FIFTH GENERATION COMPUTER PROJECT: CURRENT RESEARCH ACTIVITY AND FUTURE PLANS

Koichi Furukawa

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

1. Introduction

The FGCS project, aiming at developing a new generation of general purpose computers, began in 1982 and is now almost half way through the ten years of the project. The project was established to explore a qualitatively new approach to computer science in order to solve many difficult problems which have accumulated in the field. We considered that the two single most important and difficult problems are opening up a new market for computers to keep the computer industry growing, and overcoming the low productivity of software compared with hardware. We spent three years developing the target of a new broad market for next generation computers, and concluded that knowledge information processing will be the answer.

At the same time, we attempted to design a computer system adequate for the new application area, i.e. knowledge information processing, and produced the rough sketch of the system shown in Fig. 1. The system has two significant features: one is a highly parallel architecture deviating from the traditional von Neumann architecture, and the other is the

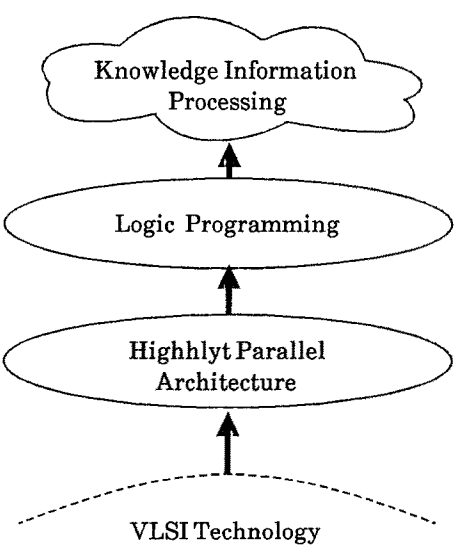


Fig.1 The role of logic programming in the FGCS project

bridge of logic programming to span the chasm between knowledge information processing and the parallel architecture.

Logic programming has been functioning as a strong guideline and as a vehicle to promote research in the project. As a result of our research so far, we have succeeded in developing a more detailed framework providing a clearer perspective on how the bridge must be constructed, the chasm spanned. It consists of a set of programming languages layered in a hierarchy and a set of program transformation methods forming the links between them.

We have designed two new logic programming languages. Guarded Horn Clauses (GHC) is an abstract machine language for parallel execution, and Complex Indeterminate Language (CIL) is an extended version of Prolog suited for writing knowledge information processing application programs. We have also worked up meta programming techniques for introducing user languages by defining their interpreters along with a general method for compiling programs written in those new languages into Prolog. The method is based on the idea of applying partial evaluation to meta programs.

Program transformation is the key to achieving both expressiveness and efficiency of programming languages. One of the new program transformation methods we developed is designed to handle the transformation of some kinds of CIL programs into normal Prolog programs, and the other transforms normal Prolog programs into GHC programs. By successively applying the transformation methods, the CIL programs can be transformed into GHC programs. These are the tentative steps we have taken so far toward establishing the link between knowledge programming and parallel architecture.

In Section 2 an informal explanation of GHC will be given using examples. Meta programming techniques and their optimization by partial evaluation will be presented in Section 3. Two program transformation methods will be described in Section 4, followed by the conclusion in Section 5.

2. Guarded Horn Clauses

GHC is a logic programming language for concurrent programming and parallel execution. It is a successor of Relational Language [Clark 81], Concurrent Prolog [Shapiro 83a] and PARLOG [Clark 84]. Another candidate for a parallel logic language is Horn logic itself, i.e. a language consisting of only pure Horn Clauses (PHC, for short). PHC has, however, serious drawbacks both in software and hardware. The software problem is that you cannot describe parallel phenomena explicitly in PHC, and the hardware problem is that you have to maintain different environments for each or-branch of computations, for which no efficient method has been discovered so far.

On the other hand, concurrent logic languages do not have these two problems (to be precise, Concurrent Prolog suffers from the multiple environment problem). This favorable characteristic derives from the restrict ion imposed on PHC to obtain "guarded" Horn clauses. Namely the restriction makes the underlining execution mechanism of concurrent logic languages much simpler than that of PHC. The only apparent drawback of concurrent logic languages compared with PHC is that they loose the capability to find all solutions satisfying given conditions based on don't know

nondeterminism. But we succeeded in removing this drawback by program transformation [Ueda 86]. The details will be discussed below in Section 4.

GHC is a language consisting of a set of guarded Horn clauses, as the name suggests. We use a vertical bar "|" called commit operator to designate the guard part of each clause: the guard part is to the left of the commit, and the body part is on the right. The guard part specifies the condition for the body to be selected for successive computation. The important features of GHC are the following suspension rules [Ueda 85]:

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

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

Rule (a) determines synchronization and rule (b) execution of bodies. Let us take a simple example of service at a counter with two queues. We need to merge these two queues into one in order to make a single queue for service. Let us define a procedure `merge(Xs,Ys,Zs)`, in which two queues `Xs` and `Ys` will be merged into a single queue `Zs`, in terms of GHC:

```
(m1) merge([X|Xs],Ys,Zs) :-  true  | Zs=[X|Us],merge(Xs,Ys,Us).
(m2) merge(Xs,[Y|Ys],Zs) :-  true  | Zs=[Y|Us],merge(Xs,Ys,Us).
(m3) merge([],Ys,Zs) :-  true  | Zs=Ys.
(m4) merge(Xs,[],Zs) :-  true  | Zs=Xs.
```

The fact that the clause (m1) waits for people arriving at the first queue is represented by its first argument `[X|Xs]`. The usual situation using the merge program is something like

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

where `queue1(As)` and `queue2(Bs)` are processes generating a sequence of people joining the queues. Suppose that the processes `queue1` and `queue2` do not generate any instances. Then, `As` (and `Bs` also) will remain as a variable and the execution of the merge process will be suspended due to the necessary unification `As=[X|Xs]` and the rule of suspension. What this mean is since the expression `[X|Xs]` asserts that it has at least one element, the unification will force the conclusion that the same fact is true for `As`, and this means the unification would give a binding to the variable appearing in the caller. When `queue1` instantiates the variable `As` to, say, `[john|Rest]`, the above unification will become `[john|Rest]=[X|Xs]` and this can be solved without giving any new binding to variables in the caller. Therefore, the guard part of the clause (m1) can be successfully solved and it can be committed for successive computation. The body of (m1) consists of a unification which will give a binding to the variable `Zs` representing the merged queue, and a recursive call to the merge process. When there are people in both queues, clauses (m1) and (m2) can both solve the guard parts. In such a case, one of the clauses is chosen nondeterministically for successive computation.

It is very easy to represent assembly line-like parallel processing in GHC. In an assembly line program, a shared variable between processes will represent a sequence of unfinished products. Programming in GHC corresponds to designing such an assembly

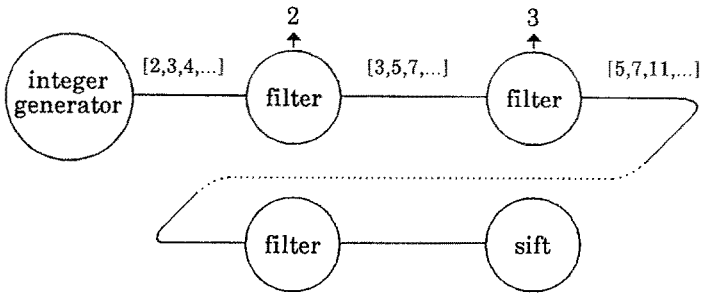


Fig.2 The process structure of a prime generation program by Aristotle's sieve

line, and its execution corresponds to building and operating it to manufacture products. Since new processes are created during execution of the GHC program, it is possible to modify the assembly line while operating it. A prime generation program by Aristotle's sieve is an example of dynamic creation of processes. The system consists of an integer generator, a sequence of dynamically created filter processes corresponding to each prime to remove multiples of the prime, and a sift process to continue creation of filter processes as shown in Fig. 2.

Shared variables among processes can be regarded as communication channels and the entire channels form a communication network defining a process structure. It is expected that the process structure will reflect the target problem's structure: in other words, in parallel programs, the process structure will play the role of a data structure in sequential programs. Furthermore, since each process is an active object, the entire process structure's function is much richer and stronger than that of a data structure.

Let us consider the problem of finding all possible paths from some point to a fixed goal. A typical programming technique in writing a Prolog or LISP program is to use a stack or queue to maintain all partial paths traversed (we do not consider using backtrack in Prolog here). The program is shown in Fig. 3. The first argument of the predicate "paths" is the stack. The same problem is easily written in GHC, as shown in Fig. 4, where no stack appears explicitly. A process structure corresponding to the control stack is created dynamically as shown in Fig. 5. Note that the structure is no more a stack than a collection of possible partial paths. Also, the GHC program is simpler and easier to understand. This comes from the fact that you can capture the program in an object-oriented way; that is, a process represents an object.

3. Knowledge Programming Methodology

A knowledge programming system is a system for building various kinds of knowledge based application systems and requires many programming paradigms such as object-oriented programming, meta programming, constraint programming, and so on to describe a wide range of application systems.

Concurrent logic languages are known to be well suited to realizing object-oriented programming [Shapiro 83b]. Several research efforts are underway aimed at designing an

```
start_paths(Start,Goal,Paths) :- paths([[Start]],Goal,Paths).

paths([],_,[]) :- !.
paths([First|Rest],Goal,[Path|Paths]) :-
        First = [Goal|_],
       reverse(First,Path),
       paths(Rest,Goal,Paths).
paths([First|Rest],Goal,Paths) :-
        expand(First,Add),
       append(Add,Rest,Next),
       paths(Next,Goal,Paths).

expand([Last|Rest],Ts) :-
        neighbors(Last,Nodes),
       removeIf(Nodes,[Last|Rest],Ts).

removeIf([],_,Z) :- !, Z=[].
removeIf([N|Ns],Path,Ts) :-
       member(N,Path),!,removeIf(Ns,Path,Ts).
removeIf([N|Ns],Path,Ts) :-
        Ts=[[N|Path]|Ts1],removeIf(Ns,Path,Ts1).

member(X,[X|_]) :- !.
member(X,[Z|Y]) :- member(X,Y).

append([],X,X).
append([U|X],Y,[U|Z]):-append(X,Y,Z).

reverse(X,Y) :- appendReverse(X,[],Y).

appendReverse([],X,X).
appendReverse([U|X],V,Y) :- appendReverse(X,[U|V],Y).

neighbors(start,Z) :- Z=[a,d].
neighbors(a,    Z) :- Z=[start,b].
neighbors(b,    Z) :- Z=[a,c,goal].
neighbors(c,    Z) :- Z=[b,d,goal].
neighbors(d,    Z) :- Z=[start,c,e].
neighbors(e,    Z) :- Z=[d,goal].
neighbors(goal, Z) :- Z=[b,c,e].
```

Fig.3 A path finding program using stack in Prolog

object-oriented programming language or system based on concurrent logic languages [Furukawa 84], [Kahn 86]. But they are not well developed yet, and we need further research to obtain significant results.

```
start_paths(Start,Paths) :- true | pathFinder(Start,[],Paths).

pathFinder(goal,History,Paths) :- true | Paths=[[goal|History]].
pathFinder(Node,History,Paths) :- Node\=goal |
        (Node -> Nexts), pathFinder1(Node,Nexts,History,Paths).

pathFinder1(Node,[],History,Paths) :- true | Paths=[].
pathFinder1(Node,[N|Ns],History,Paths) :- true |
        member(N,History,Result),
        childPathFinder(Result,N,[Node|History],P1),
        pathFinder1(Node,Ns,History,P2),
        merge(P1,P2,Paths).

childPathFinder(true,_,_,Paths) :- true | Paths=[].
childPathFinder(false,Node,History,Paths) :- true |
        pathFinder(Node,History,Paths).

member(_,[],R) :- true | R=false.
member(X,[X|_],R) :- true | R=true.
member(X,[A|Y],R) :- X\=A | member(X,Y,R).

merge([X|Xs],Ys,Zs) :- true | Zs=[X|Us],merge(Xs,Ys,Us).
merge(Xs,[Y|Ys],Zs) :- true | Zs=[Y|Us],merge(Xs,Ys,Us).
merge([],Ys,Zs) :- true | Ys=Zs.
merge(Xs,[],Zs) :- true | Xs=Zs.

start->Next :- true | Next=[a,d].
a->Next :- true | Next=[start,b].
b->Next :- true | Next=[a,c,goal].
c->Next :- true | Next=[b,d,goal].
d->Next :- true | Next=[start,c,e].
e->Next :- true | Next=[d,goal].
```

Fig. 4 A path finding program in GHC

In this section, we concentrate rather on the topics of meta programming and constraint programming.

3.1 Meta Programming

Although Prolog can deduce goals from rules and facts, its built-in fixed control structure (i.e., the top down and left-to-right strategy) has been the object of considerable criticism. Recently a methodology called meta programming has emerged and its usefulness has been shown in realizing various inference systems with different control structures than the built-in one[Sterling 84]. Examples are Production System, a bottom-up parser, a parallel parser, a symbolic computation system, and a backward reasoning system with certainty factor handling.

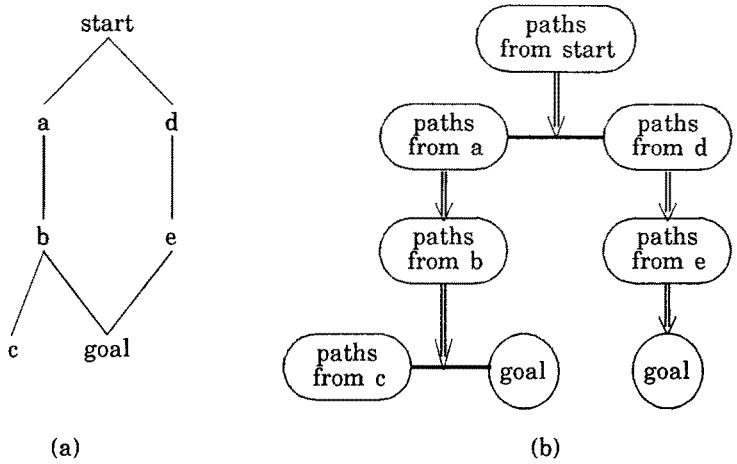


Fig. 5 A map and its corresponding process structure

These examples show that the addition of meta programming features can make Prolog into a good knowledge programming language. However, meta programming has a serious drawback concerning efficiency, because of its interpretive mode of execution. To remedy this defect, we developed a very powerful compiling method based on partial evaluation [Futamura 71,83], [Takeuchi 86].

Fig. 6 shows a simple example of rule compilation by partial execution in Fig. 6. Fig. 6 (b) is a set of rules of which the inference engine is given in Fig. 6 (a). The result obtained by partial execution of the inference engine together with the given set of rules is shown in Fig. 6 (c).

For intuitive understanding of the above transformation process, we try "symbolical" execution of the goal

```
?- solve(should_take(A,aspirin),[B]).        (1)
```

Note that this execution is symbolic because we do not have sufficient information for actual execution. By applying the fourth definition of "solve" clauses to the goal (1), we obtain the following sequence of goals:

```
?- rule(should_take(A,aspirin),B,F),
   solve(B,S), cf(F,S,B).                    (2)
```

Next, we apply the first definition of the "rule" clauses to the first goal of (2) and execute the body of the clause. Then, we find the values of B and F and obtain a new sequence of goals:

```
?- solve((complains_of(A,Symptom),
          suppresses(aspirin,Symptom),
```

```
solve(true  ,[100]).
solve((A,B) ,Z    ) :- solve(A,X), solve(B,Y), append(X,Y,Z).
solve(not(A),[CF] ) :- solve(A,[C]), C < 20, CF is 100-C.
solve(A     ,[CF] ) :- rule(A,B,F), solve(B,S), cf(F,S,CF).

cf(X,Y,Z) :- product(Y,100,YY),Z is (X*YY)/100.

product([],A,A).
product([X|Y],A,XX) :- B is X*A/100, product(Y,B,XX).

rule(A,B,F) :- ((A:-B)<>F).
rule(A,true,F) :- (A<>F).
```

(a) An inference engine with certainty factor handling

```
should_take(Person,Drug) :-
        complains_of(Person,Symptom),
        suppresses(Drug,Symptom),
        not(unsuitable(Drug,Person)) <> 70.

suppresses(aspirin,pain) <> 60.
suppresses(lomotil,diarrhoea) <> 65.

unsuitable(Drug,Person) :-
        aggravates(Drug,Condition),
        suffers_from(Person,Condition) <> 80.

aggravates(aspirin,peptic_ulcer) <> 70.
aggravates(lomotil,impaired_liver_function) <> 70.
```

(b) A set of rules with certainty factor

```
solve(should_take(A,aspirin),[B]) :-
        solve(complains_of(A,pain),C),
        solve(suffers_from(A,peptic_ulcer),D),
        cf(80,[70|D],E),E<20,F is 100-E,
        append(C,[60,F],G),cf(70,G,B).
solve(should_take(A,lomotil),[B]) :-
        solve(complains_of(A,diarrhoea),C),
        solve(suffers_from(A,impaired_liver_function),D),
        cf(80,[70|D],E),E<20,F is 100-E,
        append(C,[65,F],G),cf(70,G,B).
```

(c) A result of partial execution of the inference engine (a) together with a set of rules (b)

Fig. 6 An example of rule compilation by partial execution

```
        not(unsuitable(Drug,A))),S),          (3)
    cf(70,S,B).
```

The next execution step is the application of the second definition of the "solve" clauses, which results in the following goal sequence:

```
?- solve(complains_of(A,Symptom),S1),
   solve((suppresses(aspirin,Symptom),
          not(unsuitable(aspirin,A))),S2),    (4)
    append(S1,S2,S),        cf(70,S,B).
```

The normal Prolog processor would now proceed to the execution of the first goal of (4). But in this case we cannot execute the goal "solve(complains_of(A,Symptom))" because this goal is only solved by the input from the user when the system is actually used. Therefore, symbolic execution proceeds to execute the next goal and produces the goals:

```
?- solve(complains_of(A,Symptom),S1),
   solve(suppresses(aspirin,Symptom),S21),
   solve(not(unsuitable(aspirin,A)),S22),     (5)
   append(S21,S22,S2),
   append(S1,S2,S),cf(70,S,B).
```

Now, we try to execute the second goal "solve(suppresses(...))" of (5). Note that this goal can be completely solved by applying the first definition of the suppresses clauses, which unifies the variable "Symptom" to the constant "pain" and "S21" to 70. The resulting goal sequence is as follows:

```
?- solve(complains_of(A,pain),S1),
   solve(not(unsuitable(aspirin,A)),S22),
    append(60,S22,S2),                        (6)
   append(S1,S2,S),cf(70,S,B).
```

Note that the variable Symptom in the first predicate "solve(complains_of(A,Symptom),S1)" is also instantiated. Continuing the symbolic execution process, we finally obtain the result given in Fig. 6 (c).

We succeeded in speeding up the original interpretive program by about three times for the above example. We also examined another example of an algebraic manipulation system which was five times faster. These numbers suggest that partial evaluation is a very promising technique for optimization of meta programs. Another merit of the approach is that meta programming enforces a clear separation of object knowledge from control, which makes the programs far easier to understand than the mixed approach. Goebel et al. [Goebel 86] described a MYCIN-like diagnosis system in terms of the combination of a meta interpreter and a set of logical formulas connecting malfunctions and symptoms. In his system the connection is represented in such a way that *"If X has a malfunction A, then it has a set of symptoms B"*, instead of *"If X has a set of symptoms B, then you can conclude that X has a malfunction A"*, which is the representation style in MYCIN. The behaviour of a meta interpreter is described by something like *"If you want to identify that X has some malfunction A, then you need to show that X has all symptoms that A gives rise to."* By partially executing the meta interpreter together with logical formulas

describing the relationship between malfunctions and symptoms, we managed to obtain a set of "compiled" rules which are similar to those in MYCIN.

3.2 Constraint Programming

Constraint Programming is another important paradigm in knowledge programming. There have been many efforts to incorporate it into the logic programming framework [Colmerauer 82,86], [Mukai 85], [Dincbus 86], [Jaffar 86]. They are roughly divided into two groups: one is Colmerauer's and Mukai's work dealing with only passive constraints based on a freeze mechanism, and the other is Dincbus's and Lassez's work dealing with not only passive constraints but also active constraints.

We developed a language called CIL (Complex Indeterminate Language), a version of Prolog with passive constraints, and with the addition of the "complex indeterminate" concept from situation semantics. It possesses not only a freeze mechanism, but also "indefinite terms" to represent frame structures. Indefinite terms are used to represent semantic structures in the experimental discourse understanding system called DUALS [Mukai 85], [Yokoi 86].

Besides the natural language understanding system, we tried to use CIL to build a VLSI CAD system for solving layout problems and it was shown that constraint programming is useful for representing building block layout problems.

4. Program Transformation

The layers of logic languages we have described, that is, the concurrent logic languages like GHC and knowledge programming logic languages such as CIL are good stepping stones toward dividing the original problem of bridging the gap between knowledge information processing and highly parallel computer architecture into three smaller problems: 1. Developing knowledge information processing application systems in knowledge programming languages such as CIL; 2. Transforming programs in knowledge programming logic languages (such as CIL) into those in GHC; and 3. Developing an efficient highly parallel computer on which programs in GHC run very fast.

Here, we focus on the second problem. The general compilation method of meta programs described in Section 2 is regarded as one of the very promising techniques to partial solution of the problem. It transforms two layer programs consisting of a meta interpreter and object programs into single layer programs.

In this section, we present two other program transformation methods. One is for removing freeze from passive constraints programs, and the other is for removing backtrack. The latter method is applied to transform Prolog programs into GHC programs.

4.1 Program transformation for removing freeze

We developed several methods to transform programs with passive constraints into those without them using unfold/fold transformation method [Burstall 77].

Let us explain one of the ideas of the transformation methods briefly. We consider a problem of finding all paths between two points in a given map, possibly including cycles.

Fig. 7 shows an elegant program using passive constraints to check cycles in possible solutions to avoid wasteful search efforts. The reason why we need constraints is that the check goal must be in front of the path generation goal to protect the program from falling into an infinite loop.

The constraints are a set of inequalities for all pairs of nodes in each possible path. We need the freeze mechanism to delay the evaluation of each of these inequalities appearing as constraints until both of the variables in it are instantiated.

By shuffling the set of inequalities and rearranging properly between other goals, it becomes possible to evaluate all inequalities immediately after they are called. We found a way of shuffling analogous to the exchange of the summation order by structural commutativity such as

$$\sum_{j=1}^{n}\sum_{i=j+1}^{n} X_{ij} = \sum_{i=2}^{n}\sum_{j=1}^{i-1} X_{ij} \qquad \text{[Seki 86].}$$

4.2 Program Transformation for removing backtrack

To remedy an apparent drawback of GHC compared with Prolog, Ueda [Ueda 86] proposed a transformation method from "all solutions search" Prolog programs to

```
good_path(X,Y,Path) :-
        good_list(Path), path(X,Y,Path).

good_list(L) :- freeze(L,good_list1(L)).

good_list1([]). good_list1([X|L]) :-
         out_of(X,L), good_list(L).
out_of(X,L) :- freeze(L,out_of1(X,L)). out_of1(X,[]).
out_of1(X,[Y|L]) :- dif(X,Y), out_of(X,L).

path(X,X,[X]).
path(X,Y,[X|Path]) :-
        neighbor(X,Z), path(Z,Y,Path).

neighbor(X,Y) :- neighbor1(X,Y).
neighbor(X,Y) :- neighbor1(Y,X).

neighbor1(a,b).
neighbor1(a,c).
neighbor1(b,d).
neighbor1(b,c).
neighbor1(c,e).
neighbor1(d,e).
```

Fig. 7 An elegant path finding program using passive constraint in Prolog II

equivalent GHC programs. His method has two phases: the first phase involves performing of mode analysis to determine, for each variable, whether it is an input or an output variable, and, given the analyzed program, the transformation is performed in the second.

We will briefly explain the method using the following example:

```
(g0)   ?- append(U,V,[1,2,3]).
(a1)   append([],Z,Z).
(a2)   append([A|X],Y,[A|Z]) :- append(X,Y,Z).
```

When execution starts, we first obtain a partial answer `[1|X]` from (a2), and, after the successive recursions and backtracks, we get three values for `X`; namely `X=[]`, `X=[2]` and `X=[2,3]`.

The problem here is that we need to copy the structure `[1|X]` for all three different computations of X, which would cause a serious problem in an actual parallel execution environment due to the undecidability of the "var" check included in the copy operation [Ueda 86].

To avoid the copy problem, we rewrite the clause (a2) into the following:

```
(a2)'   append(X1,Y,[A|Z]) :- append(X,Y,Z),X1=[A|X].
```

The recursive call in (a2)' does not require any copy operations since the variable X1 does not change during the recursion. But in this case we need an extra job after the recursion: namely, to cons A to X to obtain X1. The and-or graph for the goal (g0) is shown if Fig. 8.

Now let us design an appropriate process structure of GHC to compute all possible solutions of (X,Y). By mapping or-branches in the original Prolog program to and-forks in the target GHC program, we obtain a process structure and its corresponding recursive program schema in GHC as shown in Fig. 9.

Let us consider the behaviour of a particular process, say, the process P3 in the Fig. 9. Since this process corresponds to the computation following the path (a2)' (a2)' (a1), we obtain the following equations:

```
(g1)   ?- X3=[],Y=[3],X2=[2|X3],X=[1|X2].
```

as a result of unfolding the original goal according to the given path. These equations for building solutions constitute the process Q3 in Fig. 9. To realize this final computation, necessary information such as "1", "2", and "3" is stacked during the P's recursions, and it is used in building the solution in the "cont" program. The name "cont" comes from the fact that the final computation (g1) can be interpreted as a collection of continuation tasks after the recursions in the clause (a2)'.

The transformed program is shown in Fig. 10, where ap corresponds to P, ap1 to Q and ap2 to P', respectively. Note that the computation for building solutions such as (g1) is included in ap1 as a call to the "cont" program.

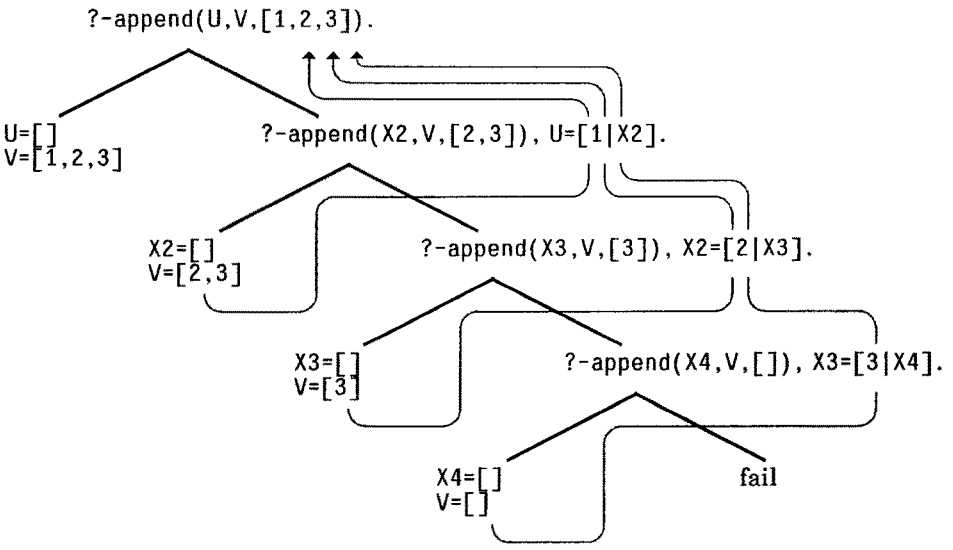
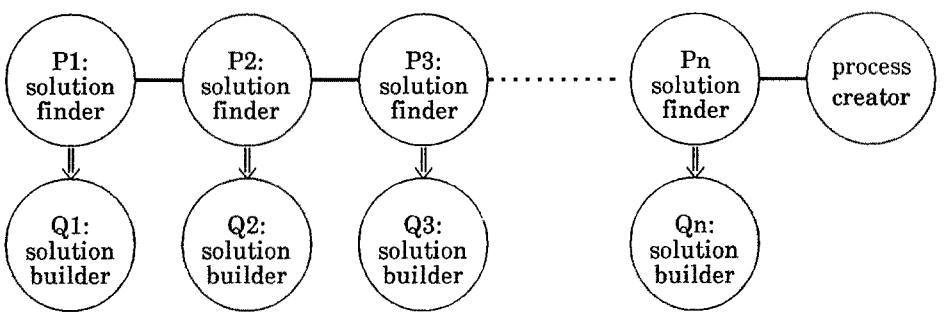


Fig. 8 An and-or tree of the goal `?-append(U,V,[1,2,3])`



(a) A process structure of the list decomposition program in GHC

```
<process_creator> :- <termination-condition> | <finalizer>.
<process_creator> :- otherwize |
       <solution_finder>,<process_creator>.

<solution_finder> :- true | <solution_builder>.
```

(b) A program schema for the process structure (a).

Fig. 9 GHC process structure and its program schema for the list decomposition problem.

```
:-..., ap(Z, 'LO', S, []), ...

ap(Z, Cont, SO, S2) :-true |
    ap1(Z, Cont, SO, S1), ap2(Z, Cont, S1, S2).
ap1(Z, Cont, SO, S1) :- true  |
    cont(Cont, [], Z, SO, S1).
ap2([A|Z], Cont, SO, S1) :- true |
    ap(Z, 'L1'(A,Cont), SO, S1).
ap2(Z,     _,    SO, S1) :- otherwise | SO=S1.

cont('L1'(A,Cont), X, Y, SO, S1) :- true |
    cont(Cont, [A|X], Y, SO, S1).
cont('LO',         X, Y, SO, S1) :- true |
    SO=[(X,Y)|S1].
```

Fig. 10 List Decomposition Program in GHC

We sketched Ueda's method to transform "all solutions search" Prolog programs to equivalent GHC programs. Although this method has a certain limitation on input programs, it is known that it covers a wide range of application programs.

5. Conclusion

We set out in the Fifth Generation Computer project to develop a highly parallel computer with logic programming as the centerpiece. We have already designed several language levels, starting from high level programming languages which handle large applications, down to the machine language that can be executed in a highly parallel manner. We also aim to develop unified program transformation techniques to fill the gaps between these language levels.

We are confident that we have produced the first approximations to solutions of this problem. Fig. 11 is the picture of the levels of programming languages we have obtained through our activities so far.

We can now see the basic outline of the path from applications to parallel execution, although the path has not taken concrete shape yet. Application programs are written in a powerful knowledge programming language equipped with meta programming facilities. These are transformed to more efficient programs written in a general purpose user language. Then these programs are further translated into the more primitive parallel logic programming language FGHC, which can be executed by Multi-PSI and/or PIM.

What remains is to give this path concrete shape. Our approach is to isolate the component problems and solve them one by one and put the results together to achieve the target.

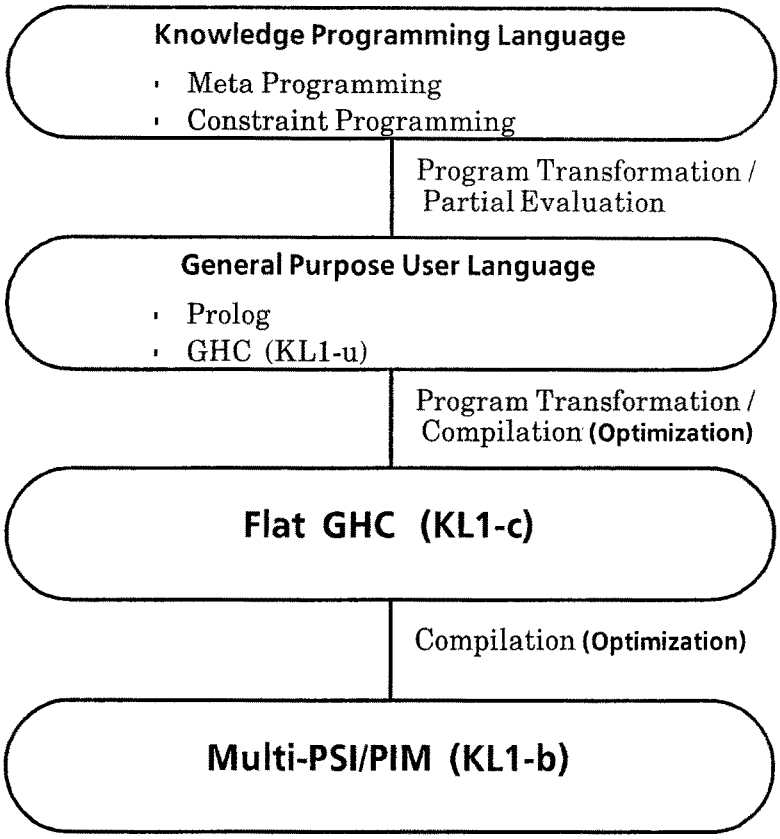


Fig. 11 Levels of programming languages and transformation methods among them

References

[Burstall 77] R. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs," JACM Vol. 24,No. 1, 1977.

[Clark 84] K. Clark and S. Gregory, "PARLOG: Parallel Programming in Logic," Research Report DOC 84/4, Imperial College, April, 1984.

[Colmerauer 82] A. Colmerauer, "Prolog and Infinite Trees," In Logic Programming, K. L. Clark and S. A. Tarnlund (eds.), Academic Press, 1982.

[Colmerauer 86] A. Colmerauer, "Theoretical Model of Prolog II," in Logic Programming and Its Applications, M. Van Caneghem and D. H. D. Warren (eds.), Albex Publishing Corp., 1986.

[Dincbus 86] M. Dincbus and P. Vanhentenryck, "Constraints and Logic Programming," Technical Report TR-LP-9, ECRC, Munich, 1986.

[Furukawa 84] K. Furukawa, A. Takeuchi, S. Kunifuji, H. Yasukawa, M. Ohki, K. Ueda, "Mandala: A Logic Based Knowledge Programming System," Proc. of the FGCS'84, 1984.

[Futamura 71] Y. Futamura, "Partial Evaluation of Computation Process: An Approach to a Compiler-Compiler," Systems, Computers,Controls 2, 1971.

[Futamura 83] Y. Futamura, "Partial Computation of Programs," Journal of IECE of Japan, Vol. 66, No. 2, 1983.

[Goebel 86] R. Goebel, K. Furukawa and D. Poole, "Using Definite Clauses and Integrity Constraints as the Basis for a Theory Formation Approach to Diagnostic Reasoning," in Proc. of the Third International Conf. on Logic Programming, London, 1986.

[Hammond 83] P. Hammond and M. Sergot, "A PROLOG Shell for Logic Based Expert Systems," Proc. of the Third BCS Expert Systems Conference, 1983.

[Jaffar 86] J. Jaffar and J-L. Lassez, "Constraint Logic Programming," Technical Report, Department of Computer Science, Monash University, June 1986.

[Kahn 86] K. Kahn, et al., "Vulcan: Logical Concurrent Objects," Proc. of the ACM Object-oriented Programming, System, Languages and Applications Conference, Oregon, 1986.

[Mukai 85] K. Mukai, H. Yasukawa, "Complex Indeterminates in Prolog and its Application to Discourse Models," New Generation Computing, Vol. 3, No. 4, 1985.

[Seki 86] H. Seki and K. Furukawa, "Compiling Control by a Program Transformation Approach," ICOT Technical Memo No.240, 1986.

[Shapiro 83a] E. Shapiro, "A Subset of Concurrent Prolog and Its Interpreter," ICOT Technical Report TR-003, 1983.

[Shapiro 83b] E. Shapiro, "Logic Programs with Uncertainties: A Tool for Implementing Rule-based Systems," Proc. of IJCAI'83, 1983.

[Sterling 84] L. Sterling, "Logical Levels of Problem Solving," Proc. of the Second International Conf. on Logic Programming, Uppsala University, 1984.

[Takeuchi 86] A. Takeuchi and K. Furukawa, "Partial Evaluation of Prolog Programs and its Application to Meta Programming," in Proc. of the IFIP Congress 86, 1986.

[Ueda 85] K. Ueda, "Guarded Horn Clauses," ICOT Technical Report TR-103, 1985, also in Logic Programming '85, Lecture Notes in Computer Science, 221, Springer-Verlag, 1986.

[Ueda 86] K. Ueda, "Making Exhaustive Search Programs Deterministic,"