right).
nt_node([search(Key,Value)|Cs],Key1,Value1,Left,Right) :-
        Key<Key1 | Left=[search(Key,Value)|Left1],
        nt_node(Cs,Key1,Value1,Left1,Right).
nt_node([search(Key,Value)|Cs],Key1,Value1,Left,Right) :-
        Key>Key1 | Right=[search(Key,Value)|Right1],
        nt_node(Cs,Key1,Value1,Left,Right1).
nt_node([update(Key,Value)|Cs],Key,Value1,Left,Right) :-
        true | nt_node(Cs,Key,Value,Left,Right).
nt_node([update(Key,Value)|Cs],Key1,Value1,Left,Right) :-
        Key<Key1 | Left=[update(Key,Value)|Left1],
        nt_node(Cs,Key1,Value1,Left1,Right).
nt_node([update(Key,Value)|Cs],Key1,Value1,Left,Right) :-
        Key>Key1 | Right=[update(Key,Value)|Right1],
        nt_node(Cs,Key1,Value1,Left,Right1).

t_node([]) :- true | true.
t_node([search(Key,Value)|Cs]) :- true |
        Value=undefined, t_node(Cs).
t_node([update(Key,Value)|Cs]) :- true |
        nt_node(Cs,Key,Value,Left,Right),
        t_node(Left), t_node(Right).
```

Fig. 3 Ordered binary tree search program in GHC

by two node processes. Instead of the data structure `nt_node`(*Key*, *Value*, *Left*, *Right*) appearing in the Prolog program, the GHC program defines a process of the form `nt_node`(*Cs*, *Key*, *Value*, *Left*, *Right*), where *Cs* is the communication variable through which messages come from the parent process, *Key* and *Value* are the internal states of the process, and *Left* and *Right* are the communication variables leading to their son processes. While *Left* and *Right* in the Prolog program are considered as data structures representing subtrees, *Left* and *Right* in the GHC program can be thought of as communication channels for passing commands such as `search`(*Key*, *Value*) and `update`(*Key*, *Value*).

The most significant difference between these two programs lies in their ways of updating. In the Prolog program, each node on the path from the root down to the updated node is copied because destructive assignment is not allowed. The GHC program, on the other hand, does not copy any data structures. Instead, it passes an `update` message along a tree branch to the target process and finally updates the value by changing the internal state of the process.

As explained above, operations on an ordered binary tree in the GHC program are designated by a sequence of commands given to the first arguments of `nt_node` processes and `t_node` processes. Thus, this program is considered to follow the object-oriented programming style.

4. Program Transformation in GHC

It is widely recognised that the program transformation technique provides a powerful, systematic tool for improving programs. Having a set of transformation rules for GHC programs will be useful for deriving efficient parallel programs from straightforward ones. Since GHC inherits many aspects of pure logic programming, one may be tempted to define the set of rules by adapting the unfold/fold rules developed for logic programs (Tamaki and Sato 1984). However, this is not a simple task because logic programming and GHC are quite different in their frameworks. We want to use GHC as a process description language. This means that our rules should preserve the *behaviour* of the processes defined by a program, whereas Tamaki's and Sato's rules were designed so as to preserve the least model semantics. Furthermore, we must be able to handle non-terminating but useful programs.

We have developed a set of transformation rules for Flat GHC programs (Ueda and Furukawa 1988). The set is based on unfolding and folding, and considers the control aspect of the language defined by the rule of synchronisation. It consists of four rules: normalisation, immediate execution, case-splitting, and folding. *Normalisation* executes the unification goals in the guard and the body of a clause as far as possible. The result is a clause with no unification goals in the guard and normalised unification goals in the body. *Immediate execution* unfolds a non-unification body goal g, replacing it by the body goals of a clause to which g can commit. A new clause is created for each clause to which g can commit. Immediate execution is applied only when the set of clauses to which g can commit is known statically; it is not applied if there is a clause to which g cannot immediately commit but some instance $g\theta$ of g can. *Case-splitting* also unfolds a non-unification body goal g, but it can promote the guards of the clauses used for unfolding to the guard of the clause being unfolded. This rule is the most complicated of the four and will be illustrated in the example below. *Folding* is very similar to the folding rule for pure logic programs.

We leave the formal definition of the rules to (Ueda and Furukawa 1988), and illustrate them using an example of process fusion (Furukawa and Ueda 1985). We consider a simple program that computes the sequence of the partial sums of an integer sequence.

```
F1:  integerSums(I,N,Sums) :- true | integers(I,N,Is), sums(Is,Sums).
F2:  integers(I,N,Is) :- I=<N |
          Is=[I|Is1], I1:=I+1, integers(I1,N,Is1).
F3:  integers(I,N,Is) :- I >N | Is=[].
F4:  sums(Is,Sums) :- true | sums1(Is,0,Sums).
F5:  sums1([],     _,Sums) :- true | Sums=[].
F6:  sums1([I|Is1],S,Sums) :- true |
          S1:=I+S, Sums=[S1|Sums1], sums1(Is1,S1,Sums1).
```

The above program uses two tail-recursive processes, `integers` and `sums1`, to compute `Sums`. Our objective is to obtain an equivalent program with a single tail-recursive process. We first execute the second body goal of F_1 so that it has two tail-recursive goals:

F_1
↓ *Immediate Execution*
F_7:
```
integerSums(I,N,Sums) :- true | integers(I,N,Is), sums1(Is,0,Sums).
```

Then we introduce a new clause for the final single process by parameterising the second argument of sums1 in F_7 and leaving Is local. The resulting clause is:

F_8:
```
fused_integerSums(I,N,S,Sums) :- true |
        integers(I,N,Is), sums1(Is,S,Sums).
```

The second argument of sums1 has been generalised to a variable S, and it is included in the clause head. Now we try to obtain a tail-recursive definition of fused_integerSums using case-splitting and folding. First, we split F_8 by case-splitting:

F_8
↓ *Case-splitting*
F_9:
```
fused_integerSums(I,N,S,Sums) :- I=<N |
        Is=[I|Is1], I1:=I+1, integers(I1,N,Is1), sums1(Is,S,Sums).
```
F_{10}:
```
fused_integerSums(I,N,S,Sums) :- I >N | Is=[], sums1(Is,S,Sums).
```

Case-splitting enumerates all the possible ways in which one of the body goals of F_8 commits first. In the case of F_8, it is impossible for sums1(Is,S,Sums) to commit before integers(I,N,Is), because sums1(Is,S,Sums) requires the value of Is, which never comes through the arguments of fused_integerSums. Therefore, F_9 and F_{10}, obtained by unfolding using F_2 and F_3, are the only cases we must consider.

For the time being we leave F_{10} and work on F_9. F_9 can be transformed further, starting from the execution of the unification goal Is=[I|Is1]:

F_9
↓ *Normalisation*
F_{11}:
```
fused_integerSums(I,N,S,Sums) :- I=<N |
        I1:=I+1, integers(I1,N,Is1), sums1([I|Is1],S,Sums).
```
↓ *Immediate execution*
F_{12}:
```
fused_integerSums(I,N,S,Sums) :- I=<N |
        I1:=I+1, integers(I1,N,Is1),
        S1:=I+S, Sums=[S1|Sums1], sums1(Is1,S1,Sums1).
```

Now we can fold integers(I1,N,Is1) and sums1(Is1,S1,Sums1) using F_8:

F_{12}
↓ *Folding by F_8*
F_{13}:
```
fused_integerSums(I,N,S,Sums) :- I=<N |
        I1:=I+1, S1:=I+S, Sums=[S1|Sums1],
        fused_integerSums(I1,N,S1,Sums1).
```

F_{10} can be simplified also:

F_{10}
$\downarrow$ *Normalisation and Immediate execution*
F_{14}: `fused_integerSums(I,N,S,Sums) :- I >N | Sums=[].`

The remaining task is to express the original predicate `integerSums` in terms of the newly introduced predicate `fused_integerSums`:

F_7
$\downarrow$ *Folding by F_8*
F_{15}: `integerSums(I,N,Sums) :- true | fused_integerSums(I,N,0,Sums).`

The resulting clauses, F_{13}, F_{14} and F_{15}, give a new definition of the `integerSums` program. This program has eliminated the intermediate stream `Is` and the operations on it.

5. Formal Semantics

There have been several proposals of the formal semantics of parallel logic programming languages (Saraswat 1987) (Gerth et al. 1988) (Murakami 1988). Here, we briefly introduce a simple semantics of Flat GHC designed for justifying the transformation rules described in Section 4. A complete description of the semantics will be found in (Ueda and Furukawa 1988).

The design criteria of our semantics are as follows:

(1) *Modelling behaviour:* A multiset of GHC goals can be regarded as a process that communicates with the outside world by observing and generating substitutions. The semantics should model this behavioral aspect.

(2) *Abstractness:* The semantics should concentrate on communication. It should abstract internal affairs of a process such as the number of (sub)goals and the number of commitments made. It should also abstract *how* unification is specified in the source text.

(3) *Modelling non-terminating programs:* It must be possible to define the semantics of programs that do not terminate but are still useful.

(4) *Modelling anomalous behaviour:* Anomalous behaviour such as the failure of a unification goal in a clause body, the irreducibility of a non-unification goal and infinite computation without observable substitution must be modelled, because we have to prove that such behaviour is not introduced by program transformation.

(5) *Simplicity and generality:* The semantics should be as simple and general as possible to be widely used. We decided to use standard tools like *finite* terms, substitutions defined over them, and least fixpoints. We decided *not* to use mode systems. We also decided *not* to handle discontinuous concepts like fairness.

(6) *Usefulness:* It should not be just a description; it should be a useful tool at least for proving the correctness of the transformation rules.

The semantics of a multiset B_0 of goals under a program $\mathcal{P}$, denoted $[\![B_0]\!]_{\mathcal{P}}$, is modelled as the set of all possible finite sequences of transactions with it. A *(normal) transaction*, denoted $\langle\alpha,\beta\rangle$, is an act of providing a multiset of goals with a possibly empty *input substitution* α and obtaining an observable (see below) *output substitution* β. An output substitution is also called a *partial answer substitution*.

The first transaction $\langle\alpha_1,\beta_1\rangle$ must be made through the variables in B_0. The above observability condition for β_1 can be written as $B_0\alpha_1\beta_1 \not\equiv B_0\alpha_1$. As a result of the first transaction, B_0 will be reduced to a multiset B_1 of goals, which represents the rest of the computation. Then the second transaction $\langle\alpha_2,\beta_2\rangle$ must be made through the variables in $B_0\alpha_1\beta_1$.

The size of a transaction depends on how the outside world observes an output substitution. Suppose B_0 returns a complex data structure t in response to an input α_1. What should β_1 be, or what should the outside world see in one transaction? The answer is that the outside world can observe any *finite* template of t (i.e., a term of which t is an instance). In our model, the result of one unification goal may be observed using two or more transactions, and the result of two or more unification goals may be observed in one transaction. A transaction is of a finite nature; it is realised by a finite number of reductions and can return only a finite data structure.

The outside world may not communicate with B_0 at all. This is modelled by always including ϵ (empty sequence) in $[\![B_0]\!]_{\mathcal{P}}$. The empty sequence is used as a base case in defining the model of B_0 inductively.

An input α_1 to B_0 may not necessarily cause a normal transaction as defined above. First, it may cause failure of a unification goal in a clause body. This is modelled by letting $[\![B_0]\!]_{\mathcal{P}} \ni \langle\alpha_1,\top\rangle$, where $\top$ means failure. Second, B_0 may succeed (i.e., be reduced out) with no observable output. Third, B_0 may deadlock (i.e., be reduced to a multiset of goals that does not allow further reduction) with no observable output. Fourth, B_0 may fall into infinite computation that generates no observable output. The last three cases mean the *inactivity* of B_0 and cannot be distinguished from outside; so they are all modeled by letting $[\![B_0]\!]_{\mathcal{P}} \ni \langle\alpha_1,\bot\rangle$, where $\bot$ stands for 'no output'. However, if necessary, these cases could be distinguished in the model by using $\bot_{success}$, $\bot_{deadlock}$ and $\bot_{divergence}$ instead of $\bot$. Failure and inactivity are called *special transactions* and are used as base cases in defining the model of B_0.

Consider a single clause program

$\mathcal{P}$: `p(X) :- true | X=f(Y), p(Y).`

and autonomous (i.e., empty input) transactions with $\mathcal{P}$. Then $[\![\texttt{p(X)}]\!]_{\mathcal{P}}$ has

ϵ,
$\langle\emptyset,\{\texttt{X}\leftarrow\texttt{f(X1)}\}\rangle$,
$\langle\emptyset,\{\texttt{X}\leftarrow\texttt{f(X1)}\}\rangle\langle\emptyset,\{\texttt{X1}\leftarrow\texttt{f(X2)}\}\rangle$,
$\langle\emptyset,\{\texttt{X}\leftarrow\texttt{f(X1)}\}\rangle\langle\emptyset,\{\texttt{X1}\leftarrow\texttt{f(X2)}\}\rangle\langle\emptyset,\{\texttt{X2}\leftarrow\texttt{f(X3)}\}\rangle$,
...

and also

$\langle \emptyset, \{$`X`$\leftarrow$`f(f(X2))`$\}\rangle$,
$\langle \emptyset, \{$`X`$\leftarrow$`f(f(f(X3)))`$\}\rangle$,
... .

$[\![$`p(X)`$]\!]_{\mathcal{P}}$ has $\langle \emptyset, \perp \rangle$ also, because the semantics allows unfair execution in favour of the recursive goal `p(Y)`.

Our model successfully circumvents the Brock-Ackerman anomaly (Brock and Ackerman 1981). Let $\mathcal{BA}$ be:

```
d([A|_],O) :- true | O=[A,A].

merge([A|X1],Y,Z) :- true | Z=[A|Z1], merge(X1,Y,Z1).
merge(X,[A|Y1],Z) :- true | Z=[A|Z1], merge(X,Y1,Z1).
merge([],Y,Z) :- true | Z=Y.
merge(X,[],Z) :- true | Z=X.

p1([A|Z1],O) :- true | O=[A|O1],p11(Z1,O1).
p11([B|_],O1) :- true | O1=[B].

p2([A,B|_],O) :- true | O=[A,B].

g1(I,J,O) :- true | d(I,X), d(J,Y), merge(X,Y,Z), p1(Z,O).
g2(I,J,O) :- true | d(I,X), d(J,Y), merge(X,Y,Z), p2(Z,O).
```

Then, the computation

$\langle \{$`I`$\leftarrow$`[5|_]`$\}, \{$`O`$\leftarrow$`[5|O']`$\}\rangle$

belongs both to $[\![$`g1(I,J,O)`$]\!]_{\mathcal{BA}}$ and to $[\![$`g2(I,J,O)`$]\!]_{\mathcal{BA}}$ (O' being a fresh variable), but

$\langle \{$`I`$\leftarrow$`[5|_]`$\}, \{$`O`$\leftarrow$`[5|O']`$\}\rangle\langle \{$`J`$\leftarrow$`[6|_]`$\}, \{$`O'`$\leftarrow$`[6]`$\}\rangle$

belongs only to $[\![$`g1(I,J,O)`$]\!]_{\mathcal{BA}}$ and not to $[\![$`g2(I,J,O)`$]\!]_{\mathcal{BA}}$.

6. Conclusion

This paper presented a parallel logic programming language GHC. It showed that GHC is a genuine parallel programming language and hence encourages programmers to write parallel programs. The paper also described transformation rules for GHC programs which will help to optimise them. To prove the correctness of the transformation rules, we introduced a simple formal semantics of Flat GHC programs which allows non-terminating computations.

In the Fifth Generation Computer Project, we are developing experimental parallel hardware for FGHC. We are developing two systems in parallel. One is a multi-processor system, called the Multi-PSI, composed of 64 Personal Sequential Inference machines (PSIs). Each PSI enables fast execution of FGHC programs (around 100 KLIPS) by firmware support of WAM-like instructions for FGHC. The main purpose of the system is to provide software researchers with a stable tool for developing software systems, including the operating system for the Multi-PSI itself. Currently, the hardware of the system is completed and its system software is under development. It is planned to be completed by the end of this fiscal year.

The other system is a VLSI-based parallel processor called the Parallel Inference Machine (PIM) which is expected to be our final target. We are planning to connect about 1000 processing elements (PEs) in the final stage. Before jumping to such a large scale, we are now developing a smaller scale prototype consisting of around 100 PEs. It has a hybrid architecture of shared memory and distributed memory. About ten PEs are connected tightly to compose a cluster of a shared memory architecture. These clusters are then connected together via a network, resulting in a distributed memory architecture. Currently, we are concentrating on the development of a single cluster. The prototype will be completed by 1989.

Much research is required to make our parallel computers truly useful. First, we need to enhance the expressive power of GHC. There have been several significant achievements in increasing the expressive power of Prolog. The introduction of constraints in Prolog and efficient algorithms for searching recursive databases are the most important. To realise the same extended functionalities in GHC has turned out to be quite difficult due to the lack of a backtracking capability. We need to realise Prolog variables in terms of GHC data structures. However, this method is expected to cause a slowdown of one order of magnitude, which we want to avoid.

Second, we need to develop parallel programming technologies for extracting maximum parallelism. There are several research subjects. The first is to develop new programming paradigms appropriate for formulating various application problems. The second is to solve the load balancing problem in the execution of programs on an actual parallel computer. The third is to develop a computation model reflecting the characteristics of real parallel processors such as the non-homogeneous distances among PEs, and to develop a useful measure of the complexity of parallel algorithms.

Acknowledgments

We wish to express our thanks to Kazuhiro Fuchi, Director of ICOT Research Center, who provided us with the opportunity to pursue this research. We would also like to thank Ryuzo Hasegawa, Chief of ICOT First Research Laboratory, and its members who contributed a lot to the research reported here.

References

Brock, J. D. and Ackerman, W. B. (1981) Scenarios: A Model of Non-determinate Computation. In *Formalization of Programming Concepts*, LNCS 107, Springer-Verlag, pp. 252–259.

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. (1984) *PARLOG: Parallel Programming in Logic*. Research Report DOC 84/4, Dept. of Computing, Imperial College of Science and Technology, London. Also in *ACM. Trans. Prog. Lang. Syst.*, Vol. 8, No. 1 (1986), pp. 1–49.

Dijkstra, E. W. (1975) Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Comm. ACM*, Vol. 18, No. 8, pp. 453–457.

Furukawa, K. and Ueda, K. (1985) GHC Process Fusion by Program Transformation. In *Second Conf. Proc. Japan Soc. Softw. Sc. Tech.*, pp. 89–92.

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

Murakami, M. (1988) A Declarative Semantics of Parallel Logic Programs with Perpetual Processes. To be presented at the Int. Conf. on Fifth Generation Computer Systems 1988, Tokyo.

Saraswat, V. J. (1987) GHC: Operational Semantics, Problems and Relationship with CP(↓,|). In *Proc. 1987 Symposium on Logic Programming*. IEEE Computer Society Press, pp. 347–358.

Shapiro, E. Y. (1983) *A Subset of Concurrent Prolog and Its Interpreter*. Tech. Report TR-003, Institute for New Generation Computer Technology, Tokyo.

Shapiro, E. Y. (1984) Systolic Programming: A Paradigm of Parallel Processing. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, ICOT, Tokyo, pp. 458–470.

Tamaki, H. and Sato, T. (1984) Unfold/Fold Transformation of Logic Programs. In *Proc. Second Int. Logic Programming Conf.*, Uppsala Univ., Sweden, pp. 127–138.

Ueda, K. (1985) Guarded Horn Clauses. ICOT Tech. Report TR-103, ICOT, Tokyo (revised in 1986). Revised version in *Proc. Logic Programming '85*, Wada, E. (ed.), LNCS 221, Springer-Verlag, 1986, pp. 168–179.

Ueda, K. and Furukawa, K. (1988) Transformation Rules for GHC Programs. To be presented at the Int. Conf. on Fifth Generation Computer Systems 1988, Tokyo.