PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. 



# Summary of Basic Research Activities of the FGCS Project

Koichi Furukawa

Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
furukawa@icot.or.jp

## Abstract

The Fifth Generation Computer Project was launched in 1982, with the aim of developing parallel computers dedicated to knowledge information processing. It is commonly believed to be very difficult to parallelize knowledge processing based on symbolic computation. We conjectured that *logic programming* technology would solve this difficulty.

We conducted our project while stressing two seemingly different aspects of logic programming: one was establishment of a new information technology, and the other was pursuit of basic AI and software engineering research.

In the former, we developed a concurrent logic programming language, GHC, and its extension for practical parallel programming, KL1. The invention of GHC/KL1 enabled us to conduct parallel research on the development of software technology and parallel hardware dedicated to the new language.

We also developed several constraint logic programming languages which are very promising as high level languages for AI applications. Though most of them are based on sequential Prolog technology, we are now integrating constraint logic programming and concurrent logic programming and developing an integrated language, GDCC.

In the latter, we investigated many fundamental AI and software engineering problems including hypothetical reasoning, analogical inference, knowledge representation, theorem proving, partial evaluation and program transformation.

As a result, we succeeded in showing that logic programming provides a very firm foundation for many aspects of information processing: from advanced software technology for AI and software engineering, through system programming and parallel programming, to parallel architecture.

The research activities are continuing and latest as well as earlier results strongly indicate the truth of our conjecture and also the fact that our approach is appropriate.

## 1 Introduction

In the Fifth Generation Computer Project, two main research targets were pursued: knowledge information processing and parallel processing. Logic programming was adopted as a key technology for achieving both targets simultaneously. At the beginning of the project, we adopted Prolog as our vehicle to promote the entire research of the project. Since there were no systematic research attempts based on Prolog before our project, there were many things to do, including the development of a suitable workstation for the research, experimental studies for developing a knowledge-based system in Prolog and investigation into possible parallel architecture for the language. We rapidly succeeded in promoting research in many directions.

From this research, three achievements are worth noting. The first is the development of our own workstation dedicated to ESP, Extended Self-contained Prolog. We developed an operating system for the workstation completely in ESP [Chikayama 88]. The second is the application of partial evaluation to meta programming. This enabled us to develop a compiler for a new programming language by simply describing an interpreter of the language and then partially evaluating it. We applied this technique to derive a bottom-up parser for context free grammar given a bottom up interpreter for them. In other words, partial evaluation made meta programming useful in real applications. The third achievement was the development of constraint logic programming languages. We developed two constraint logic programming languages: CIL and CAL. CIL is for natural language processing and is based on the incomplete data structure for representing "Complex Indeterminates" in situation theory. It has the capability to represent structured data like Minsky's frame and any relationship between slots' values can be expressed using constraints. CIL was used to develop a natural language understanding system called DUALS. Another constraint logic programming language, CAL, is for non-linear equations. Its inference is done using the Buchberger algorithm for computing the Gröbner Basis which is a variant of the Knuth-Bendix completion algorithm for a term rewriting

system.

We encountered one serious problem inherent to Prolog: that was the lack of concurrency in the fundamental framework of Prolog. We recognized the importance of concurrency in developing parallel processing technologies, and we began searching for alternative logic programming languages with the notion of concurrency.

We noticed the work by Keith Clark and Steve Gregory on Relational Language [ClarkGregory 81] and Ehud Shapiro on Concurrent Prolog [Shapiro 83]. These languages have a common feature of committed choice nondeterminism to introduce concurrency. We devoted our efforts to investigating these languages carefully and Ueda finally designed a new committed choice logic programming language called GHC [Ueda 86a] [UedaChikayama 90], which has simpler syntax than the above two languages and still have similar expressiveness. We recognized the importance of GHC and adopted it as the core of our kernel language, KL1, in this project. The introduction of KL1 made it possible to divide the entire research project into two parts: the development of parallel hardware dedicated to KL1 and the development of software technology for the language. In this respect, the invention of GHC is the most important achievement for the success of the Fifth Generation Computer Systems project.

Besides these language oriented researches, we performed many fundamental researches in the field of artificial intelligence and software engineering based on logic and logic programming. They include researches on nonmonotonic reasoning, hypothetical reasoning, abduction, induction, knowledge representation, theorem proving, partial evaluation and program transformation. We expected that these researches would become important application fields for our parallel machines by the affinity of these problems to logic programming and logic based parallel processing. This is now happening.

In this article, we first describe our research efforts in concurrent logic programming and in constraint logic programming. Then, we discuss our recent research activities in the field of software engineering and artificial intelligence. Finally, we conclude the paper by stating the dirction of future research.

# 2 Concurrent Logic Programming

In this section, we pick up two important topics in concurrent logic programming research in the project. One is the design principles of our concurrent logic programming language Flat GHC (FGHC) [Ueda 86a] [UedaChikayama 90], on which the aspects of KL1 as a concurrent language is based. The other is search paradigms in FGHC. As discussed later, one drawback of FGHC, viewing as a logic programming language, is the lack of search capability inherent in Prolog. Since the capability is related to the notion of completeness in logic programming, recovery of the ability is essential.

## 2.1 Design Principles of FGHC

The most important feature of FGHC is that there is only one syntactic extension to Prolog, called the commitment operator and represented by a vertical bar "|". A commitment operator divides an entire clause into two parts called the guard part (the left-hand side of the bar) and the body part (the right-hand side). The guard of a clause has two important roles: one is to specify a condition for the clause to be selected for the succeeding computation, and the other is to specify the synchronization condition. The general rule of synchronization in FGHC is expressed as *dataflow synchronization*. This means that computation is suspended until sufficient data for the computation arrives. In the case of FGHC, guard computation is suspended until the caller is sufficiently instantiated to judge the guard condition. For example, consider how a ticket vending machine works. After receiving money, it has to wait until the user pushes a button for the destination. This waiting is described as a clause such that "if the user pushed the 160-yen button, then issue a 160-yen ticket".

The important thing is that dataflow synchronization can be realized by a simple rule governing head unification which occurs when a goal is executed and a corresponding FGHC clause is called: the information flow of head unification must be one way, from the caller to the callee. For example, consider a predicate representing service at a front desk. Two clauses define the predicate: one is for during the day, when more customers are expected, and another is for after-hours, when no more customers are expected. The clauses have such definitions as:

```
serve([First | Rest]) :- <extra-condition> |
     do_service(First), serve(Rest).
serve([]) :- true | true.
```

Besides the *serve* process, there should be another process *queue* which makes a waiting queue for service. The top level goal looks like:

```
?- queue(Xs), serve(Xs).
```

where "?-" is a prompt to the user at the terminal. Note that the execution of this goal generates two processes, queue and serve, which share a variable Xs. This shared variable acts as a channel for data transfer from one process to the other. In the above example, we assume that the queue process instantiates Xs and the serve process reads the value. In other words, queue acts as a generator of the value of Xs and serve acts as the consumer. The process queue instantiates Xs either to a

list of servees represented by [<first-servee>, <second-servee>,...] or to an empty list []. Before the instantiation, the value of Xs remains undefined.

Suppose Xs is undefined. Then, the head unification invoked by the goal serve(Xs) suspends because the equations Xs = [First | Rest] and Xs = [] cannot be solved without instantiating Xs. But such instantiation violates the rule of one-way unification. Note that the term [First | Rest] in the head of serve means that the clause expects a non-empty list to be given as the value of the argument. Similarly, the term [] expects an empty list to be given. Now, it is clear that the unidirectionality of information flow realizes dataflow synchronization.

This principle is very important in two aspects: one is that the language provides a natural tool for expressing concurrency, and the other is that the synchronization mechanism is simple enough to realize very efficient parallel implementation.

## 2.2 Search Paradigms in FGHC

There is one serious drawback to FGHC because of the very nature of committed choice; that is, it no longer has an automatic search capability, which is one of the most important features of Prolog. Prolog achieves its search capability by means of automatic backtracking. However, since committed choice uniquely determines a clause for succeeding computation of a goal, there is no way of searching for alternative branches other than the branch selected. The search capability is related to the notion of completeness of the logic programming computation procedure and the lack of the capability is very serious in that respect.

One could imagine a seemingly trivial way of realizing search capability by means of *or-parallel* search: that is, to copy the current computational environment which provides the binding information of all variables that have appeared so far and to continue computations for each alternative case in parallel. But this does not work because copying non-ground terms is impossible in FGHC. The reason why it is impossible is that FGHC cannot guarantee when actual binding will occur and there may be a moment when a variable observed at some processor remains unchanged even after some goal has instantiated it at a different processor.

One might ask why we did not adopt a Prolog-like language as our kernel language for parallel computation. There are two main reasons. One is that, as stated above, Prolog does not have enough expressiveness for concurrency, which we see as a key feature not only for expressing concurrent algorithms but also for providing a framework for the control of physical parallelism. The other is that the execution mechanism of Prolog-like languages with a search capability seemed too complicated to develop efficient parallel implementations.

We tried to recover the search capability by devising programming techniques while keeping the programming language as simple as possible. We succeeded in inventing several programming methods for computing all solutions of a problem which effectively achieve the completeness of logic programming. Three of them are listed as follows:

(1) Continuation-based method [Ueda 86b]
(2) Layered stream method [OkumuraMatsumoto 87]
(3) Query compilation method [Furukawa 92]

In this paper, we pick up (1) and (3), which are complementary to each other. The continuation-based method is suitable for the efficient processing of rather-algorithmic problems. An example is to compute all ways of partitioning a given list into two sublists by using *append*. This method mimics the computation of *OR-parallel* Prolog using *AND-parallelism* of FGHC. *AND-serial* computation in Prolog is translated to *continuation* processing which remembers continuation points in a stack. The intermediate results of computation are passed from the preceding goals to the next goals through the continuation stack kept as one of the arguments of the FGHC goals. This method requires input/output mode analysis before translating a Prolog program into FGHC. This requirement makes the method impractical for database applications because there are too many possible input-output modes for each predicate.

The query compilation method solves this problem. This method was first introduced by Fuchi [Fuchi 90] when he developed a bottom-up theorem prover in KL1. In his coding technique, the multiple binding problem is avoided by reversing the role of the caller and the callee in straightforward implementation of database query evaluation. Instead of trying to find a record (represented by a clause) which matches a given query pattern represented by a goal, his method represents each query component with a compiled clause, represents a databasae with a data structure passed around by goals, and tries to find a query component clause which matches a goal representing a record and recurses the process for all potentially applicable records in the database[1]. Since every record is a ground term, there is no variable in the caller. Variable instantiation occurs when query component clauses are searched and an appropriate clause representing a query component is found to match a currently processed record. Note that, as a result of reversing the representation of queries and databases from straightforward representation, the information flow is now from the caller (database) to the callee (a query component). This inversion of information flow avoids deadlock in query processing. Another important trick is that each time a query clause is called, a fresh variable is created for each variable in the query component. This mechanism is used for making a new environment

[1] We need an *auxiliary* query clause which matches every record after failing to match the record to all the *real* query clauses.

for each OR-parallel computation branch. These tricks make it possible to use KL1 variables to represent object level variables in database queries and, therefore, we can avoid different compilation of the entire database and queries for each input/output mode of queries.

The new coding method stated above is very general and there are many applications which can be programmed in this way. The only limitation of this approach is that the database must be more instantiated than queries. In bottom-up theorem proving, this requirement is referred to as the range-restrictedness of each axiom. Range-restrictedness means that, after successfully finding ground model elements satisfying the antecedent of an axiom, the new model element appearing as the consequent of the axiom must be ground.

This restriction seems very strong. Indeed, there are problems in the theorem proving area which do not satisfy the condition. We need a top-down theorem prover for such problems. However, many real life problems satisfy the range-restrictedness because they almost always have finite concrete models. Such problems include VLSI-CAD, circuit diagnosis, planning, and scheduling. We are developing a parallel bottom-up theorem prover called MGTP (Model Generation Theorem Prover) [FujitaHasegawa 91] based on SATCHMO developed by Manthey and Bry [MantheyBry 88]. We are investigating new applications to utilize the theorem prover. We will give an example of computing abduction using MGTP in Section 5.

# 3 Constraint Logic Programming

We began our constraint logic programming research almost at the beginning of our project, in relation to the research on natural language processing. Mukai [MukaiYasukawa 85] developed a language called CIL (Complex Indeterminates Language) for the purpose of developing a computational model of situation semantics. A *complex indeterminate* is a data structure allowing partially specified terms with indefinite arity. During the design phase of the language, he encountered the idea of *freeze* in Prolog II by Colmerauer [Colmerauer 86]. He adopted *freeze* as a proper control structure for our CIL language.

From the viewpoint of constraint satisfaction, CIL only has a passive way of solving constraint, which means that there is no active computation for solving constraints such as constraint propagation or solving simultaneous equations. Later, we began our research on constraint logic programming involving active constraint solving. The language we developed is called CAL. It deals with non-linear equations as expressions to specify constraints. Three events triggered the research: one was our preceding efforts on developing a term rewriting system called METIS for a theorem prover of linear algebra [OhsugaSakai 91]. Another event was our encounter with Buchberger's algorithm for computing the Gröbner Basis for solving non-linear equations. Since the algorithm is a variant of the Knuth-Bendix completion algorithm for a term rewriting system, we were able to develop the system easily from our experience of developing METIS. The third event was the development of the CLP(X) theory by Jaffar and Lassez which provides a framework for constraint logic programming languages [JaffarLassez 86].

There is further remarkable research on constraint logic programming in the field of general symbol processing [Tsuda 92]. Tsuda developed a language called cu-Prolog. In cu-Prolog, constraints are solved by means of program transformation techniques called unfold/fold transformation (these will be discussed in more detail later in this paper, as an optimization technique in relation to software engineering). The unfold/fold program transformation is used here as a basic operation for solving combinatorial constraints among terms. Each time the transformation is performed, the program is modified to a syntactically less constrained program. Note that this basic operation is similar to *term rewriting*, a basic operation in CAL. Both of these operations try to rewrite programs to get certain canonical forms. The idea of cu-Prolog was introduced by Hasida during his work on *dependency propagation* and *dynamical programming* [Hasida 92]. They succeeded in showing that context-free parsing, which is as efficient as *chart parsing*, emerges as a result of dependency propagation during the execution of a program given as a set of grammar rules in cu-Prolog. Actually, there is no need to construct a parser. cu-Prolog itself works as an efficient parser.

Hasida [Hasida 92] has been working on a fundamental issue of artificial intelligence and cognitive science from the aspect of a computational model. In his computation model of dynamical programming, computation is controlled by various kinds of *potential energies* associated with each atomic constraint, clause, and unification. Potential energy reflects the degree of constraint violation and, therefore, the reduction of energy contributes constraint resolution.

Constraint logic programming greatly enriched the expressiveness of Prolog and is now providing a very promising programming environment for applications by extending the domain of Prolog to cover most AI problems.

One big issue in our project is how to integrate constraint logic programming with concurrent logic programming to obtain both expressiveness and efficiency.

This integration, however, is not easy to achieve because (1) constraint logic programming focuses on a control scheme for efficient execution specific to each constraint solving scheme, and (2) constraint logic programming essentially includes a search paradigm which re-

quires some suitable support mechanism such as automatic backtracking.

It turns out that the first problem can be processed efficiently, to some extent, in the concurrent logic programming scheme utilizing the data flow control method. We developed an experimental concurrent constraint logic programming language called GDCC (Guarded Definite Clauses with Constraints), implemented in KL1 [HawleyAiba 91]. GDCC is based on an ask-tell mechanism proposed by Maher [Maher 87], and extended by Saraswat [Saraswat 89]. It extends the guard computation mechanism from a simple one-way unification solving problem to a more general provability check of conditions in the guard part under a given set of constraints using the *ask* operation. For the body computation, constraint literals appearing in the body part are added to the constraint set using the *tell* operation. If the guard conditions are not known to be provable because of a lack of information in the constraints set, then computation is suspended. If the conditions are disproved under the constraints set, then the guard computation fails. Note that the provability check controls the order of constraint solving execution. New constraints appearing in the body of a clause are not included in the constraint set until the guard conditions are known to be provable.

The second problem of realizing a search paradigm in a concurrent constraint logic programming framework has not been solved so far. One obvious way is to develop an OR-parallel search mechanism which uses a full unification engine implemented using ground term representation of logical variables [Koshimura *et al.* 91]. However, the performance of the unifier is 10 to 100 times slower than the built in unifier and, as such, it is not very practical. Another possible solution is to adopt the new coding technique introduced in the previous section. We expect to be able to efficiently introduce the search paradigm by applying the coding method. The paradigm is crucial if parallel inference machines are to be made useful for the numerous applications which require high levels of both expressive and computational power.

# 4 Advanced Software Engineering

Software engineering aims at supporting software development in various dimensions; increase of software productivity, development of high quality software, pursuit of easily maintainable software and so on. Logic programming has great potential for many dimensions in software engineering. One obvious advantage of logic programming is the affinity for correctness proof when given specifications. Automatic debugging is a related issue. Also, there is a high possibility of achieving automatic program synthesis from specifications by applying proof techniques as well as from examples by applying induction. Program optimization is another promising direction where logic programming works very well.

In this section, two topics are picked up: (1) meta programming and its optimization by partial evaluation, and (2) unfold/fold program transformation.

## 4.1 Meta Programming and Partial Evaluation

We investigated meta programming technology as a vehicle for developing knowledge-based systems in a logic programming framework inspired by Bowen and Kowalski's work [BowenKowalski 83]. It was a rather direct way to realize a knowledge assimilation system using the meta programming technique by regarding integrity constraints as meta rules which must be satisfied by a knowledge base. One big problem of the approach was its inefficiency due to the meta interpretation overhead of each object level program. We challenged the problem and Takeuchi and Furukawa [TakeuchiFurukawa 86] made a breakthrough in the problem by applying the optimization technique of partial evaluation to meta programs. We first derived an efficient *compiled* program for an expert system with uncertainty computation given a meta interpreter of rules with certainty factor. In this program, we succeeded in getting three times speedup over the original program. Then, we tried a more non-trivial problem of developing a meta interpreter of a bottom-up parser and deriving an efficient *compiled* program given the interpreter and a set of grammar rules. We succeeded in obtaining an object program known as BUP, developed by Matsumoto [Matsumoto *et al.* 83]. The importance of the BUP meta-interpreter is that it is not a vanilla meta-interpreter, an obvious extension of the Prolog interpreter in Prolog, because the control structure is totally different from Prolog's top-down control structure.

After our first success of applying partial evaluation techniques in meta programming, we began the development of a self-applicable partial evaluator. Fujita and Furukawa [FujitaFurukawa 88] succeeded in developing a simple self-applicable partial evaluator. We showed that the partial evaluator itself was a meta interpreter very similar to the following Prolog interpreter in Prolog:

```
solve(true).
solve((A,B)) :-  solve(A),   solve(B).
solve(A)     :- clause(A,B), solve(B).
```

where it is assumed that for each program clause, $H$ :- $B$, a unit clause, `clause(`$H$`,`$B$`)`, is asserted[2]. A goal, `solve(`$G$`)`, simulates an immediate execution of the subject goal, $G$, and obtains the same result.

This simple definition of a Prolog self-interpreter, `solve`, suggests the following *partial solver*, `psolve`.

[2] `clause(_,_)` is available as a built-in procedure in the DECsystem-10 Prolog system.

```
psolve(true,true).
psolve((A,B),(RA,RB)) :-
     psolve(A,RA), psolve(B,RB).
psolve(A,R) :-
     clause(A,B), psolve(B,R).
psolve(A,A) :- '$suspend'(A).
```

The partial solver, psolve($G$,$R$), partially solves a given goal, $G$, returning the result, $R$. The result, $R$, is called the *residual goal(s)* for the given goal, $G$. The residual goal may be true when the given goal is totally solved, otherwise it will be a conjunction of subgoals, each of which is a goal, $R_i$, suspended to be solved at '$suspend'($R_i$), for some reason. An *auxiliary predicate*, '$suspend'($P$), is defined for each goal pattern, $P$, by the user.

Note that psolve is related to solve as:

```
solve(G) :- psolve(G,R), solve(R).
```

That is, a goal, $G$, succeeds if it is partially solved with the residual goal, $R$, and $R$ in turn succeeds (is totally solved). The total solution for $G$ is thus split into two tasks: partial solution for $G$ and total solution for the residual goal, $R$.

We developed a self-applicable partial evaluator by modifying the above psolve program. The main modification is the treatment of built-in predicates in Prolog and those predicates used to define the partial evaluator itself to make it self-applicable. We succeeded in applying the partial evaluator to itself and generated a compiler by partially evaluating the psolve program with respect to a given interpreter, using the identical psolve. We further succeeded in obtaining a compiler generator, which generates different compilers given different interpreters, by partially evaluating the psolve program with respect to itself, using itself.

Theoretically, it was known that self-application of a partial evaluator generates compilers and a compiler generator [Futamura 71]. There were many attempts to realize self-applicable partial evaluators in the framework of functional languages for a long time, but no successes were reported until very recently [Jones *et al.* 85], [Jones *et al.* 88], [GomardJones 89]. On the other hand, we succeeded in developing a self-applicable partial evaluator in a Prolog framework in a very short time and also in a very elegant way. This proves some merits of logic programming languages over functional programming languages, especially in its binding scheme based on unification.

## 4.2 Unfold/Fold Program Transformation

Program transformation provides a powerful methodology for the development of software, especially the derivation of efficient programs either from their formal specification or from decralative but possibly inefficient programs. Programs written in declarative form are often inefficient under Prolog's standard left to right control rule. Typical examples are found in programs based on a generate and test paradigm. Seki and Furukawa [SekiFurukawa 87] developed a program transformation method based on unfolding and folding for such programs. We will explain the idea in some detail. Let gen_test(L) be a predicate defined as follows:

```
gen_test(L) :- gen(L), test(L).
```

where L is a variable representing a list, gen(L) is a generator of the list L, and test(L) is a tester for L. Assume both gen and test are incremental and are defined as follows:

```
gen([]).
gen([X|L]) :- gen_element(X), gen(L).

test([]).
test([X|L]) :- test_element(X), test(L).
```

Then, it is possible to fuse two processes gen and test by applying unfold/fold transformation as follows:

```
gen_test([X|L]) :- gen([X|L]), test([X|L]).
```

*unfold* at gen and test

```
gen_test([X|L]) :- gen_element(X), gen(L),
    test_element(X), test(L).
```

*fold* by gen_test

```
gen_test([X|L]) :- gen_element(X),
    test_element(X), gen_test(L).
```

If the tester is not incremental, the above unfold/fold transformation does not work. One example is to test that all elements in the list are different from each other. In this case, the test predicate is defined as follows:

```
test([]).
test([X|L]) :- non_member(X,L), test(L).

non_member(_,[]).
non_member(X,[Y|L]):-
    dif(X,Y), non_member(X,L).
```

where dif(X,Y) is a predicate judging that X is not equal to Y. Note that this test predicate is not incremental because a test for the first element X of the list requires the information of the entire list. The solution we gave to this problem was to replace the test predicate with an equivalent predicate with incrementality. Such an equivalent program test' is obtained by adding an *accumulator* as an extra argument of the test predicate defined as follows:

```
test'([],_).
test'([X|L],Acc) :-
    non_member(X,Acc), test'(L,[X|Acc]).
```

The relationship between test and test' is given by the following theorem:

Theorem

$$\texttt{test(L)} \equiv \texttt{test'(L,[])}$$

Now, the original gen_test program becomes

```
gen_test(L) :- gen(L), test'(L,[]).
```

We need to introduce the following new predicate to perform the unfold/fold transformation:

```
gen_test'(L,Acc) :- gen(L), test'(L,Acc).
```

By applying a similar transformation process as before, we get the following fused recursive program of gen_test':

```
gen_test'([],_).
gen_test'([X|L],Acc) :- gen_element(X),
    non_member(X,Acc), gen_test'(L,[X|Acc]).
```

By symbolically computing the two goals

```
?- test([X1,...,Xn]).

?- test'([X1,...,Xn]).
```

and comparing the results, one can find that the reordering of pair-wise comparisons by the introduction of the accumulator is analogous to the exchange of double summation $\Sigma_{i=1}^{N}\Sigma_{j=i}^{N}x_{ij} = \Sigma_{j=1}^{N}\Sigma_{i=1}^{j}x_{ij}$. Therefore, we refer to this property as structural commutativity.

One of the key problems of unfold/fold transformation is the introduction of a new predicate such as gen_test' in the last example. Kawamura [Kawamura 91] developed a syntactic rule for finding suitable new predicates. There were several attempts to find appropriate new predicates using domain dependent heuristic knowledge, such as append optimization by the introduction of difference list representation. Kawamura's work provides some general criteria for selecting candidates for new predicates. His method first analyzes a given program to be transformed and makes a list of patterns which may possibly appear in the definition of new predicates. This can be done by unfolding a given program and properly generalizing all resulting patterns to represent them with a finite number of distinct patterns while avoiding over-generalization. One obvious strategy to avoid over-generalization is to perform least general generalization by Plotkin [Plotkin 70]. Kawamura also introduced another strategy for suppressing unnecessary generalization: a subset of clauses of which the head can be unifiable to each pattern is associated with the pattern and only those patterns having the same associated subset of clauses are generalized. Note that a goal pattern is unfolded only by clauses belonging to the associated subset. Therefore the suppression of over-generalization also suppresses unnecessary expansion of clauses by unnecessary unfolding.

# 5 Logic-based AI Research

For a long time, deduction has played a central role in research on logic and logic programming. Recently, two other inferences, abduction and induction, received much attention and much research has been done in these new directions. These directions are related to fundamental AI problems that are open-ended by their nature. They include the frame problem, machine learning, distributed problem solving, natural language understanding, common sense reasoning, hypothetical reasoning and analogical reasoning. These problems require non-deductive inference capabilities in order to solve them.

Historically, most AI research on these problems adopted ad hoc heuristic methods reflecting problem structures. There was a tendency to avoid a logic based formal approach because of a common belief in the limitation of the formalism. However, the limitation of logical formalism comes only from the deductive aspect of logic. Recently it has been widely recognized that abduction and induction based on logic provide a suitable framework for such problems requiring open-endedness in their formalism. There is much evidence to support this observation.

- In natural language understanding, unification grammar is playing an important role in integrating syntax, semantics, and discourse understanding.
- In non-monotonic reasoning, logical formalism such as circumscription and default reasoning and its compilation to logic based programs are studied extensively.
- In machine learning, there are many results based on logical frameworks such as the Model Inference System, inverse resolution, and least general generalization.
- In analogical reasoning, analogy is naturally described in terms of a formal inference rule similar to logical inference. The associated inference is deeply related to abductive inference.

In the following, three topics related to these issues are picked up: they are hypothetical reasoning, analogy, and knowledge representation.

## 5.1 Hypothetical Reasoning

A logical framework of hypothetical reasoning was studied by Poole et al. [Poole *et al.* 87]. They discussed the relationship among hypothetical reasoning, default reasoning and circumscription, and argued that hypothetical reasoning is all that is needed because it is simply and efficiently implemented and is powerful enough to implement other forms of reasoning. Recently, the relationship of these formalisms was studied in more detail and many attempts were made to translate non-monotonic reasoning problems into equivalent logic programs with negation as failure.

Another direction of research was the formulation of abduction and its relationship to negation as failure. There was also a study of the model theory of a class of logic programs, called general logic programs, allowing negation by failure in the definition of bodies in the clausal form. By replacing negation-by-failure predicates by corresponding abducible predicates which usually give negative information, we can formalize negation by failure in terms of abduction [EshghiKowalski 89]

A proper semantics of general logic programs is given by stable model semantics [GelfondLifschitz 88]. It is a natural extension of least fixpoint semantics. The difference is that there is no $T_P$ operator to compute the stable model directly, because we need a complete model for checking the truth value of the literal of negation by failure in bottom-up fixpoint computation. Therefore, we need to refer to the model in the definition of the model. This introduces great difficulty in computing stable models. The trivial way is to assume all possible models and see whether the initial models are the least ones satisfying the programs or not. This algorithm needs to search for all possible subsets of atoms to be generated by the programs and is not realistic at all.

Inoue [Inoue *et al.* 92] developed a much more efficient algorithm for computing all stable models of general logic programs. Their algorithm is based on bottom-up model generation method. Negation-by-failure literals are used to introduce hypothetical models: ones which assume the truth of the literals and the others which assume that they are false. To express assumed literals, they introduce a modal operator. More precisely, they translate each rule of the form:

$$A_l \leftarrow A_{l+1} \wedge \ldots \wedge A_m, \mathit{not}\, A_{m+1} \wedge \ldots \wedge \mathit{not}\, A_n$$

to the following disjunctive clause which does not contain any negation-by-failure literals:

$$A_{l+1} \wedge \ldots \wedge A_m \rightarrow$$
$$(\mathsf{NK}A_{m+1} \wedge \ldots \wedge \mathsf{NK}A_n \wedge A_l) \vee \mathsf{K}A_{m+1} \vee \ldots \vee \mathsf{K}A_n .$$

The reason why we express the clause with the antecedent on the left hand side is that we intend to use this clause in a bottom-up way; that is, from left to right. In this expression, NK$A$ means that we *assume* that A is false, whereas, K$A$ means that we *assume* that A is true. Although K and NK are modal operators, we can treat K$A$ and NK$A$ as new predicates independent from $A$ by adding the following constraints:

$$\mathsf{NK}A, A \rightarrow \quad \text{for every atom } A. \tag{1}$$

$$\mathsf{NK}A, \mathsf{K}A \rightarrow \quad \text{for every atom } A. \tag{2}$$

By this translation, we obtain a set of clauses in first order logic and therefore it is possible to compute all possible models for the set using a first order bottom-up theorem prover, MGTP, described in Section 2. After computing all possible models for the set of clauses, we need to select only those models $M$ which satisfy the following condition:

$$\text{For every ground atom } A, \text{ if } \mathsf{K}A \in M, \text{ then } A \in M. \tag{3}$$

Note that this translation scheme defines a coding method of original general logic programs which may contain negation by failure in terms of pure first order logic. Note also that the same technique can be applied in computing abduction, which means to find possible sets of hypotheses explaining the observation and not contradicting given integrity constraints.

Satoh and Iwayama [SatohIwayama 92] independently developed a top-down procedure for answering queries to a general logic program with integrity constraints. They modified an algorithm proposed by Eshghi and Kowalski [EshghiKowalski 89] to correctly handle situations where some proposition must hold in a model, like the requirement of (3).

Iwayama and Satoh [IwayamaSatoh 91] developed a mixed strategy combining bottom-up and top-down strategies for computing the stable models of general logic programs with constraints. The procedure is basically bottom-up. The top-down computation is related to the requirement of (3) and as soon as a hypothesis of K$A$ is asserted in some model, it tries to prove $A$ by a top-down expectation procedure.

The formalization of abductive reasoning has a wide range of applications including computer aided design and fault diagnosis. Our approach provides a uniform scheme for representing such problems and solving them. It also provides a way of utilizing our parallel inference machine, PIM, for solving these complex AI problems.

## 5.2 Formal Approach to Analogy

Analogy is an important reasoning method in human problem solving. Analogy is very helpful for solving problems which are very difficult to solve by themselves. Analogy guides the problem solving activities using the knowledge of how to solve a similar problem. Another aspect of analogy is to extract good guesses even when there is not enough information to explain the answer.

There are three major problems to be solved in order to mechanize analogical reasoning [Arima 92]:

- searching for an appropriate base of analogy with respect to a given target,
- selecting important properties shared by a base and a target, and
- selecting properties to be projected through an analogy from a base to a target.

Though there was much work on mechanizing analogy, most of this only partly addressed the issues listed above. Arima [Arima 92] proposed an attempt to answer all the issues at once. Before explaining his idea, we need some preparations for defining terminology.

Analogical reasoning is expressed as the following inference rule:

$$\frac{S(B) \wedge P(B)}{\begin{array}{l} S(T) \\ \hline P(T) \end{array}}$$

where $T$ represents the *target* object, $B$ the *base* object, $S$ the *similarity property* between $T$ and $B$, and $P$ the *projected property*.

This inference rule expresses that if we assume an object $T$ is similar to another object $B$ in the sense that they share a common property $S$ then, if $B$ has another property $P$, we can analogically reason that $T$ also has the same property $P$. Note that the syntactic similarity of this rule to *modus ponens*. If we generalize the object $B$ to a universally quantified variable $X$ and replace the *and* connective to the *implication* connective, then the first expression of the rule becomes $S(X) \supset P(X)$, thereby the entire rule becomes *modus ponens*.

Arima [Arima 92] tried to link the analogical reasoning to deductive reasoning by modifying the expression $S(B) \wedge P(B)$ to

$$\forall x.(J(x) \wedge S(x) \supset P(x)), \tag{4}$$

where $J(x)$ is a hypothesis added to $S(x)$ in order to logically conclude $P(x)$. If there exists such a $J(x)$, then the analogical reasoning becomes pure deductive reasoning. For example, let us assume that there is a student ($Student_B$) who belongs to an orchestra club and also neglects study. If one happens to know that another student ($Student_T$) belongs to the orchestra club, then we tend to conclude that he also neglects study. The reason why we derive such a conclusion is that we guess that the orchestra club is very active and student members of this busy club tend to neglect study. This reason is an example of the hypothesis mentioned above.

Arima analyzed the syntactic structure of the above $J(x)$ by carefully observing the analogical situation. First, we need to find a proper parameter for the predicate $J$. Since it is dependent on not only an object but also the similarity property and the projected property, we assume that $J$ has the form of $J(x, s, p)$, where $s$ and $p$ represent the similarity property and the projected property.

From the nature of analogy, we do not expect that there is any direct relationship between the object $x$ and the projected property $p$. Therefore, the entire $J(x, s, p)$ can be divided into two parts:

$$J(x, s, p) \;=\; J_{att}(s, p) \wedge J_{obj}(x, s), \tag{5}$$

The first component, $J_{att}(s, p)$, corresponds to information extracted from a base. The reason why it does not depend on $x$ comes from the observation that information in the base of the analogy is independent from the choice of an object $x$.

The second component, $J_{obj}(x, s)$, corresponds to information extracted from the similarity and therefore it does not contain $p$ as its parameter.

### Example: Negligent Student

First, let us formally describe the hypothesis described earlier to explain why an orchestra member is negligent of study. It is expressed as follows:

$$\begin{array}{c} \forall x, s, p.(\ Enthusiastic(x, s) \wedge BusyClub(s) \\ \wedge Obstructive\_to(p, s) \wedge Member\_of(x, s) \\ \supset Negligent\_of(x, p)\ ) \end{array} \tag{6}$$

where $x, s$, and $p$ are variables representing a person, a club and some human activity, respectively. The meaning of each predicate is easy to understand and the explanations are omitted. Since we know that both $Student_B$ and $Student_T$ are members of an orchestra, $Members\_of(X, s)$ corresponds to the similarity property $S(x)$ in (4). On the other hand, since we want to reason the negligence of a student, the projected property $P(x)$ is $Negligent\_of(x, p)$. Therefore, the rest of the expression in (6): $Enthusiastic(x, s) \wedge BusyClub(s) \wedge Obstructive\_to(p, s)$ corresponds to $J(x, s, p)$. From the syntactic feature of this expression, we can conclude that

$J_{obj}(x, s) = Enthusiastic(x, s)$,
$J_{att}(s, p) = BusyClub(s) \wedge Obstructive\_to(p, s)$.

The reason why we need $J_{obj}$ is that we are not always aware of an important similarity like *Enthusiastic*. Therefore, we need to infer an important hidden similarity from the given similarity such as $Member\_of$. This inference requires an extra effort in order to apply the above framework of analogy.

The restriction on the syntactic structure of $J(x, s, p)$ is very important since it can be used to prune a search space to access the right base case given the target. This function is particularly important when we apply our analogical inference framework to case based reasoning systems.

### 5.3 Knowledge Representation

Knowledge representation is one of the central issues in artificial intelligence research. Difficulty arises from the fact that there has been no single knowledge representation scheme for representing various kinds of knowledge while still keeping the simplicity as well as the efficiency of their utilization. Logic was one of the most promising candidates but it was weak in representing structured knowledge and the changing world. Our aim in developing a knowledge representation framework based on logic and logic programming is to solve both of these problems. From the structural viewpoint, we developed an extended relational database which can handle non-normal forms and its corresponding programming language, CRL [Yokota 88a]. This representation allows users to describe their databases in a structured way in the logical framework [Yokota *et al.* 88b].

Recently, we proposed a new logic-based knowledge representation language, Quixote [YasukawaYokota 90]. Quixote follows the ideas developed in CRL and CIL: it inherits object-orientedness from the extended version of CRL and partially specified terms from CIL. One of the main characteristics of the object-oriented features is the notion of object identity. In Quixote, not only simple data atoms but also complex structures are candidates for object identifiers [Morita 90]. Even circular structures can be represented in Quixote. The non-well founded set theory by Aczel [Aczel 88] was adopted to characterize them as a mathematical foundation for such objects, and unification on infinite trees [Colmerauer 82] was adopted as an implementation method.

## 6 Conclusion

In this article, we summarized the basic research activities of the FGCS project. We emphasized two different directions of logic programming research. One followed logic programming languages where constraint logic programming and concurrent logic programming were focussed. The other followed basic research in artificial intelligence and software engineering based on logic and logic programming.

This project has been like solving a jigsaw puzzle. It is like trying to discover the hidden picture in the puzzle using logic and logic programming as clues. The research problems to be solved were derived naturally from this image. There were several difficult problems. For some problems, we did not even have the right evaluation standard for judging the results. The design of GHC is such an example. Our entire picture of the project helped in guiding our research in the right direction.

The picture is not completed yet. We need further efforts to fill in the remaining spaces. One of the most important parts to be added to this picture is the integration of constraint logic programming and concurrent logic programming. We mentioned our preliminary language/system, GDCC, but this is not yet matured. We need a really useful language which can be efficientlly executed on parallel hardware. Another research subject to be pursued is the realization of a database in KL1. We are actively constructing a parallel database but it is still in the preliminary stages. We believe that there is much affinity between databases and parallelism and we expect a great deal of parallelism from database applications. The third research subject to be pursued is the parallel implementation of abduction and induction. Recently, there has been much work on abduction and induction based on logic and logic programming frameworks. They are expected to provide a foundation for many research themes related to knowledge acquisition and machine learning. Also, both abduction and induction require extensive symbolic computation and, therefore, fit very well with PIM architecture.

Although further research is needed to make our results really useful in a wide range of large-scale applications, we feel that our approach is in the right direction.

## Acknowledgements

This paper reflects all the basic research activities in the Fifth Generation Computer Systems project. The author would like to express his thanks to all the researchers in ICOT, as well as those in associated companies who have been working on this project. He especially would like to thank Akira Aiba, Jun Arima, Hiroshi Fujita, Kôiti Hasida, Katsumi Inoue, Noboru Iwayama, Tadashi Kawamura, Ken Satoh, Hiroshi Tsuda, Kazunori Ueda, Hideki Yasukawa and Kazumasa Yokota for their help in greatly improving this work. Finally, he would like to express his deepest thanks to Dr. Fuchi, the director of ICOT, for providing the opportunity to write this paper.

## References

[Arima 92] J. Arima, Logical Structure of Analogy. *In Proc. of the International Conf. on Fifth Generation Computer Systems 1992*, Tokyo, 1992.

[Aczel 88] P. Aczel, *Non-Well Founded Set Theory.* CLSI Lecture Notes No. 14, 1988.

[Aiba *et al.* 88] A. Aiba, K. Sakai, Y. Sato, D.J. Hawley, and R. Hasegawa, Constraint Logic Programming Language CAL. In *Proc. of the International Conf. on Fifth Generation Computing Systems 1988,* Tokyo, 1988.

[BowenKowalski 83] K. Bowen and R. Kowalski, Amalgamating Language and Metalanguage

in Logic Programming. In *Logic Programming*, K. Clark and S. Tarnlund (eds.), Academic Press, 1983.

[ClarkGregory 81] K. L. Clark and S. Gregory, *A Relational Language for Parallel Programming*. In Proc. ACM Conf. on Functional Programming Languages and Computer Architecture, ACM, 1981.

[ClarkGregory 86] K. L. Clark and S. Gregory, *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.

[Chikayama 88] T. Chikayama, Programming in ESP – Experiences with SIMPOS –, In *Programming of Future Generation Computers*, Fuchi and Nivat (eds.), North-Holland, 1988.

[Colmerauer 82] A. Colmerauer, Prolog and Infinite Trees. In *Logic Programming*, K. L. Clark and S. Å. Tärnlund (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.

[FuchiFurukawa 87] K. Fuchi and K. Furukawa, *The Role of Logic Programming in the Fifth Generation Computer Project*. New Generation Computing, Vol. 5, No. 1, Ohmsha-springer, 1987.

[EshghiKowalski 89] K. Eshghi and R.A. Kowalski, Abduction compared with negation by failure, in: *Proceedings of the Sixth International Conference on Logic Programming*, Lisbon, Portugal, 1989.

[Fuchi 90] K. Fuchi, *An Impression of KL1 Programming - from my experience with writing parallel provers* -. In Proc. of KL1 Programming Workshop '90, ICOT, 1990 (in Japanese).

[FujitaFurukawa 88] H. Fujita and K. Furukawa, *A Self-Applicable Partial Evaluator and Its Use in Incremental Compilation*. New Generation Computing, Vol. 6, Nos.2,3, Ohmsha/Springer-Verlag, 1988.

[FujitaHasegawa 91] H. Fujita and R. Hasegawa, *A Model Generation Theorem Prover in KL1 Using a Ramified-Stack Algorithm*. In Proc. of the Eighth International Conference on Logic Programming, Paris, 1991.

[Furukawa 92] K. Furukawa, Logic Programming as the Integrator of the Fifth Generation Computer Systems Project, *Communication of the ACM*, Vol. 35, No. 3, 1992.

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

[GelfondLifschitz 88] M. Gelfond and V. Lifschitz, The stable model semantics for logic programming, In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seattle, WA, 1988.

[GomardJones 89] C. K. Gomard and N. D. Jones, Compiler Generation by Partial Evaluation: A Case Study. In *Proc. of Information Processing 89*, G. X. Ritter (ed.), North-Holland, 1989.

[Hasida 92] K. Hasida, Dynamics of Symbol Systems - An Integrated Architecture of Cognition. *In Proc. of the International Conf. on Fifth Generation Computer Systems 1992*, Tokyo, 1992.

[HawleyAiba 91] D. Hawley and A. Aiba, *Guarded Definite Clauses with Constraints – Preliminary Report*. Technical Report TR-713, ICOT, 1991.

[Inoue *et al.* 92] K. Inoue, M. Koshimura and R. Hasegawa, Embedding Negation as Failure into a Model Generation Theorem Prover. *To appear in CADE-11: The Eleventh International Conference on Automated Deduction*, Saratoga Springs, NY, June 1992.

[IwayamaSatoh 91] N. Iwayama and K. Satoh, *A Bottom-up Procedure with Top-down Expectation for General Logic Programs with Integrity Constraints*. ICOT Technical Report TR-625, 1991.

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

[Jones *et al.* 85] N.D. Jones, P. Sestoft, and H. Søndergaard, An Experiment in Partial Evaluation: The Generation of a Compiler Generator. In J-.P. Jouannaud (ed.), *Rewriting Techniques and Applications,* LNCS-202, Springer-Verlag, pp.124–140, 1985.

[Jones *et al.* 88] N. D. Jones, P. Setstoft and H. Sondergaard, MIX: a self-applicable partial evaluator for experiments in compiler generator,Journal of LISP and Symbolic Computation, 1988.

[Kawamura 91] T. Kawamura, *Derivation of Efficient Logic Programs by Synthesizing New Predicates.* Proc. of 1991 International Logic Programming Symposium, pp.611 – 625, San Diego, 1991.

[Koshimura *et al.* 91] M. Koshimura, H. Fujita and R. Hasegawa,
*Utilities for Meta-Programming in KL1.*
In Proc. of KL1 Programming Workshop'91, ICOT, 1991 (in Japanese).

[Maher 87] M. J. Maher, *Logic semantics for a class of committed-choice programs.* In Proc. of the 4th Int. Conf. on Logic Programming, MIT Press, 1987.

[MantheyBry 88] R. Manthey and F. Bry, *SATCHMO: A Theorem Prover Implemented in Prolog.* In Proc. of CADE-88, Argonne, Illinois, 1988.

[Matsumoto *et al.* 83] Yuji Matsumoto, H. Tanaka, H. Hirakawa, H. Miyoshi and H. Yasukawa, BUP: A Bottom-up Parser Embedded in Prolog, *New Generation Computing,* Vol. 1, 1983.

[Morita *et al.* 90] Y. Morita, H. Haniuda and K. Yokota, *Object Identity in Quixote.* Technical Report TR-601, ICOT, 1990.

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

[OhsugaSakai 91] A. Ohsuga and K. Sakai, Metis: A Term Rewriting System Generator. In *Software Science and Engineering,* I. Nakata and M. Hagiya (eds.), World Scientific, 1991.

[OkumuraMatsumoto 87] Akira Okumura and Yuji Matsumoto, Parallel Programming with Layered Streams, In *Proc. 1987 International Symposium on Logic Programming,* pp. 224-232, San Francisco, September 1987.

[Plotkin 70] G. D. Plotkin, *A note on inductive generalization.* In B. Meltzer and D. Michie (eds.), Machine Intelligence 5, 1970.

[Poole *et al.* 87] D. Poole, R. Goebel and R. Aleliunas, Theorist: A logical Reasoning System for Defaults and Diagnosis, N. Cercone and G. McCalla (eds.), *The Knowledge Frontier: Essays in the Representation of Knowledge,* Springer-Verlag, pp.331–352 (1987).

[SakaiAiba 89] K. Sakai and A. Aiba, *CAL: A Theoretical Background of Constraint Logic Programming and its Applications.* J. Symbolic Computation, Vol.8, No.6, pp.589-603, 1989.

[Saraswat 89] V. Saraswat, *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, Computer Science Department, 1989.

[SatohIwayama 92] K. Satoh and N. Iwayama, A Correct Top-down Proof Procedure for a General Logic Program with Integrity Constraints. In *Proc. of the 3rd International Workshop on Extensions of Logic Programming,* E. Lamma and P. Mello (eds.), Facalta di Ingegneria, Universita di Bologna, Italy, 1992.

[SekiFurukawa 87] H. Seki and K. Furukawa, *Notes on Transformation techniques for Generate and Test Logic Programs.* In Proc. 1987 Symposium on Logic Programming, IEEE Computer Society Press, 1987.

[Shapiro 83] E. Y. Shapiro, *A Subset of Concurrent Prolog and Its Interpreter.* Tech. Report TR-003, Institute for New Generation Computer Technology, Tokyo, 1983.

[Sugimura *et al.* 88] R. Sugimura, K. Hasida, K. Akasaka, K. Hatano, Y. Kubo, T. Okunishi, and T. Takizuka, A Software Environment for Research into Discourse Understanding Systems. In *Proc. of the International Conf. on Fifth Generation Computing Systems 1988,* Tokyo, 1988.

[TakeuchiFurukawa 86] A. Takeuchi and K. Furukawa, *Partial Evaluation of Prolog Programs*

*and Its Application to Meta Programming.* In Proc. IFIP'86, North-Holland, 1986.

[Taki 88] K. Taki, *The Parallel Software Research and Development Tool: Multi-PSI system.* In Programming of Future Generation Computers, K. Fuchi and M. Nivat (eds.), North-Holland, 1988.

[Taki 89] K. Taki, *The FGCS Computing Architecture.* In Proc. IFIP'89, North-Holland, 1989.

[TanakaYoshioka 88] Y. Tanaka, and T. Yoshioka, Overview of the Dictionary and Lexical Knowledge Base Research. In *Proc. FGCS'88*, Tokyo, 1988.

[Tsuda 92] H. Tsuda, cu-Prolog for Constraint-based Grammar. *In Proc. of the International Conf. on Fifth Generation Computer Systems 1992*, Tokyo, 1992.

[Ueda 86a] K. Ueda, *Guarded Horn Clauses.* In Logic Programming '85, E. Wada (ed.), Lecture Notes in Computer Science, 221, Springer-Verlag, 1986.

[Ueda 86b] K. Ueda, *Making Exhaustive Search Programs Deterministic.* In Proc. of the Third Int. Conf. on Logic Programming, Springer-Verlag, 1986.

[UedaChikayama 90] K. Ueda and T. Chikayama, *Design of the Kernel Language for the Parallel Inference Machine.* The Computer Journal, Vol. 33, No. 6, pp. 494–500, 1990.

[Warren 83] D. H. D. Warren, *An Abstract Prolog Instruction Set.* Technical Note 304, Artificial Intelligence Center, SRI, 1983.

[YasukawaYokota 90] H. Yasukawa and K. Yokota, *Labeled Graphs as Semantics of Objects.* Technical Report TR-600, ICOT, 1990.

[Yokota 88a] K. Yokota, *Deductive Approach for Nested Relations.* In Programming of Future Generation Computers II, K. Fuchi and L. Kott (eds.), North-Holland, 1988.

[Yokota *et al.* 88b] K. Yokota, M. Kawamura and A. Kanaegami, Overview of the Knowledge Base Management System(KAPPA). In *Proc. of the International Conf. on Fifth Generation Computing Systems 1988,* Tokyo, 1988.