GUARDED HORN CLAUSES

Kazunori Ueda

March 1986

GUARDED HORN CLAUSES

by

Kazunori Ueda

A thesis submitted to
the Information Engineering Course
of
the University of Tokyo, Graduate School
in partial fulfillment of the Requirements for
the Degree of

Doctor of Engineering

March 1986

ABSTRACT

This thesis introduces the programming language Guarded Horn Clauses which is abbreviated to GHC. Guarded Horn Clauses was born from the examination of existing logic programming languages and logic programming in general, with special attention paid to parallelism.

The main feature of GHC is its extreme simplicity compared with the other parallel programming languages. GHC is a restriction of a resolution-based theorem prover for Horn-clause sentences. The restriction has two aspects: One is the restriction on data flow caused by unification, and the other is the introduction of choice nondeterminism. The former is essential for a general-purpose language and it also provides GHC with a synchronization primitive. The latter is required by the intended applications which include a system interacting with the outside world. What is characteristic with GHC is that all the restrictions have been imposed as the semantics given to the sole additional syntactic construct, guard.

Although Guarded Horn Clauses can be classified into the family of logic programming languages, it has close relationship to other formalisms including dataflow languages, Communicating Sequential Processes, and functional languages for multiprocessing. Except for the lack of higher-order facilities, GHC can be viewed as a generalization of these frameworks. The simplicity and generality of GHC will make it suitable for a standard not only of parallel logic programming languages but of parallel programming languages. Moreover, it is simple enough to be regarded as a computation model as well as a programming language.

Attention has always been paid to the possibility of efficient implementation during the design stage of GHC. We showed that stream merging and distribution which are expected to be heavily used can be implemented with the same time-complexity as the time-complexity of many-to-one communication in procedural languages. Furthermore, we made available an efficient compiler-based implementation of a subset of GHC on top of Prolog.

GHC has lost the completeness as a theorem prover deliberately, not as a result of compromise. Nevertheless, it can be used for efficient implementation of exhaustive solution search for Horn-clause programs. We showed how to automatically compile a Horn-clause program for exhaustive search into a GHC program.

ACKNOWLEDGMENTS

The author would like to express his deepest gratitude to his supervisor, Eiiti Wada for his continuous encouragements. He would also like thank the judging committee members Setsuo Ohsuga, Hirochika Inoue, Hidehiko Tanaka and Masaru Kitsuregawa, for their valuable suggestions.

The author joined the logic programming community through the research with Hideyuki Nakashima and Satoru Tomura at the University of Tokyo in 1982. The motivation for this thesis goes back to the unsolved problems in that research. Intensive discussions with them and with all the other colleagues of the author at the University of Tokyo were always valuable for his learning on programming and programming languages. In addition, the author learned a methodology of reviewing programming languages through the detailed examination of the preliminary Ada* language with the members of the Tokyo Study Group.

The author joined NEC Corporation in 1983 and started to work with the members of ICOT (Institute for New Generation Computer Technology). Yasuo Katoh, Katsuya Hakozaki, Masahiro Yamamoto, Kazuhiro Fuchi and Koichi Furukawa provided him with a very stimulating place in which to work, and gave him continuous encouragements. The content of the thesis is greatly benefited and reinforced by daily discussions with Akikazu Takeuchi, Toshihiko Miyazaki, Jiro Tanaka, Koichi Furukawa, Takashi Chikayama, Rikio Onai and other ICOT members, as well as discussions with the members of ICOT Working Groups and the KL1 (Kernel Language version 1) Task Group. Discussions with researchers feeling uneasy about parallel logic programming, as well as with those advocating it, were very helpful. This thesis was set using TEX82 installed by Shigeyuki Takagi.

Discussions with the visiting researchers to ICOT from all over the world were always stimulating and also useful. In particular, Ehud Shapiro, Keith Clark and Steve Gregory gave him invaluable suggestions on the design of programming languages. The thesis is also benefited by discussions with foreign researchers via electric and hard mails.

Martin Nilsson, Takafumi Saito, and Keiichi Kaneko made helpful comments on the earlier version of the thesis.

This research was done as part of the R&D activities of the Fifth Generation Computer Systems Project of Japan.

Finally, the author would like to thank his parents and his wife for their devoted support.

* Ada is a registered trademark of the U. S. Government.

CONTENTS

Chapter 1
INTRODUCTION

1.1. Motivation

Logic has attracted attention as a programming language since 1970's. The first logic programming language Prolog, based on Horn-clause logic, was quite successful; it is still the most important logic programming language in practice. The implementation technique of Prolog has been studied very well, and compiler-based systems have been and are being developed for many computers.

However, although important, Prolog should be considered an approximation to the ultimate logic programming language; we must continue to search for better approximations. Actual Prolog systems have a variety of extralogical concepts such as sequential control, a cut operator, database update and input/output by side effects, and system predicates sensitive to the current instantiation of a variable. These extralogical facilities must have been introduced because they were necessary and/or useful. However, we must examine whether they are truly essential and whether they can be introduced into the framework of logic programming in a cleaner way.

Of the above extralogical concepts, the problem of sequential control is closely related also to the performance of execution. Several researchers made attempts to design logic programming languages for parallel execution. Of these, Relational Language and its successor PARLOG by Clark and Gregory [1981][1984a] and Concurrent Prolog by Shapiro [1983a] contributed very much to the practice of parallel logic programming by showing many non-trivial program examples. However, many of the proposed parallel logic programming languages were developed for the purpose of alleviating the restriction of sequential control, and did not go so far as to exclude any inessential sequentiality.

Therefore, it is interesting to know how much concurrency we can exploit conceptually and practically, and to design a language not by extending the semantics of Prolog which is a special proof procedure suited for sequential execution, but by restricting more general proof procedures for Horn clauses. The thesis is mainly motivated by this interest. Since the solution seemed to lie not far from PARLOG and Concurrent Prolog, we judged that the thorough examination of these languages should be useful for designing a new language. These languages were defined informally, so it seemed effective to take an examination method similar to that employed when a part of the preliminary version of Ada was thoroughly examined by Ueda [1982].

Another motivation of this research is the desire to promote from practical aspects parallel logic programming founded by Clark, Gregory, Shapiro and others.

Not a few people doubted whether languages like Concurrent Prolog and PARLOG could be used for efficient execution of non-trivial programs, so we must show an efficient implementation. Not a few people disliked Concurrent Prolog since it abandoned the facilities to obtain all answer substitutions for a given goal which they thought a unique and indispensable feature of logic programming. We must give them a persuasive answer that search problems can be efficiently solved in a parallel logic programming language without any dedicated feature for exhaustive search. Research on implementation and applicability of a programming language is no less important than the design of language features, especially for the prevalence of the language.

1.2. Objectives

As stated in the previous section, our main objective is to examine the current practice of logic programming and logic programming languages and to propose a better alternative to the existing languages. We stick to the framework of resolution-based theorem proving for a set of Horn clauses, though it may not be the only framework of logic programming. This framework has been by far the best studied and put into practice extensively, so a good proposal would have great influence on the logic programming community. The language we propose is called Guarded Horn Clauses for its appearance; Guarded Horn Clauses will be abbreviated to GHC in the sequel.

Special interest and consideration are given to parallelism. In fact, parallelism is one of the most important design principles of GHC. Employing parallelism as a principle of language design could be justified by the fact that a formula in the classical first-order logic expresses no sequentiality.

Another objective is to justify the proposed language GHC from a practical point of view. It is a crucial matter for a practical language to have an efficient running implementation. Even sequential implementation should be useful to put the programming in GHC into practice and to make it widespread. Sequential implementation is more than a prototype, since it can be used to form an interactive system which communicates with human beings and other peripheral devices, in which case stream communication of GHC enables much more natural programming than the side-effective I/O of Prolog. Of course, we must not be satisfied with sequential implementation for a parallel language. Parallel implementation, however, is one of the main objectives of the Japanese 5G project which began in 1982 and will last 10 years; it is truly a far-reaching goal. Therefore, the best we can demonstrate now is to show that GHC can be implemented efficiently at least on a sequential computer, and we will show how we can compile a GHC program into an efficient Prolog program.

Expressive power of the language is also an important factor for practicality. In order for the language to be accepted by people in the other communities, we have

to show how the practices in those communities can be exercised in our framework. Such important practices will include many-to-one communication and the use of mutable (rewritable) objects in procedural programming, and exhaustive search in logic programming; we show how they can be exercised in the GHC framework.

Several remarks will be helpful for clarifying the purpose of this research and the method taken.

(1) We examine logic programming as a framework for general, practical programming rather than for problem solving. Particularly, we regard fully-controllable input/output facilities as an essential feature for a general programming language.

(2) We regard the basic framework and the primitives of a language as of primary importance, and user-oriented facilities as secondary. We are interested in what is the result of computation and what kind of delay is allowed in computation, rather than what parentheses should be used for grouping entities.

(3) Our approach is not very formal. We know that formality is necessary and useful for many aspects of programming and programming languages. However, we know also that informal consideration can be very useful as well.

Items (1) and (2) suggest that our principles for language design are quite different from that of Nakashima [1983a] when he proposed a language Prolog/KR as a knowledge representation system. While Nakashima's interest was in how the fundamental concepts emerged from artificial intelligence research can be dealt with in the logic programming framework, our main interest is to extract a minimal set of concepts which is truly essential for a programming language which allows parallelism.

1.3. Contributions

The contributions of the research in this thesis are as follows:

(1) We examined existing logic programming languages, and proposed a simple and general language Guarded Horn Clauses by introducing a notion of dataflow restriction into the logic programming framework.

(2) We showed that the computational model provided by GHC is uniform and general enough to express most notions appearing in the existing computational models.

(3) We provided a starting point of designing more specialized but more efficient languages for applications in which performance is more important than flexibility.

(4) We showed the viability of GHC as a practical language as well as a conceptual language by demonstrating that it allows efficient implementation at least on a sequential computer.

(5) We demonstrated that exhaustive solution search in the usual framework of logic programming can be done efficiently by using GHC as a base language. The method proposes a much simpler alternative to the dedicated OR-parallel machine approach.

Guarded Horn Clauses is going to provide a basis of the Kernel Language Version 1 (KL1) for multi-SIM (Sequential Inference Machine) and PIM (Parallel Inference Machine) being developed at Institute for New Generation Computer Technology (ICOT).

1.4. Structure of the Thesis

This thesis is organized as follows. Chapter 2 reviews logic programming in the original sense to provide a foundation of the investigation of parallel logic programming in the subsequent chapters. It then reviews the *practice* of logic programming which in part deviates from the basic framework. The review includes the examination of the extralogical features in Prolog some of which will be resolved in GHC. Chapter 3 introduces parallel logic programming. After giving some motivations, it discusses parallel execution of logic programs and clarifies the problems in OR-parallelism and AND-parallelism. Then it surveys previous works on parallel execution of logic programs and proposals of parallel logic programming languages for controlled AND-parallelism. Of those proposals, we select Concurrent Prolog for detailed examination and points out some problems which will be resolved in GHC.

Chapter 4 introduces Guarded Horn Clauses. It first gives the design principles of GHC and then describes its syntax and semantics. Several program examples follow. Then it discusses the primitive operations of GHC. It shows how elegantly the idea of process interpretation of logic applies to GHC. Finally, it reviews the design of GHC with justification and mentions possible extensions of the language and implementation issues. Chapter 5 compares GHC with other languages and computational models related to parallel and/or logic programming, and clarifies that GHC gives a uniform and general framework for parallel computation.

Chapter 6 discusses two implementation issues. Firstly, it reinforces the viability of GHC as a process description language by showing that the time-complexities of many-to-one and one-to-many communication and of the operations on mutable arrays are no worse than those of procedural languages on conventional sequential computers. Then, it describes an efficient implementation of a subset of GHC and Concurrent Prolog on top of Prolog. Chapter 7 gives an answer to the applicability of GHC for search problems. It shows that a vast class of Horn-clause programs for exhaustive solution search can be compiled into GHC and runs efficiently. Finally, Chapter 8 summarizes the thesis and gives a prospect for the future.

Chapter 2
LOGIC PROGRAMMING

In this chapter we first review the basic concepts of logic programming in the original sense, and we clarify the advantages of the framework by comparing it with the framework of procedural languages. Then we consider the logic programming framework from the programming language point of view rather than a theorem prover point of view, which will provide a different view of logic programming. The purpose of this consideration is to see the viability of logic programming languages as a general language rather than a special-purpose language.

Next, we examine the current practice of Prolog programming which is different from the programming in pure Horn-clause logic in many points, and clarify what extralogical notions are used in it.

As we stated earlier, we restrict our scope of interest to resolution-based theorem provers for a set of Horn clauses throughout the thesis; logic programming in the other senses is not considered.

2.1. Basic Concepts of Logic Programming

We introduce the basic framework of logic programming founded by Kowalski [1974], van Emden and Kowalski [1976], Clark [1979], Apt and van Emden [1982] and others, as well as many preceding works on automated theorem proving. Since similar introduction can be found in other documents, for example in the book by Lloyd [1984], we do not pursue generality of the exposition but state only what is needed to make the thesis self-contained.

Note that this section introduces logic programming as a special and simple case of theorem proving suitable for a basic framework of a programming language. The more practical aspects of logic programming will be discussed in detail in Sections 2.2 to 2.4.

2.1.1. Syntax of Logic Programs

We begin by defining syntactic constructs. We use the following classes of symbols which can vary from program to program:

(1) variables

(2) function symbols

(3) predicate symbols

A predicate symbol is also called a *predicate* or a *relation* when its semantics is in question. In this thesis, we begin variables with capital alphabetical letters, and

function and predicate symbols with small alphabetical letters. We also use the letter '_' to denote an anonymous variable. Each occurrence of '_' denotes a variable distinct from any other variables; that is, two occurrences of '_' denote distinct variables. Moreover, we use combinations of non-alphabetical and alphabetical symbols for function and predicate symbols; for example, '`1`' will be used as function symbols and '`:=`' and '`$END`' as predicate symbols in this thesis.

Each function or predicate symbol has its *arity* as a property, which is an integer value indicating how many arguments follow that symbol. A function symbol with the arity 0 is called a *constant symbol*.

Function and predicate symbols may *overload*, that is, a function symbol and a predicate symbol may be literally identical and two function or predicate symbols with different arities may be literally identical. When it is necessary to avoid confusion, we use the notation f/n where f is a function or predicate symbol and n is its arity.

In addition to the above symbols, we use logical connectives and punctuation symbols, whose meanings will be explained where they appear.

A *term* is either

(1) a variable, or

(2) an n-ary function symbol followed by n terms, where the terms, if any (i.e., if $n > 0$), are separated by commas and surrounded by parentheses.

When a term T is of the form $f(T_1, \ldots, T_n)$, f is called the *principal function symbol* of T and T_i's are called its *arguments*. A term which contains no variables is called a *ground term*. A term which is not a ground term is called a *non-ground term*. A non-ground term is sometimes called an *incomplete data structure* since it contains undetermined parts.

An *atomic formula*, also called an *atom*, is an n-ary predicate symbol followed by n terms, where the terms, if any, are separated by commas and surrounded by parentheses.

An *expression* is either a term or an atomic formula.

A *definite clause*, also called a *program clause*, is of the form

$$\forall(H \subset B_1 \wedge \ldots \wedge B_n), \qquad n \geq 0$$

where H and B_i's are atoms. The symbol $\forall$ followed by no quantified variable indicates that all the variables appearing free in H and B_i's are universally quantified. A program clause with $n = 0$ is called a *unit clause*, where an empty set of B_i's are understood as denoting *true*, a unit of conjunction. H is called a *clause head* or simply a *head*; a set of B_i's are called a *clause body* or simply a *body*. Each B_i is called a *(body) goal*.

A *goal clause* is of the form

$$\forall\neg(G_1 \wedge \ldots \wedge G_n), \qquad n \geq 0$$

where G_i's are atoms. Each G_i is called a *goal*. A goal clause with $n = 0$ is called an *empty clause* which are understood as denoting contradiction.

A *Horn clause*, also simply called a *clause* in this thesis, is either a definite clause or a goal clause. A goal appearing in a Horn clause is said to *belong to* that clause.

A *logic program*, also simply called a *program* in this thesis, is a conjunction of definite clauses. A logic program is sometimes called a *world* or an *(internal) database*, according to the intended interpretation. A conjunction of definite clauses whose heads have the same predicate symbol is called a *procedure*. Clauses in a procedure are called *sibling clauses*.

Our framework is based on first-order predicate logic, but we use only its subset; the formulae we handle are limited to Horn clauses and their conjunctions. This subset of first-order logic is called *Horn-clause logic*. We are interested in this subset for the following reasons:

(1) A definite clause allows a procedural interpretation which is analogous to procedures in conventional languages.

(2) For this subset, we have a sound and complete proof procedure with respect to its declarative semantics (see Section 2.1.4) which is practical and efficient compared with the sound and complete proof procedures for larger subsets of first-order logic.

For notational convenience, hereafter we use the following syntactic conventions: A definite clause is written as

$$H \texttt{ :- } B_1\texttt{, } \ldots\texttt{, } B_n\texttt{.}$$

and when $n = 0$, it is abbreviated to

$$H\texttt{.}$$

A goal clause is written as

$$\texttt{:- } G_1\texttt{, } \ldots\texttt{, } G_n\texttt{.}$$

and when $n = 0$, it is also written as $\Box$.

The connective '`:-`' reads as "is implied by", and the connective '`,`' denotes conjunction. The variables occurring in a clause are understood as universally quantified.

We prepare special notations for lists: We write the constant '`nil`' as '`[]`', and '`cons(`A`,`B`)`' as '`[`A`|`B`]`', where A and B are terms. Moreover, we write

the structure '`[`A`|[`*something*`]]`' as '`[`A`,` *something*`]`', where A is a term and *something* is a string of symbols which is balanced with respect to square brackets.

We sometimes use function and predicate symbols as operators. A unary function/predicate symbol may be used as a prefix or postfix operator, and a binary function/predicate symbol may be used as an infix operator.

2.1.2. Substitution and Unification

Next, we give a framework for assigning values to the variables in a program.

A *substitution* is a mapping from a set of variables to a set of terms. Usually this mapping acts as an identity except for finitely many variables. We write substitutions as a postfix operator throughout the thesis. A substitution θ can be naturally extended to a mapping from a set of expressions to a set of expressions; $T\theta$ is an expression obtained by simultaneously replacing the all the occurrences of variables in T according to the mapping θ. Here, the expression $T\theta$ is called an *instance* of T. A substitution θ can be represented by a set of *bindings* of the form T/V, where V is a variable and T denotes $V\theta$ such that $V\theta \neq V$. We say that a substitution θ *binds* a variable V to a term T if $T = V\theta \neq V$.

We say that an expression T_1 is a *variant* of an expression T_2 if T_1 is an instance of T_2 and T_2 is an instance of T_1. A term `p(X,Z)` is a variant of `p(X,Y)`, `p(Y,X)` is a variant of `p(X,Y)`, but `p(X,X)` is not a variant of `p(X,Y)`. A *new variant* U of an expression T is informally defined as a variant of T such that all the variables in U are distinct from any variables that have been used in the course of discussion or computation before U is created.

An expression E is said to be *instantiated* by a substitution θ if $E\theta$ is not a variant of E. As a special case, a variable V is said to be *instantiated* by a substitution θ if θ binds V to a *non-variable* term.

We say that a substitution θ is *at least as general as* a substitution σ if there is a substitution τ such that $\sigma = \theta\tau$, that is, if for any expression T, $T\sigma$ is an instance of $T\theta$. We say that a substitution θ is *as general as* a substitution σ if θ is at least as general σ and vice versa, that is, if for any expression T, $T\theta$ is a variant of $T\sigma$. We say that a substitution θ is the most general in the set S of substitutions if θ is at least as general as any substitutions in S.

Now we can define unification. A substitution θ is said to *unify* two expressions T and U, or solve the equation $T = U$, if $T\theta$ and $U\theta$ are identical. Such a substitution is called a *unifier* of T and U. A unifier θ of T and U is called the *most general unifier* if θ is the most general in the set of all unifiers of T and U. A unification between T and U is said to *bind* a variable V to a term W if their most general unifier maps V to W.

2.1.3. Procedural Semantics

The procedural (or operational) semantics of a logic program is identified with a proof procedure of a goal clause with respect to the program. We use SLD-resolution (linear resolution with selection function for definite clauses; see Apt and van Emden [1982]) in the proof procedure described below; this is the only strategy we consider until we introduce parallelism in Chapter 3.

Given a goal clause "$:- G_1, \ldots, G_n.$", the proof procedure tries to derive from it an empty clause, which means contradiction, using the clauses in the program. If this is successful, we say that the original goal clause has been *refuted*.

To make a refutation, we use SLD-resolution. SLD-resolution is the following derivation rule for rewriting a goal clause to another:

Let the goal clause to be rewritten be

$$:- G_1, \ldots, G_k, \ldots, G_n.$$

where G_k is the *selected* atom under the selection rule employed. Then find a new variant of a program clause

$$H :- B_1, \ldots, B_m. \qquad \text{or} \qquad H.$$

such that G_k and H are unifiable with the most general unifier σ, and rewrite the goal clause to

$$:- G_1\sigma, \ldots, G_{k-1}\sigma, B_1\sigma, \ldots, B_m\sigma, G_{k+1}\sigma, \ldots, G_n\sigma.$$

We call the above rewriting a *derivation step*, a *reduction step*, or a *logical inference*.

SLD-resolution is a kind of linear input resolution (Chang and Lee [1973]). Note that there may be more than one program clause that can be used for each derivation step. Thus, although the proof procedure is called linear since it always uses the latest goal clause to make a new one, the *behavior* of the proof procedure may not be linear; it involves *searching* to find a refutation. We can conceive a possibly infinite search tree called a *derivation tree*

(1) whose root corresponds to the original goal clause,

(2) whose nodes (except the root) correspond to the goal clauses rewritten by derivation steps, and

(3) whose arcs correspond to the program clauses used for deriving the goal clauses corresponding to the son nodes from the goal clauses corresponding to the parent clauses,

under some selection rule of atoms. In addition to a goal clause, each arc is given the most general unifier obtained in the derivation step it represents. A path from

the root ending with an empty clause expresses refutation. For each refutation path on which substitutions $\theta_1, \ldots, \theta_k$ appear in this order, we call the composition $\theta_1 \ldots \theta_k$ the *answer substitution* of this refutation. In logic programming, we usually consider answer substitutions as the result of computation, though this point will be re-examined in Section 2.3.

We can say that the task of the proof procedure is to find refutation paths in a derivation tree and to obtain corresponding answer substitutions. Note that not all the paths in a derivation tree lead to an empty clause; some may be infinite and others may end with an non-empty clause which allows no further derivation.

There are many search strategies as well as selection rules for atoms a proof procedure can take to find a refutation. A *depth-first* proof procedure uses the strategy that a goal clause derived most recently is always used for deriving a new goal clause whenever it is possible. A proof procedure is said to be *fair* if it finds all refutation paths in a finite time.

We must note that the above formalism of theorem proving gives little consideration to parallel execution like most literature on theorem proving. We will discuss in Chapter 3 what kind of parallelism we can exploit from the above formalism.

2.1.4. Declarative Semantics

In Section 2.1.3, we gave a proof procedure for logic programs to obtain answer substitutions for a goal clause. We must further make sure how the answer substitutions are meaningful. As long as the proof procedure cannot be generally accepted as the basic computational model, we must resort to something considered more fundamental in order to clarify the meaning of the result of computation. The most appropriate for such a foundation is first-order logic. We briefly introduce the results showing that computed answer substitutions are *correct* in an agreeable sense. These results are mainly due to Clark [1979].

We denote a logic program by P and a goal clause of the form "`:- `G_1`, ... ,`G_m`.`" by G. An answer substitution θ is said to be *correct* if $\forall(G_1\theta \wedge \ldots \wedge G_m\theta)$ is a logical consequence of P. A closed formula (i.e., a logical formula without free occurrences of variables) F is said to be a *logical consequence* of another closed formula G if F is interpreted as true in every model of G. Then, the following properties hold for the proof procedure described in Section 2.1.3:

(1) (*soundness*) An answer substitution obtained from the proof procedure is correct.

(2) (*completeness*) For any *correct* answer substitution θ and under a given selection rule of atoms, a fair proof procedure computes a substitution σ at least as general as θ.

We can further capture the semantics of a logic program P by means of the specific model of the program called the minimal Herbrand model M_P. We do

not go into detail but only note the important properties of the minimal Herbrand model, which is represented by the set of all elements of the Herbrand base B_P of P which are interpreted as true in the model.

(1) M_P is a set of all elements of the Herbrand base of P which are logical consequences of P.

(2) M_P is a least fixpoint of the mapping T_P defined as

$$T_P(I) = \{H \mid \text{there is a } \textit{ground instance } H \text{ :- } B_1, \ldots, B_n \text{ of a clause of } P \text{ such that } B_i \in I \text{ for } i = 1, \ldots, n\},$$

where a ground instance of a clause is an instance obtained by applying a substitution mapping all the variables in it to the elements of B_P.

The minimal Herbrand model is important at least in the following three points:

(1) It is a general and free model in that it effectively gives no interpretations to function symbols.

(2) It says just what are the ground logical consequences of P.

(3) We can enumerate its elements.

For some applications, there may be programs that never terminate but are still useful by being equipped with an appropriate means to observe and/or to control the result of computation. Clearly we cannot give the semantics in terms of the 'postmortem' answer substitutions for such programs. Some people try to capture the semantics of non-terminating programs by extending Herbrand models to allow infinite atoms (van Emden and de Lucena [1982], Hagiya [1983], Lloyd [1984]). They show that the semantics of a non-terminating logic program can be characterized by the greatest fixpoint of the extended (or complete) Herbrand base.

2.2. Advantages of the Logic Programming Framework

The following can be pointed out as advantages of the logic programming framework over the framework provided by conventional procedural languages:

(1) *Non-Strict Data Structures*

Since a variable in a logic programming language is 'monotonic', we have a natural notion of the value of some variable being 'unknown' or 'undefined yet'. By the word 'monotonic' we mean that the knowledge about the value of some variable may *increase* as computation proceeds but that it never *changes*. Such a variable could also be called a single-assignment variable since its top-level value (i.e., its principal function symbol) can be determined at most once. Moreover, we can handle a data structure which may contain uninstantiated

variables in a natural way. Such a data structure is called *non-strict* or *incomplete*.

By contrast, in many procedural languages the 'undefined' state of a variable is regarded as exceptional rather than one of possible values, despite the fact that every variable is initially undefined. In order to disallow referring to the value of an undefined variable, these languages impose awkward language rules whose violation is not usually detected at compile time or at run time.

The capability to handle incomplete data structures is effectively used in programming with difference lists (Clark and Tärnlund [1977]). A difference list is a pair of some list called *head* and its sublist called *tail*. By leaving its tail undefined, a difference list can be used as a part to construct a longer difference list or a complete list terminating with '`nil`'. The notion of difference lists provides us with the freedom of the order in which to determine each portion of a concrete list structure.

(2) *Unification Replaces Assignment, Parameter Passing, and Equality Check*

Procedural languages use assignment and a couple of parameter passing mechanisms to transfer data. In logic programming languages, these are achieved by one mechanism, unification. The semantics of assignment is awkward in many theoretical models, and unification has much cleaner semantics. Moreover, procedural languages should provide a separate primitive for checking the equality of two values, while in logic programming unification can be used also for equality check.

(3) *A Program Looks Like a Naive Logical Specification*

Any programming language may have its own logic. For example, a procedural language could be formalized by Hoare logic (Hoare [1969]). However, we do not call it a logic programming language. So what characterizes a logic programming language should be that it is based on a naive classical logic which is originated as a tool for formal treatment of human inference rather than a logic dedicated for computer languages. Classical logic has been used as a fundamental tool to describe the specification of procedural programs (Dijkstra [1975]). Using the same specification, however, we can derive a logic program more easily, thanks to the smaller semantic gap.

The following are also often claimed as advantages of logic programming. However, we reserve our judgment and examine these facilities closer in Section 2.3.

(1) *Clarity of Semantics*

As mentioned in Section 2.1, the semantics of a logic program is characterized by the least Herbrand model, and SLD-resolution has been proved to be a

sound and complete proof procedure with respect to the least Herbrand model. Thus logic programming indeed has a simple and clear model-theoretic and operational semantics in the framework of theorem proving. However, this does not necessarily mean that it has a clear semantics when it is amended to a general-purpose programming language.

(2) *Automatic Backtracking and Exhaustive Search*

The completeness of the proof procedure, or the capability of obtaining all solutions (answer substitutions) of a given goal is often convenient and sometimes regarded as a unique feature of logic programming. However, there are also many situations in which we do not want all the solutions but only one.

(3) *Multiple Uses of Predicates*

The capability to use a single predicate definition in two or more modes of information flow is also claimed as a convenient feature. For example, an '`append`' program can be used for decomposing a list into two sublists and for checking if some list is a concatenation of two other lists, in addition to the normal use suggested by the verb 'append'. In many cases, however, we have only one intended mode for each predicate we define.

2.3. Closer Look at Logic Programming As a General Programming Language

This and the next sections examine the characteristics of logic programming in more detail. Emphasis will be placed on the characteristics of logic programming as a framework for general programming. Of course, general programming includes specific areas such as problem solving and knowledge processing which are usually considered main targets of logic programming and which are really important. The point is that we examine logic programming from a programming language point of view rather than from specific application areas. Since Prolog is gaining more and more application areas and becoming a general-purpose programming language, it should be meaningful to examine the practice of Prolog programming and clarify how it is different from logic programming in the pure sense as described in Section 2.1.

This section gives general observations, and Section 2.4 enumerates and examines the extralogical features of Prolog.

2.3.1. What Is the Virtue of Being Logical?

As we stated in Section 2.2, what characterizes a logic programming language is that it is based on a naive classical logic originated as a tool for formal treatment of human inference rather than a logic dedicated for computer languages. Nevertheless, as we will see later, an actual programming language should be more than a simple

theorem prover because we have many requirements for an actual programming language which a simple theorem prover cannot fulfill. Therefore, we must anyway devise less familiar and more complicated logic to formally describe more aspects of a programming language. This, however, does not mean that naive classical logic loses its role in the practice of logic programming: Simple logic is still useful in programming, in reasoning about programs, and so on.

2.3.2. On Completeness and Multiple Uses of a Predicate

Completeness is highly respected as a desirable property of a proof procedure. In logic programming or Prolog programming also, we often make use of this property. However, we can also observe the following:

(1) Due to the depth-first search strategy of Prolog implementation, it is far more difficult to write a Prolog predicate as a fair generator of solutions than to write a predicate to be used only for checking. We often write a trial-and-error program for exhaustive search in a trial-and-error manner. Unless it is necessary, we seldom write programs within the subclass of Horn-clause logic for which the depth-first proof procedure is complete.

(2) The capability of computing all solutions and the capability of using some predicate in more than one mode of information flow are independent. In many cases, each goal appearing in a program has only one intended mode. For example, the following Prolog program is often used as an example of naive programs:

```
sort(X,Y) :- permutation(X,Y), ordered(Y).
```

However, even the most naive programmer would not use '`ordered`' as a generator in the following way,

```
sort(X,Y) :- ordered(Y), permutation(X,Y).
```

since this would render the definition of '`ordered`' much more complicated, and even if he succeeds in defining it, the whole program would not run as efficiently. This example shows the importance of the notion of data flow.

(3) We are not always interested in all solutions of a goal. We cannot, however, express this within the original framework of logic. This is why we often rely on some control facilities such as a cut operator (see Section 2.4.1).

(4) The original framework of logic programming allows us to 'search' all solutions of a goal exhaustively, but it never means that the obtained solutions can be 'collected' together within the framework (see Sections 2.3.3, 2.4 and Chapter 7).

2.3.3. What Is the Result of Execution of a Logic Program?

It is often claimed (e.g. by Lloyd [1984]) that we are interested in answer substitutions computed by a proof procedure as well as a success/failure signal it returns. However, are answer substitutions displayed by the top-level loop of a Prolog system really a standard way of seeing the result of computation? We believe they are not. The following analogy will help understand this. The procedural language counterpart of answer substitutions displayed by the system is the values of variables displayed by the postmortem dump facilities supported by most operating systems. However, postmortem dump is never a standard way of seeing the result of computation in practical programs. Perhaps it is useful only for testing small pieces of programs and for locating incomprehensible bugs in large programs.

Most non-trivial programs must communicate with its outside world by using input and output facilities. Therefore, a practical programming language must provide a capability for full control over input and output. Furthermore, since the most important medium for human-computer interaction is character strings (Nakashima, Ueda and Tomura [1983]), a self-contained language must be able to handle character strings and must be expressive enough to define conversion procedures between the internal representation of a data structure and its string representation. Prolog fulfills these requirements, but only in an awkward way. Input and output are performed by side effects, hence they completely rely on the specific control structure of Prolog. Side effect is an obstacle to verification, optimization, and parallel processing.

In general, a program is a transformer of an input data sequence to an output data sequence, and a logic program should express the relation of these two sequences in its own framework, just as it naturally expresses the relation of two internal data structure. In fact, this can be exercised if the program is not interactive; we can read all the input data first, transform them, and then write all the output data. However, this programming style cannot be used for interactive programs due to the control strategy of Prolog. This problem is discussed further in Section 3.3.2.

In addition to the input and output facilities, we must examine also the original way of seeing answer substitutions in more detail. What we get as an answer substitution from a proof procedure is a finite set of substitution components. The finiteness of the answer implies that it tells us which variables have been left uninstantiated. The important point is that such information should be considered a meta-level information, because without any extralogical facilities, a goal in a program has no means to know whether some variable is instantiated or not.

2.4. Extralogical Features in Prolog

Prolog programming and logic programming are not the same. Prolog has yielded a number of programming techniques specific to Prolog, most of which are related to its control structure based on depth-first execution and backtracking.

These techniques are a kind of tricks in that they should be of little value or even harmful from the software engineering viewpoint.

Beside the control structure, we often use extralogical facilities especially when we write a system program. In general, such extralogical features are needed when we manipulate Prolog programs as data, when we execute programs created as data, when we want some meta-level information which depends on the progress of the proof procedure, and so forth.

In this section we survey the extralogical features of Prolog, and clarify how they are used and why they are extralogical. Surveys on extralogical features in Prolog have been done also by Dömölski and Szeredi [1983] and by Kowalski [1985a], but of course from the standpoints of their own. Note that we will deal with some 'logical' features which may look extralogical as well as truly extralogical ones.

2.4.1. Cut Symbol

A cut symbol or a cut operator is used for controlling the depth-first execution of a logic program. When the proof procedure encounters a cut operator, it ignores the sibling clauses of the clause textually containing the cut operator which are not yet executed. In other words, a cut operator is used for intentionally losing the completeness of the proof procedure.

One of the most safe uses of cut will be to use it for eliminating alternatives which would return the same result. An example follows:

```
abs(X, Y) :- X >= 0, !, Y is X.
abs(X, Y) :- X =< 0, !, Y is -X.
```

We do not care which clause returns the absolute value of 0, and the both clauses return the same result. Therefore, inserting cut operators avoids unwanted backtracking without losing any solutions. Of course, if '`abs`' were used in the reverse direction to obtain `X`, the cut operator would lose possible answers, but it seems that we do not need to consider such a case.

Another use of cut will be to use it for expressing indifference to the selection of the result:

```
sign(X, Y) :- X >= 0, !, Y = '+'.
sign(X, Y) :- X =< 0, !, Y = '-'.
```

The difference from the '`abs`' example is that we do not care the sign of 0, much less the clause returning it. In this case, the use of cut seems reasonable since it prevent the proof procedure from returning two sets of answer substitutions whose difference is indifferent for us. It could be argued that the above definition is less safe than the '`abs`' example, since it fails to answer the question whether -5 and -0 can have the same sign:

```
:- sign(-5, S), sign(-0, S).
```

However, such a question seems contrary to the intent expressed by the definition of '`sign`', because it *cares* the possible signs of 0.

The above two examples show the use of cut for realizing a kind of input guard first introduced by Dijkstra [1975] in the form of guarded commands. Although the use of cut for this purpose may cause a program to deviate somewhat from its declarative semantics, it should be little problematic as long as the predicate is used in an intended direction.

Even when we have nothing we do not care, that is, even when clauses are mutually exclusive, a cut operator is sometimes used for efficiency purposes:

```
p([A|X], ... ) :- !, ...
p([],    ... ).
```

The cut operator in the first clause explicitly lets the compiler know the alternative clause need not be considered. Then the compiler can do tail-recursion optimization (Warren [1980]) to avoid unnecessary growth of the stack. For the case where the first argument of the caller of '`p`' is a non-variable term, tail-recursion optimization is possible without the cut operator. However, if a programmer does not know how sophisticated optimization the compiler can do, he or she would do defensive programming and would provide as much information for optimization as possible. A compiler of ESP (Chikayama [1984]), a logic programming language augmented with object-oriented features, even discourages one from writing a clause without cut by beeping upon encountering such a clause. Wada, Tomura, Nakashima and Kimura [1982] argue when and how they had to avoid unwanted backtracking.

Other uses of a cut operator include those which make declarative reading of a program almost meaningless. A typical example is a cut operator for expressing non-monotonicity, i.e., defaults and exceptions:

```
can_fly(penguin) :- !, fail.
can_fly(ostrich) :- !, fail.
can_fly(_).
```

Another example is a cut operator for implementing 'negation as failure' (see Section 2.4.6):

```
\+(P) :- call(P), !, fail.
\+(_).
```

These clauses allows no declarative readings, but the cut operators together with the depth-first proof strategy give them important meanings.

It must be noted that some Prolog dialects try to discourage the use of a cut operator by providing a variety of high-level primitives for backtracking control; Prolog/KR (Nakashima [1983a]) is the most eager in the elimination of the cut operator.

2.4.2. Input and Output

As mentioned in Section 2.3.3, input and output facilities of Prolog are very awkward. They use side effects and hence completely rely on the specific control strategy of Prolog. Input and output by side effect are compatible with procedural programming but quite heterogeneous in logic programming. In particular, the effects of input and output are usually not undone upon backtracking. One might argue that output is declaratively transparent since adding an output goal to some clause never changes the declarative reading of the program. However, the transparency of output means that it is *completely extralogical*: Logic says nothing about what output is obtained.

The input and output facilities of Prolog has another point to be considered. A Prolog system is able to display non-ground terms, in which each occurrence of a variable is indicated by some character string appropriately given by the system. However, the conversion of a non-ground term to a character string involves an extralogical operation to see if each of its subterms is instantiated or not.

2.4.3. Primitives for Modifying Internal Database

Most interpreter-based Prolog systems support primitives (usually called '`assert`' and '`retract`') for modifying the internal database in which program clauses are stored. In other words, the support primitives for modifying the program being used for execution. These primitives are often used for realizing a large, modifiable data structure we are familiar with in procedural languages. Although there are a number of proposals for 'logical' array facilities for Prolog (see Eriksson and Rayner [1984] and Cohen [1984], for example), many Prolog implementations still lack array and database features which allow efficient access and update within the framework of logic.

Sometimes the internal database is used also for the purpose for which even a logical array or database cannot help, that is, to pass information from some branch of a proof tree to another. Such programs rely on the illogical property of the database update primitives that their effects are not undone upon backtracking. Warren [1984] proposed a method of logical database updates in pure Prolog.

The primitive '`retract`' for deleting a clause is by no means logical since it causes non-monotonic change of the internal database. On the other hand, the primitive '`assert`' for adding a clause can be used in a safer way if we carefully restrict our use of '`assert`' to the monotonic change of the database. For example, one can add a fact which can be inferred from a current program for the purpose of optimization; this never changes the declarative semantics of the program. Another example is the addition of new facts in a knowledge acquisition system. In this case, the internal database must be considered an open world.

2.4.4. Time-Dependent Operations

A Prolog variable is deemed to have a binary state: Instantiated to a non-variable term or not. The state of a variable can change at most once during the execution from *uninstantiated* to *instantiated*. The knowledge about the state of a variable could be called a meta-level knowledge, since it depends on how the proof is performed and when it is observed. At the 'object level' (i.e., the level of the program being executed), change of the state of a variable never means that something has *changed*—what happened is just that something has become *known* which was unknown previously. Meta-level knowledge must be clearly distinguished from the object-level knowledge on the values of instantiated variables.

Nevertheless, we sometimes need to have access to the current state of a variable, for example when we implement a unification routine with occur check (Plaisted [1984]) which is not supported by most Prolog implementations. In general, we are obliged to use the meta-level knowledge when we write a system program in Prolog. The meta-level knowledge exists anyway, whether it is accessible or not. It can be confined in the implementation of Prolog only if we abandon writing system programs. Prolog systems usually provide several primitives to examine the current state of a variable: '`nonvar(`X`)`' and '`var(`X`)`' to examine if X is instantiated to a non-variable term or not, 'X`==`Y' 'X`\==`Y' to examine if X and Y are literally identical (i.e., unifiable without any substitution) or not, and so on. Nakashima, Ueda and Tomura [1984] propose higher-level meta-level primitives.

2.4.5. And-Sequentiality and Backtracking As a Control Structure

The fact that the proof procedure of Prolog performs depth-first search can be utilized for efficiency purposes. Although efficiency is an extralogical matter, it is independent of the result of computation and no problem arises. However, Prolog's specific control strategy is often utilized for guaranteeing the result of other extralogical operations such as described above.

It is often said that a Prolog program is hard to debug because the behavior of the program is difficult to understand due to backtracking. This should be addressed as a problem, since a Prolog programmer must worry to a greater or lesser degree about control issues for efficiency and/or completeness. In a program which uses side-effects, backtracking is often used for realizing repetition. The repetition would start with the goal '`repeat`' and end with '`fail`', where '`repeat`' is defined as follows:

```
repeat.
repeat :- repeat.
```

This control structure is entirely meaningless without side effects. This method is often counted as a programming technique, but it is nothing more than a trick specific to Prolog programming.

2.4.6. Call and Negation As Failure

Most Prolog systems provide a system predicate, often named '`call`', to 'execute' its argument as a goal. We can consider that the predicate '`call`' interprets the given term T appropriately and that T is possibly instantiated according to the interpretation. There should be no harm at all as long as we do not care *how* the given term is interpreted.

The idea of 'negation as failure' (Clark [1978]) is often used in Prolog. It can be defined in Prolog as:

```
\+(P) :- call(P), !, fail.
\+(P).
```

where the symbol '`\+`' was used to distinguish it from logical 'not', following DEC-10 Prolog (Bowen, Byrd, Pereira, Pereira and Warren [1983]). Note that we have used the cut operator and have relied on the specific order of clauses. It is often pointed out that the above '`\+`' is different from 'not' because if '`\+`' is a true 'not', the clause

```
q :- \+(p(X))
```

can be read as

for all `X`, `q` *if not* `p(X)`

which is equivalent to

`q` *if (for some* `X`, *not* `p(X)`*),*

while actually it works according to the interpretation

`q` *if not (for some* `X`, `p(X)`*).*

What is worse, a clause containing '`\+`' sometimes allows no declarative reading in first-order logic. For example, the extralogical predicate '`var`' can be defined as follows (Nakashima [1983b]):

```
var(X) :- \+(\+(X=1)), \+(\+(X=2)).
```

This indicates that '`\+`' itself is extralogical.

However, if the argument of '`\+`' contains no variables when it is executed, we can give the declarative semantics to this construct by using Clark's idea of completion [1978], and also there is a sound extension of SLD-resolution to handle a negative atom as a goal. The restriction that a negative atom must be ground can be weakened; even if it contains variables, there is no problem as long as its proof does not instantiate them. IC-Prolog (Clark and McCabe [1980]) guarantees the soundness by reporting an error message when a negated atom can be proved only by instantiating it. Prolog-II (Colmerauer [1982]) has a system predicate `dif(`T_1`,`T_2`)` whose success is postponed until T_1 and T_2 are proved to be ununifiable. Thus '`dif`' deals with the simplest case of logical negation.

We may have to point out that the sequentiality among sibling clauses is often used for implicit negation. For instance, the predicate '`abs`' in Section 2.4.1 tends to be defined as follows for the reason of efficiency:

```
abs(X, X) :- X >= 0, !.
abs(X, Y) :- Y is -X.
```

2.4.7. Examining and Constructing Non-variable Terms

System programmers often use facilities

(1) for extracting the principal function symbol and the arity of a non-variable term,

(2) for getting the argument of the specified position of a non-variable term, and

(3) for constructing the most general non-variable term with the specified principal function symbol.

In DEC-10 Prolog, Items (1) and (3) can be done by the system predicate '`functor`' and Item (2) by '`arg`' whose semantics should be clear from the following example:

```
:- functor(f(a,b(B),C,C), X, Y).  ⟶ X = f, Y = 4
:- functor(X, g, 3).              ⟶ X = g(_,_,_)
:- arg(2, f(a,b(B),C,C), X).      ⟶ X = b(B).
```

The function symbol '`f`' with four arguments and the symbol '`f`' with no argument are different entities which happen to have the same representation. However, we need not think those primitives as extralogical. The predicate '`functor`' can be thought of as relating a non-variable term to some constant; we could write down the definition of '`functor`' by giving one clause for each function symbol. We could also write down the definition of '`arg`'.

2.4.8. Meta-level Nature of the Result of Computation

As we examined in Section 2.3.3, the result of computation shown as possible sets of answer substitutions contains meta-level information. In front of the terminal, we get from a Prolog system a lot of extralogical information and make a lot of extralogical decisions, for example to modify the current program. This means that the task of a man at the terminal cannot easily be 'automated' (i.e., replaced by a program) without introducing meta-level features. We need an appropriate framework for meta-programming to construct programming systems.

An important application of meta-programming is knowledge assimilation and belief revision (Kowalski [1985b]) (Miyachi, Kunifuji, Kitakami, Furukawa, Takeuchi and Yokota [1984]). Meta-level extension of Prolog is discussed in (Bowen and

Weinberg [1985]) and (Bowen [1985]). Another important application is to collect all solutions of a given goal as a single data structure. This involves a meta-level operation because different solutions are originally independent. However, the capability of Prolog to obtain all solutions is an important advantage, and this advantage would be diminished if we have no means to collect all solutions for the next processing phase. Most Prolog systems therefore support primitives for collecting solutions; for example DEC-10 Prolog provides two predicates '`setof`' and '`bagof`'.

However, when we allow meta-level features, the underlying logic becomes more complex even if we have one. This means that the formal or mechanical treatment of programs would become harder. This affects not only programming systems which would support verification, debugging, and so forth, but also the efficiency of the programs since the interface between the meta level and the object level may be hard to optimize. Therefore, the use of meta-level facilities must be limited to the case where it is really necessary. For example, we show in Chapter 7 that exhaustive solution search can be done very efficiently for a non-trivial class of programs without any meta-level features.

2.4.9. Summary

To sum up, too much burden was imposed on Prolog to meet our practical needs which went far beyond pure Horn-clause logic, and as a result Prolog had a number of extralogical extensions for which only complex semantics can be given in the special control strategy of Prolog. We must review our needs and try to find a way to meet these requirements in a cleaner way.

Of course, it is not impossible to have a formal semantics of the extralogical features, because sequential Prolog is completely deterministic at meta level, that is, the proof procedure always takes exactly the same behavior for the same program. However, the fact that some feature can be defined formally does not necessarily justify that feature; we can invent a formal definition for anything as long as it is unambiguously defined informally. The important issue is how lucidly it can be defined.

The features we have to reconstruct can be divided into two: application-oriented features and general features. The former include access to meta-level information, meta-level programming, non-monotonicity, database update, and so on. Many researchers are searching for cleaner constructs to support them. However, extension of Horn-clause logic for specific applications is out of the scope of the thesis, and in this thesis we concentrate ourselves on general features, the most important of which are input and output.

Very often, input and output are designed only in an ad hoc manner, forming the dirtiest part of a programming language. However, we could take a completely different approach: We may adopt as a central issue of language design the clarity of the semantics of input and output and its uniformity with other language constructs.

If we succeed in designing a language along this principle, the effect of improvement can be quite general because every non-trivial program will output something.

In a logic programming language context, an alternative way of input and output was independently proposed in (Nakashima, Ueda and Tomura [1983]) and in IC-Prolog (Clark and McCabe [1980]). These proposals use a notion of streams to make input and output declarative. Stream-oriented input and output has one major advantage of enhancing modularity, reusability, and modifiability of programs. If what we see at the terminal is formalized as a stream of characters, it could easily be directed to the input of another program. If what we input from the terminal is formalized as a stream of characters, the source of input data can easily be replaced by the output of another program. This means that a program with stream-oriented input and output can serve as a module of a larger program without any modification. A different way of input was proposed by Sergot [1983], which is called the 'query-the-user' facility. The difference from the stream-oriented input is that the query-the-user facility utilizes monotonic change of internal database as input.

Chapter 3

PARALLEL LOGIC PROGRAMMING

This chapter introduces parallel logic programming. Parallelism in logic programming can be considered at different levels: Implementation, language/programming, and application. Of these levels, we take a central interest in the language/programming level.

In Section 3.1, we present the motivations for considering parallelism in logic programming. Then we go back to the procedural semantics of logic programs given in Chapter 2 and discuss in Section 3.2 what kinds of parallelism can be exploited. The main sources of parallelism are parallel execution of different derivations and parallelism in a single derivation; they are called OR-parallelism and AND-parallelism, respectively. In Section 3.3, we survey previous research on parallel execution of logic programs. Non-trivial AND-parallelism, or AND-parallelism involving shared variables, requires some control for practical use, and several programming languages and features have been proposed which provides facilities for controlling AND-parallel execution. Although these features spoil the completeness, they increase the descriptive power of Horn-clause logic as a programming language from a practical point of view. Section 3.3 deals with these proposals also, and in Section 3.4 we scrutinize Shapiro's Concurrent Prolog which seems to be the most flexible language of them.

3.1. Parallelism in Logic Programming

Parallelism in logic programming can be considered at various levels: from the level of application domain to the level of hardware implementation. Each level of parallelism has its own significance as stated below.

A. Parallelism in the Application Domains

When we simulate, control, or reason about parallel systems or real-time systems, it is convenient to use a programming language that allows natural and concise description of parallel systems. Temporal logic is ubiquitously used as a framework for describing parallel systems, since we are most interested in state transition of a system along the time axis.

One may wonder why we can and must treat time separately from space once he learns a little bit of the relativity theory. Nevertheless, special treatment of time can be meaningful as long as the system is physically small and slow enough. Several programming languages have been proposed based on various kinds of temporal or modal logic, which includes Templog (Yonezaki, Atarashi and Hourai [1985]), Temporal Prolog (Sakuragawa [1985]), Tokio (Aoyagi, Fujita and Moto-oka [1985]), a language by Aida [1984] and a language by Hagiya [1984b]. However, the motivation of these languages is natural description of time dependency and not all of

these languages are aimed at the description of parallel systems. Note also that not all of these language are aimed at parallel execution. *(end of A.)*

B. Parallelism in Languages and Programming

A logic program is inherently a parallel program in that it expresses no sequentiality. Of course, the *use* of a logic program is *directed* in the sense that an answer substitution is generated with respect to a given goal. However, this does not mean that the computation must proceed sequentially. Prolog introduces sequentiality, but it is primarily for compatibility with sequential computers. Prolog has certainly become expressive by introducing extralogical features (including the sequential execution itself) which are meaningful only under sequential execution, but this never justifies sequential execution because these extralogical features must be reconsidered anyway.

In any programming languages, sequentiality tends to make a program overspecific. Suppose that two variables `X` and `Y` must be initialized to `0` in a Pascal program. There should be no reason to place one of the assignment statements in front of the other; a natural description should not impose a specific order. Inessential sequentiality makes a program and an algorithm less natural and less general.

In procedural languages, some sequentiality is essential because of the destructive assignment to variables. However, logic programming is free from destructive assignments, and therefore the purposes of sequential control are limited to the following:

(1) to gain efficiency, and

(2) to guarantee the availability of data necessary for executing some system predicates such as arithmetics.

Consider a Prolog goal '`Y is X+1`' which unifies `Y` with the *value* of the expression `X+1`. It causes an error if `X` is uninstantiated upon call. However, the above goal need not cause an error if we can postpone it until `X` is instantiated. Thus we can say that the above error condition indicates and results from the discrepancy of data flow and control flow. Control based on data flow should be more natural than sequentiality. It is worth noting that basically sequential as they are, Prolog-II (Colmerauer [1982]) and KL0 (Taki, Yokota, Yamamoto, Nakashima and Mitsuishi [1984]) try to alleviate the inconvenience of sequentiality by their '`freeze`' and '`bind_hook`' primitives, respectively. The mechanism of delayed execution is essential also for realizing inequality of Prolog-II (see Section 2.4.6).

Note that control mechanisms such as delayed execution and coroutines are conceptually a kind of parallel control, though they are sequential-machine oriented. Parallelism at the level of languages and programming is independent of parallelism in implementation, as we will state below. *(end of B.)*

C. Parallelism in Implementation

The motivation for parallelism in implementation is quite simple: Faster execution of programs. Parallelism in implementation is worth exploiting independent of whether the programs are written in a sequential language or a parallel language. This can be well understood by seeing that vector processors exploit parallelism in Fortran programs for numerical computation and performs many mega-FLOPS or even giga-FLOPS.

Conversely, it is also the case that some program in a parallel language is the most suitable for sequential implementation. Suppose that a program creates many processes but just one of them is active at any time. For such a program, a sequential computer will perform as well as a parallel computer. However, it never follows that such a program must have been written in a sequential language; it may really require (pseudo-) parallel control features for concise description, and an optimizing compiler may generate a good object code for it.

The above discussions illustrate that physical parallelism in implementation is independent of logical parallelism at the higher levels. The latter parallelism is often called *concurrency* to distinguish it from physical parallelism. *(end of C.)*

In this thesis, we take a central interest in the level of programming and programming languages, but our interest will extend also to implementation. Although we stated that a logic program is inherently a parallel program, the framework of logic programming as given in Section 2.1 is inadequate for a practical parallel programming language for the following reasons:

(1) The input and output facilities prerequisite for a practical language is hard to keep compatible with uncontrolled parallel execution (see Section 3.3.2).

(2) The framework has not been fully considered in terms of parallel execution. We have not yet clarified what must be considered as primitive operations. Each derivation step in SLD-resolution is too large for a primitive operation, since it involves unification and rewriting of a goal clause.

Therefore, it is not at all a trivial task to design a practical parallel programming language based on Horn-clause logic. In addition, this task is interesting also for the following viewpoint. Much of the previous research on parallelism were devoted to enhancing the performance of sequential programs and to enhancing the expressive power of sequential languages. An example of the former is vector processor implementation of Fortran, which exploits parallelism from sequential programs. However, a piece of a Fortran program allows vectorization only if it has parallelism inherently, that is, if the program piece is overspecific. Therefore, it is meaningful to design a programming language in which parallelism inherent in a problem can be more naturally expressed.

Examples of the latter can be seen in most of the practical parallel languages designed as extension of sequential languages. However, once we have a notation for synchronizing or ordering two operations, it could replace sequential control in principle. One of the reasons why those languages retain sequential control will be for efficiency. However, if a compiler can exploit *sequentiality* from a *parallel* program and generate an efficient code for a sequential computer, we could abolish sequential control. We propose to shift our default control from sequential to parallel, and to consider sequential execution as an *optimization* to meet the current computer architecture.

3.2. Parallel Execution of Logic Programs

In this section, we discuss what kinds of parallelism can be exploited from the procedural semantics of logic programs shown in Section 2.1.3. Conery and Kibler [1981] and Hogger [1984] also discuss parallel execution of logic programs, but without reference to the resolution principle; we find the sources of parallelism in SLD-resolution and argue that parallelism does not lose the soundness and completeness of SLD-resolution.

3.2.1. OR-parallelism and AND-parallelism

Parallelism in a logic program can be divided into two categories. One is parallel search for refutations for a given goal clause. As we stated in Section 2.1.3, finding a refutation of a given goal clause involves searching, since there may be more than one program clause that can be used for each derivation step. It is a natural idea to use parallelism in extending a derivation tree in which to find refutation paths. This parallelism is called OR-parallelism. Note that it is called OR-parallelism though the program clauses used in each derivation step are in conjunction; this is because those clauses are used in refutation.

The above-mentioned OR-parallelism includes a couple of special cases. One is called *database parallelism* or *search parallelism*. Database parallelism means that program clauses usable for some derivation step are searched in parallel. This can be effective when there are many program clauses (which are usually unit clauses) potentially usable for the derivation step. Techniques studied as relational database technology should be applicable in such a case. The other special case is called *pipeline parallelism* or *backup parallelism*. Assume that we have to refute the goal clause "`:-` G_1`,` G_2`, ... .`" and that we sequentially process all the tasks involved in solving each G_i. Moreover, we assume that we always select the leftmost atom in each derivation step. Pipeline parallelism means that the process for G_i sequentially generates all the answer substitutions for the subgoal "`:-` G_1`, ...,` G_i`.`", which are incrementally fed to the process for G_{i+1}. In other words, when the process for G_{i+1} is extending some path in the derivation tree, the process for G_i tries to extend another path. Note that in pipeline parallelism, two goals in the same derivation

path are never solved in parallel though goals in different derivation paths can be. This is why pipeline parallelism is a kind of OR-parallelism.

The other source of parallelism lies in the manner in which each derivation path is constructed. In Section 2.1.3, we sequentially constructed each path in a derivation tree. However, it does not matter how refutations are constructed, as long as those refutations yield correct answer substitutions. Hence it may be possible to select two or more goals simultaneously for derivation and to exploit parallelism in unification. This is usually called AND-parallelism because each refutation is successful when and only when all the parallel tasks involved are successful.

3.2.2. Correctness of Parallel Execution

It is obvious that OR-parallelism does not affect the soundness or the completeness of the procedural semantics since it only exploits independence of different derivation paths. On the other hand, AND-parallelism would require justification, which follows.

Instead of a linear derivation path, we first consider a *proof tree* defined as follows:

(1) The root corresponds to the original goal clause.

(2) Each non-root node corresponds to a program clause used for a derivation step.

(3) Each arc corresponds to a body goal of the clause represented by the parent node.

A proof tree is basically the same as a 'proof term' appearing in (Hagiya [1984a]).

In addition to a program clause C, each non-root node is considered as representing an equation $G = H$ where G is the goal represented by the arc from the parent node and H is the head of the clause C. The goal G is called the *caller* of C, and G is said to *call* C. A proof tree is said to be *closed* if all the leaf nodes correspond to unit clauses. The *answer substitution* of a proof tree is the most general substitution of all the substitutions which simultaneously solve the equations represented by all non-root nodes. A proof tree may not have an answer substitution. A proof tree that has an answer substitution is called a *non-ordered refutation*.

Getting a non-ordered refutation involves two kinds of tasks:

(1) construction of a proof tree and

(2) unification needed to solve equations represented by the non-root nodes.

The AND-parallel proof procedure finds parallelism in these tasks: We may construct a proof tree in parallel, solve equations in parallel, and perform these two tasks in parallel. This proof method is called *parallel input resolution*. The primitive operation in the construction of a proof tree is to put a new node under some

parent node. This corresponds to just providing a body goal with a program clause, leaving unification of the goal and the clause head as a separate operation. The primitive operations in solving a set of equations are the primitive operations in parallel unification. Different models for parallel or nondeterministic unification are proposed by Martelli and Montanari [1982], Yasuura [1983], Dwork, Kanellakis and Mitchell [1984], and Ueda [1986]; we do not choose a specific method but only request that the parallel unification algorithm used in the proof procedure be *correct*, that is, it stops and calculates the most general unifier if and only if the original unification problem has it.

The soundness of parallel input resolution, or the correctness (see Section 2.1.4) of answer substitutions, can be shown by constructing an SLD-refutation from a non-ordered refutation obtained by parallel input resolution.

Given a non-ordered refutation, we first give a total order to its non-root nodes, and hence to the corresponding program clauses and the goals attached to the incoming arcs of those nodes, by topological sorting. Let the obtained sequences of program clauses and goals be $C_1, \ldots, C_n$ and $G_1, \ldots, G_n$. Now we can construct an SLD-refutation (refutation by SLD-resolution) as follows. Assume we have already performed $(k-1)$ derivation steps, obtaining unifiers $\theta_1, \ldots, \theta_{k-1}$ and the kth goal clause. Then let the selected atom be $G_k\theta_1 \ldots \theta_{k-1}$ and employ a new variant of C_k to derive the $(k+1)$th goal clause. It is obvious

(1) that the kth goal clause contains the goal $G_k\theta_1 \ldots \theta_{k-1}$,

(2) that $G_k\theta_1 \ldots \theta_{k-1}$ is unifiable with the clause head of C_k,

(3) that the $(n+1)$th goal clause is empty, and

(4) that $\theta_1 \ldots \theta_n$ is equal to the answer substitution of the non-ordered refutation.

Hence, the construction of an SLD-refutation is guaranteed to succeed. Since SLD-resolution is sound as we seen in Section 2.1.4, we get the soundness of parallel input resolution.

The completeness of parallel input resolution also can be shown by means of the completeness of SLD-resolution. We have seen that for any correct answer substitution θ and under a given selection rule of atoms, a fair proof procedure computes a refutation R with the substitution σ at least as general as θ. Therefore, it suffices to show that a fair proof procedure using parallel input resolution computes a non-ordered refutation yielding a substitution as general as σ.

Given an SLD-refutation R, we can construct a corresponding closed proof tree T straightforwardly. Firstly, a fair proof procedure can generate *any* closed proof tree including T without failure, since the construction of a proof tree is independent of unification. Of course it may stop constructing a tree as an optimization when unification turns out to fail; this, however, does not mean that the construction of a proof tree has failed. Now it suffices to show that the proof procedure finds that T

is a non-ordered refutation with the answer substitution as general as σ. However, a set of equations in T obviously has the most general unifier σ, and we have assumed that the parallel unification algorithm used in parallel input resolution stops and calculates the most general unifier if and only if the original unification problem has it. Therefore, the proof procedure finds that T is a non-ordered refutation with an answer substitution as general as σ.

3.2.3. Restricting and Controlling AND-Parallelism

It may be argued that non-ordered resolution is not realistic because of the large search space. Indeed, the search space includes *all* proof trees having the given goal clause as its root. We can reduce the search space by detecting the failure of unification as early as possible. However, this is often still insufficient.

Consider a unary predicate 'p' with a lot of unit clauses. A goal `p(X)` will generate many answer substitutions if it is called leaving `X` uninstantiated. However, suppose further that there is a goal `q(X)` in conjunction with `p(X)`, which generates only a small number of answer substitutions. Then it would be better to introduce *control* based on the dataflow concept: The goal `p(X)` should wait until `q(X)` finds an answer substitution for `X`.

In general, we must impose some restriction on AND-parallelism in order to minimize computation which does not contribute to final answer substitutions. One way is to disallow AND-parallel execution of two goals G_i and G_j which share variables. If G_i and G_j do not share variables, all answer substitutions of G_i and G_j contribute to the set of answer substitutions of their conjunction. We call this type of AND-parallelism *restricted AND-parallelism* after DeGroot [1984].

The other way is to introduce control as illustrated above. We call it *controlled AND-parallelism*. The purpose of control is to let each goal wait until it is instantiated enough to avoid fruitless computation. Hence the most straightforward way of control will be the control based on dataflow, or to put it differently, the control based on the degree to which each goal is instantiated by the execution of other goals. This is one of the reasons why we introduce a dataflow concept into a logic programming language in Chapter 4.

3.3. Previous Works on Parallel Logic Programming

This section surveys the previous works on parallel execution of logic programs and the proposals of parallel logic programming languages.

3.3.1. OR-Parallelism

Research on OR-parallelism has been done mainly from architectural aspects. The most important issue characteristic of OR-parallel execution of logic programs

is the management of multiple environments, i.e., multiple substitutions generated by different paths of a derivation tree. Other issues include resource management such as scheduling and spatial allocation of processes, but they seem to be common to all parallel processing systems.

We will explain in terms of SLD-resolution what is difficult with multiple environments. Let the goal clause to be rewritten be

$$:- \ G_1 , \ \ldots , G_k , \ \ldots , G_n . \qquad (3.3.1\text{–}1)$$

where G_k is the selected atom. Let the following be new variants of two different program clauses

$$H_i \ :- \ B_{i1} , \ \ldots , B_{im_i} . \qquad \text{or} \qquad H_i . \qquad (3.3.1\text{–}2)$$

$$H_j \ :- \ B_{j1} , \ \ldots , B_{jm_j} . \qquad \text{or} \qquad H_j . \qquad (3.3.1\text{–}3)$$

such that H_i and H_j are unifiable with G_k by the most general unifiers σ_i and σ_j, respectively. Then, the goal clause is rewritten in two ways:

$$:- \ G_1\sigma_i , \ \ldots , G_{k-1}\sigma_i , B_{i1}\sigma_i , \ \ldots , B_{im_i}\sigma_i , G_{k+1}\sigma_i , \ \ldots , G_n\sigma_i . \qquad (3.3.1\text{–}4)$$

$$:- \ G_1\sigma_j , \ \ldots , G_{k-1}\sigma_j , B_{j1}\sigma_j , \ \ldots , B_{jm_j}\sigma_j , G_{k+1}\sigma_j , \ \ldots , G_n\sigma_j . \qquad (3.3.1\text{–}5)$$

A depth-first sequential proof procedure usually takes advantage of the fact that (3.3.1–4) and (3.3.1–5) never co-exists in making both derivations. That is, it records what variables are rewritten by σ_i when it derives (3.3.1–4). When (3.3.1–4) turns out to fail and (3.3.1–5) is derived, the proof procedure restores (3.3.1–1) from (3.3.1–4) by *undoing* all bindings by σ_i using the recorded data. These recording and undoing operations are usually much cheaper than to record (3.3.1–1) itself.

However, this technique cannot be used in OR-parallel execution, since OR-parallelism means that (3.3.1–4) and (3.3.1–5) may co-exist. One way of having (3.3.1–4) and (3.3.1–5) simultaneously is to obtain those clauses not by rewriting (3.3.1–1) but by copying. This method is a kind of *shallow binding*; it requires fairly expensive copying operations, but guarantees efficient access to the new goal clause. Another way is to manage σ_i and σ_j together with the common original clause. This method is a kind of *deep binding*; creating new goal clauses is cheap, but access to a new goal clause requires applying a substitution to the ancestor clause (3.3.1–1) which itself may have to be obtained in the same manner.

Most of the proposals of OR-parallel Prolog machines so far use the copying scheme. In PIE (Goto, Tanaka and Moto-oka [1984]; Moto-oka, Tanaka, Aida, Hirata and Maruyama [1984]), the reduction subprocess in an OR-parallel unify process performs copying; that is, it makes a new goal clause from an old one and a binding information. In PIM-D (Ito and Masuda [1984]; Ito, Shimizu, Kishi, Kuno and Rokusawa [1985]), copying is performed by its 'substitute' operator. PIM-R (Onai, Aso, Shimizu, Masuda and Matsumoto [1985]) also adopts the copying

scheme. On the other hand, K-Prolog (Matsuda, Tamura, Kohata, Kaneda and Maekawa [1985]) employs the deep binding scheme.

Ciepielewski and Haridi [1983] and Ciepielewski [1984] are well aware of the implementation issue of multiple environments and propose a different scheme. Their approach could be called *demand copying*. They use the structure sharing scheme (Boyer and Moore [1972]), where the environment consists of

(1) contexts which store binding information and

(2) one environment dictionary associated with each node of a derivation tree whose entries point to context items.

The size of each environment dictionary depends on the level of the node of the derivation tree with which the dictionary is associated. For each derivation step, a new environment dictionary is created and context items pointed to by the dictionary of the parent node and containing uninstantiated variables are copied unless we introduce any optimization. They introduced lazy construction of the dictionary and lazy copying of the context items it points to. Then the initialization of a new environment becomes cheap as in the deep binding scheme. The first access to the value of some variable requires looking up the current and the parent environment dictionaries and possibly updating the dictionaries and the contexts, which can be as expensive as the deep binding scheme. However, the subsequent accesses to the same variable can be done in a constant time like the shallow binding scheme. Avoiding eager copying will be effective if only a small portions of copies are referenced.

The author (Ueda [1985c]) proposes a quite different approach to the OR-parallel execution of logic programs. He proposes to compile OR-parallelism into (controlled) AND-parallelism. By using mode analysis, his method 'compiles away' difficulties in the management of multiple environments with uninstantiated variables. Detailed description of the method can be found in Chapter 7.

Warren [1984] proposes a memory management scheme for non-depth-first but sequential execution of logic programs. His method also mixes deep binding and shallow binding: A tree of activation records serves as association lists and a binding array holds bindings corresponding to the current node in a derivation tree. Entries of the binding array are filled 'on demand' from the data of the activation records. When this method is applied to a multiprocessor environment, a binding array need not be prepared for each *process* executing some path of a derivation tree but for each *processor* which executes a number of processes by repeated context switching.

Architectures for database parallelism have been studied in (Nakagawa [1984]), (Warren, Ahamad, Debray and Kalé [1984]) and (Taylor, Lowry, Maguire and Stolfo [1984]). Pipeline parallelism has been studied in (Nitta, Matsumoto and Furukawa [1983]), (Kanada [1985]) and (Matsuda, Tamura, Kohata, Kaneda and Maekawa [1985]). Kanada showed how to realize OR-parallelism on a supercomputer with

pipeline processing capabilities. We must note also that Tick and Warren [1984] study pipelining at the very low level; they propose a processor which efficiently executes a sequence of compiled codes by internal pipelining. It will be meaningless to classify such low-level parallelism into AND-parallelism or OR-parallelism.

3.3.2. AND-Parallelism

As we have seen in Section 3.2.3, AND-parallelism requires restriction and/or control for practical use. Therefore, research on AND-parallelism has been done mainly on the mechanisms for the restriction and control.

First, we briefly review restricted AND-parallelism. Conery and Kibler [1985] and Conery [1983] propose a method for serializing any two goals with shared variables. Serialization is realized by computing the partial order among the set of current goals; two goals can be executed in parallel only if there is no ordering between them. The problem is that the partial order must be computed many times at runtime. DeGroot [1984] proposes a coarser but more efficient algorithm which checks absence of shared variables between two goals. PIM-D executes two goals in parallel only when it is told to do so.

The rest of this section deals with controlled AND-parallelism. Research on controlled AND-parallelism goes back to the research on coroutining. Although it may look a little bit complex, coroutining is a natural control scheme which enables scheduling based on the availability of data. Coroutining makes naive generate-and-test programs practical to some extent by the early detection of failure.

Two methods for coroutining have been proposed: One is coroutining based on the number of reduction steps, and the other is dataflow coroutining. The former is used in Epilog (Porto [1982]) and appears also in (Pereira [1982]). They achieve coroutining by means of the delay primitive which causes one-cycle delay to the reduction of the specified goal. The problem of this method is that the primitive is rather machine-oriented: The amount of delay must be correctly specified according to the given coroutining interpreter, and the semantics of the primitive can be understood only in terms of that interpreter. In addition, this method is not easily amendable to parallel evaluation because the semantics is defined in terms of the centralized interpreter.

Dataflow coroutining is a more user-oriented way of specifying coroutining. IC-Prolog (Clark and McCabe [1980]), among a variety of its features, allows a programmer to specify what occurrence of a shared variable should act as a generator, and it schedules goals based on this information. The specification of the generator (or the consumers) is done by annotations on variables. This scheme is appropriate also for (pseudo-)parallel processing, and IC-Prolog actually allows both parallelism and coroutining. Here, parallelism means that the generation of bindings by the producer goal can go ahead of the consumption of them, while coroutining means that the producer cannot generate a new binding until the consumer processes the

previous one. However, the control primitives are only very informally described, and so the semantics of the language is not so simple and clear as its successors such as the Relational Language described below.

Coroutining and parallelism controlled by annotations provide an ideal framework for programming with input and output (Nakashima, Ueda and Tomura [1983]). As we stated in Section 2.3.3, a program with input and output should keep the *relational* programming style, that is, it should be written as the relation of a sequence of input data and a sequence of output data:

```
:- in(In), transform(In, Out), out(Out).
```

While the above goals could be processed sequentially, parallelism or coroutining produces the following advantages. Firstly, it realizes pipelined processing of the input sequence `In` and enables incremental generation of the output data `Out`. Secondly, it saves memory space because the part of the input and output sequences already processed can be discarded as garbage. Moreover, the relational style is advantageous in modularity and reusability of programs, and it encourages *differential programming*. The output of a program can easily be fed as the input of another program if we handle input and output data as sequences.

Sequences of input/output data, which are usually implemented as linear lists, are often called *streams*. More generally, a sequence of data transferred from one goal to another by means of a shared variable is called a stream. Controlled AND-parallelism is often called *stream AND-parallelism*, since a stream is an important pragmatic concept in controlled AND-parallelism and large-grain parallelism can be exploited by parallel execution of goals communicating by streams. Goals which consume and generate streams are often called *processes*.

Van Emden and de Lucena [1982] presented a *process interpretation* of logic programs (see Section 4.8) by using simple examples. They showed a network of parallel processes can be described in logic programming. However, although their exposition well describes the basic idea, it does not deal with many subtle points in the semantics of parallel execution. Like other early proposals of stream AND-parallelism, parallelism was introduced as an additional construct to the sequential framework, and each process is regarded as a stack of computation. They did not use annotations for synchronization; the direction of dataflow is dynamically determined by what arguments have been instantiated upon call.

The above researches paid little attention to the determinacy of communicated data, i.e., whether the bindings transmitted are the only possible ones or they have alternatives. The determinacy of bindings, however, is very important from a practical point of view. If many processes distributed on multiple processors are to send and receive nondeterminate bindings, we must implement so-called *distributed backtracking*, a mechanism for withdrawing already broadcasted bindings. Even more importantly, data to be output to an external device must be determinate,

because it is almost impossible to erase a sequence of characters once they are typed out on paper.

One remedy to this problem is allow only *determinate* data to be output or transmitted between processes. This restriction greatly simplifies implementation. Relational Language (Clark and Gregory [1981]) introduced the guard mechanism of Dijkstra [1975] for this purpose, as well as its successors PARLOG (Clark and Gregory [1983]; Clark and Gregory [1984a]), Concurrent Prolog (Shapiro [1983a]) and Guarded Horn Clauses (Ueda [1985b]). Roughly speaking, all these languages have the following features in common:

(1) Goals in each program clause are divided into guard goals and body goals.

(2) For any goal in a goal clause, only one of the program clauses that have solved head unification (unification of the goal and the clause head) and guard goals is allowed to solve its body goals. This mechanism is called *committed-choice nondeterminism*, and the selection of one of the program clauses is called *commitment*.

(3) When head unification and guard goals of some clause are executed to refute a goal clause, they cannot instantiate the goal clause before commitment.

(4) As a consequence of (2) and (3), all bindings are determinate.

The above languages are different in the way in which (3) is guaranteed. Relational Language asks a programmer to put an annotation on the generator occurrence of a shared variable which they call a channel variable. Then it imposes three restrictions. Firstly, head unification must not bind a non-generator occurrence of a variable in the goal; if this is violated, the clause becomes an *input suspended clause*. Secondly, when head unification have succeeded, all the guard goals must be instantiated to true ground atoms. Clearly the second restriction guarantees that the guard goals generate no bindings. The third restriction is that output bindings generated for the generator occurrence of a variable are applied to the whole goal clause after commitment. Although not emphasized in the original paper, the last restriction should not be overlooked; without it, output bindings from different program clauses may collide.

PARLOG uses mode declaration instead of annotation. Each predicate must have its own mode declared. Mode declaration declares each argument of a predicate as input or output. Then the following restrictions are imposed. Head unification must not generate bindings to the input arguments of the goal. If the head unification can succeed only by violating this restriction, it suspends. Another restriction is that guard goals cannot generate bindings to those input arguments either. A guard (i.e., a set of guard goals) which satisfies this restriction is called a *safe* guard. Although safety of a guard could be checked at run time, it is intended to be checked at compile time. The third restriction is that bindings to the output arguments of a goal are generated only after commitment.

In Concurrent Prolog, the mechanism for synchronization and that for guaranteeing determinacy of bindings are provided independently. It employs *read-only annotation* for synchronization. The annotated occurrence of a variable cannot be instantiated to a non-variable term by the unification in which the occurrence is involved. For determinacy, it uses the local (and multiple) environment mechanism. Until commitment, bindings generated by head unification or guard goals which would instantiate the goal clause are recorded locally, which form a local environment for each program clause. Since there may exist two or more local environments at the same time which might bind the same variable to different terms, we must implement a multiple environment mechanism. The locally recorded bindings can be exported (i.e., applied) to the goal clause after commitment, but only one program clause is allowed to do so. Thus all bindings given to the goal arguments are determinate.

Guarded Horn Clauses guarantees the determinacy of bindings by making head unification and guard goals suspend if they would otherwise instantiate the goal clause. Chapter 4 contains a detailed explanation.

Let us compare these languages from other viewpoints. A communication channel of Relational Language allowed only one-way communication; if we want to receive answers of messages, we had to prepare another stream for them. Concurrent Prolog enabled a more flexible way of communication by means of the back communication technique. The technique uses an incomplete message, a message with an uninstantiated variable. When the generator of a stream instantiates its head to an incomplete message, the consumer binds its uninstantiated variable to an answer. Concurrent Prolog appears to be the most simple and flexible parallel logic programming language of the ancestors of Guarded Horn Clauses at least syntactically, so we will examine it in detail in Section 3.4.

While Concurrent Prolog realized back communication in a language with multiple environments, PARLOG realized the same feature in a single-environment language. In addition, PARLOG showed a compilation technique for head unification together with an algorithm for checking safety of a guard which is applicable for a large class of programs.

Comparison of Guarded Horn Clauses with its ancestors will be made in Chapter 5, in which we will go into more technical detail. Finally, we note that the surveys of stream-AND-parallel logic programming languages can be found in (Nitta [1984]) and (Gregory [1985b]) also.

3.4. Examination of Concurrent Prolog

In this section we scrutinize the language rules of Concurrent Prolog in Shapiro's original paper. The main result is that some sequentiality must be assumed for unification in order to reasonably define the semantics of unification and commitment. This means that some 'logical' transformation of a program clause may change its semantics. Another point is that there are semantical problems in multiple environments and a commitment operation. The results of this section are based on (Ueda [1985a]) which is a substantially extended version of (Ueda [1984]). Problems with the semantics of Concurrent Prolog are examined also by Saraswat [1985] independently and from a different viewpoint.

3.4.1. Motivation and Method

In the previous sections, we have examined parallelism in logic programming in general. Here we will go into more detail and examine one of the best-known parallel logic programming languages, Concurrent Prolog (Shapiro [1983a]). Thorough examination of a specific language will help identifying subtle points in parallel logic programming and will possibly justify designing an alternative language.

Concurrent Prolog was chosen because its language rules were very concise and it looked expressive enough. We judged that it should be appropriate as the first approximation toward the ultimate parallel logic programming language. Our programming experience at Institute for New Generation Computer Technology (ICOT) then showed that Concurrent Prolog was fairly expressive. However, we have not examined whether it is really concise; it is still described very informally except for the operational semantics by Hirata [1984]. A simple language rule expressed in a natural language may be formalized into a set of quite complex rules. Therefore, we must examine every subtle point of Concurrent Prolog and make necessary clarifications or modifications.

We must consider the following points in the examination. Firstly, it must not contain ambiguous or inconsistent rules. Secondly, it should be considered in the light of both parallelism at the language level and parallel implementation. Thirdly, since Concurrent Prolog is a programming language originated in Horn-clause logic, we must consider to what extent the properties of logic programming in the original sense are retained and how essential the deviation from Horn-clause logic is.

In the following, the language Concurrent Prolog will be re-examined in the light of the above criteria. We will regard (Shapiro [1983a]) as the defining document of Concurrent Prolog, since it is the original and the most detailed text. The fundamental method of examining a language defined informally is to examine every defining sentence thoroughly. This is quite useful for detecting problems in a language definition; for example, the author applied the same technique to examine the language rules of the preliminary version of Ada (Ueda [1982]).

However, here we will not make comments sentence by sentence: Our purpose is to try to obtain a correct semantics of Concurrent Prolog and not to criticize the defining sentences. For this purpose, we examined other documents on Concurrent Prolog by Shapiro [1984] and by Shapiro and Takeuchi [1983] also, but no significant difference from the original document or new information which might help us to reach the correct interpretation was found as for the materials discussed in this section.

3.4.2. The Definition of Concurrent Prolog

This section introduces the syntax and the semantics of Concurrent Prolog as described in the original paper (Shapiro [1983a]). We will quote important paragraphs from that paper and number them for later references. Note that although the language defined in (Shapiro [1983a]) was first called "a subset of Concurrent Prolog", it has simply been called Concurrent Prolog among the community ever since.

3.4.2.1. Syntax

[1] (Section 3.1) A Concurrent Prolog program is a finite set of guarded clauses. A guarded clause is a universally quantified axiom of the form

$$A \text{ :- } G_1, G_2, \ldots, G_m \mid B_1, B_2, \ldots, B_n. \qquad m, n \geq 0$$

where the G's and the B's are atomic goals. The G's are called the *guard* of the clause and B's are called its *body*. When the guard is empty the commit operator is omitted. The clause may contain variable marked "read-only".

[2] (Section 3.1) The commit operator generalizes and cleans sequential Prolog's cut. Declaratively, it reads like a conjunction: A is implied by the G's *and* the B's. ...

3.4.2.2. Semantics

[2] (Section 3.1) ... Operationally, a guarded clause functions similarly to an alternative in a guarded-command. It can be used to reduce process $A1$ to a system B if A is unifiable with $A1$ and, following the unification, the system G is invoked and terminates successfully.

[3] (Section 3.1) ... The unification of a read-only term $X?$ with a term Y is defined as follows. If Y is non-variable then the unification succeeds only if X is non-variable, and X and Y are recursively unifiable. If Y is a variable then the unification of $X?$ and Y succeeds, and the result is a read-only variable. The symmetric algorithm applies to X and $Y?$.

. . .

[4] (Section 3.1) This definition of unification implies that being "read-only" is not an inherited property, i.e. variables that occur in a read-only term are not necessarily read-only. Stating it differently, the scope of a read-only annotation is only the principal functor of a term, but not its arguments. . . .

[5] (Section 3.1) The definition of unification also implies that the success of a unification may be time-dependent: a unification that fails now, due to violation of a read-only constraint, may succeed later, after the principal functor of a shared read-only variable is determined by another process, in which this variable does not occur as read-only.

[6] (Section 3.2) The execution of a Concurrent Prolog system $\boldsymbol{S}$, running a program $\boldsymbol{P}$, can be described informally as follows. Each process A in $\boldsymbol{S}$ tries asynchronously to reduce itself to other processes, using the clauses in $\boldsymbol{P}$. A process A can reduce itself by finding a clause $A1$:- G | B whose head $A1$ unifies with A and whose guard system G terminates following that unification. The system $\boldsymbol{S}$ terminates when it is empty. It may become empty only if some of the clauses in $\boldsymbol{P}$ have empty bodies.

[7] (Section 3.2) The computation of a Concurrent Prolog program gives rise to a hierarchy of systems. Each process may invoke several guard systems, in an attempt to find a reducing clause, and the computation of these guard systems in turn may invoke other systems. The communication between these systems is governed by the commitment mechanism. Subsystems spawned by a process A have access only to variables that occur in A. As long as a process A does not commit to a reducing clause, these subsystems can access only read-only variables in A, and all binding they compute to variables in A which are not read-only are recorded on privately stored copies of these variables, which is not accessible outside of that subsystem. Upon commitment to a clause $A1$:- G | B, the private copies of variables associated with this clause are unified against their public counterparts, and if the unification succeeds the body system B of the chosen clause replaces A.

[8] (Section 3.2) A more detailed description of a distributed Concurrent Prolog interpreter uses three kinds of processes: an and-dispatcher, an or-dispatcher, and a unifier; these processes should not be confused with the Concurrent Prolog processes themselves, which are unit goals.

[9] (Section 3.2) The computation begins with a system $\boldsymbol{S}$ of Concurrent Prolog processes, and progresses via indeterminate process reduction. After an and-dispatcher is invoked with $\boldsymbol{S}$, the computation proceeds as follows:

- An *and-dispatcher*, invoked with a system $\boldsymbol{S}$, spawns an or-dispatcher for every Concurrent Prolog processes A in $\boldsymbol{S}$, and waits for all its child or-dispatchers to report success. When they do, it reports success and terminates.

- An *or-dispatcher*, invoked with a Concurrent Prolog process A, operates as follows. For every clause $A1 \ :- \ G \mid B$, whose head is potentially unifiable with A, it invokes a unifier with A and the clause $A1 \ :- \ G \mid B$. Following that the or-dispatcher waits for any of the unifiers to report success. When one such report arrives, the or-dispatcher reports success to its parent and-dispatcher and terminates.

- A *unifier*, invoked with a Concurrent Prolog process A and a guarded clause $A1 \ :- \ G \mid B$, operates as follows. It attempts to unify A with $A1$, storing bindings made to non read-only variables in A on private storage. If and when successful, it invokes an and-dispatcher with G, and waits for it to report success. When this report arrives, the unifier attempts to commit, as explained below. If the commitment completed successfully it reports success, but in either case it terminates.

[10] (Section 3.2) At most one unifier spawned by an or-dispatcher may commit.

...

[11] (Section 3.2) To commit, a unifier first has to gain a permission to do so. The mutual exclusion algorithm must guarantee that if at least one unifier wants to commit, then exactly one unifier will be given permission to do so. After gaining such a permission, the unifier attempts to unify the local copies of its variables against their corresponding global copies. If successful, then the commitment completes successfully.

[12] (Section 3.2) ... Another useful optimization is the deletion of brother unifiers, once the first such process is ready to commit.

[13] (Section 3.2) When committing, the unifier is not required to perform the unification of the public and private copies of variable as an "atomic action". The only requirement is that the unification be "correct", in the sense that it should not modify already instantiated variables, which can be achieved in a shared memory model with a test-and-set primitive.

...

[14] (Section 3.2) Since a unification that currently fails may succeed later, the phrase "attempts to unify" in the description of a unifier should be interpreted as a continuous activity, which terminates only upon success. This can be implemented using a busy-waiting strategy, but several optimizations can be incorporated. ...

We correct a small error in Paragraph [9] here. An or-dispatcher should report a set of processes B to its parent and-dispatcher instead of the simple message "success". The and-dispatcher must recognize these processes as newly created processes which replace the original one, and must spawn or-dispatchers for them. The and-dispatcher reports success and terminates when $\boldsymbol{S}$ is reduced to an empty

set of goals. Alternatively, an or-dispatcher may invoke a new and-dispatcher for the body of a clause whose corresponding unifier has reported success, and let the report of this and-dispatcher be the report of the or-dispatcher in question.

3.4.2.3. Parallel Programming in Concurrent Prolog

[15] (Section 3.1) A system of processes corresponds to a conjunctive goal, and a unit goal to a process. The state of a system is the union of the states of its processes, where the state of a process is the value of its arguments. *And*-parallelism—solving several goals simultaneously—provides the system with concurrency. *Or*-parallelism—attempting to solve a goal in several ways simultaneously—provides each process with the ability to perform indeterminate actions. Variables shared between goals serve as the process communication mechanism; and the synchronization of processes in a system is done by denoting which processes can "write" on a shared variable, i.e. unify it with a non-variable term, and which processes can only "read" the content of a shared variable X, i.e. can unify X with a non-variable term T only after X's principal functor is determined, possibly by another process. ...

3.4.3. Multiple Environments and a Commitment Operation

A commitment operation as described in Paragraphs [11] and [13] is a process rather than an event. It is preceded by the permission, and it completes when the unification of local and global copies of variables has terminated successfully. It is not explicitly specified when the commitment starts, but it should be some time after the permission and not later than the start of the unification.

The most controversial issue is the nature of the permission. The second sentence of Paragraph [11] says that the mutual exclusion algorithm must guarantee that if at least one unifier wants to commit, then exactly one unifier will be given permission to do so. However, it is not clear whether

(1) this permission is revoked when the unification of local and global copies of variables does not succeed, thus providing the other clauses with the opportunity of commitment, or

(2) this permission is eternal, i.e., the other clauses can no longer attempt a commitment operation once some clause has gained a permission.

The failure of the unification can happen when a global variable is further instantiated by some other goals after its local copy is created, but Paragraphs [7], [9], and [11] define only the successful case. If the permission is never revoked, failure of the unification means the failure of its grandparent and-dispatcher, i.e., the parent of its parent or-dispatcher. If the permission can be revoked, the unification must be performed in a way in which the intermediate result of the unification is

invisible from other processes. This is because this unification may eventually fail, in which case the other clauses must retain the possibility of commitment.

Paragraphs [5] and [14] say that unification is a continual activity which terminates only upon success. So one interpretation could be that the commitment operation is also a continual activity which terminates only upon success of the unification involved in it. This suggests that the permission need not be revoked. However, this interpretation is unfortunately inconsistent with the description in Paragraph [9] that the unifier terminates whether or not the commitment completed successfully, though Paragraph [9] is problematic in that it does not say at all how the unifier can terminate when the commitment has not completed successfully.

The complete lack of the description of a locking operation, which is shown below to be necessary if the permission can be revoked, suggests that the permission is of an eternal nature. However, the interpreter shown in (Shapiro [1983a]) adopts the opposite interpretation. It seems impossible to derive a correct answer from the original description; the better way should be to make a thorough examination of the merits and demerits of the both alternatives.

3.4.3.1. The First Alternative: Permission Is Revocable

We first assume that the permission of commitment is revoked when the unification of local and global copies of variables does not succeed. When the permission is revoked, the intermediate result of unification must be kept invisible from other goals. Since it is only upon success of the unification that the revocation of the permission is known to be unnecessary, we must always perform the unification of local and global information in a way in which its partial result is invisible from other goals. The problem is how to implement a commitment operation which meets the above requirement.

The only possible solution seems to be to perform the following operations in the given order:

(a) Lock (at least) all the relevant global data, that is, all variables appearing in goal arguments for which local copies have been made.

(b) Try to export local bindings by unifying local copies with its global counterpart. This could start before Step (a) is finished, as long as no bindings are made to unlocked global variables.

(c) If the unification is successful, do (c1); else do (c2):

  (c1) If the unification is successful, then simply unlock the global data locked in Step (a).

  (c2) If the unification is unsuccessful, then undo all the bindings made in Step (b), and then unlock the global data locked in Step (a). The unlocking can start before undoing is finished, as long as no global variable is unlocked

without being unbound. Independently of these operations, return the permission of commitment.

The problem lies in the locking operation in Step (a). The simplest locking scheme would be to lock the whole memory whenever commitment is attempted, but this is definitely unacceptable because it serializes all commitment operations. If we do not want to lose parallelism, we must minimize the locked area.

The smallest unit of locking is a single variable. However, variable-by-variable locking is not easy when we have to lock two or more variables in one commitment operation, as studied in the area of distributed databases and operating systems. Assume there are two clauses (say C_1 and C_2) attempting to be selected and that both of them must lock the variables X and Y. If C_1 tries to lock X first and C_2 tries to lock Y first, they may deadlock.

This deadlock problem can be easily resolved if we can order the variables. If we can give an invariant order to a set of variables to be locked, each clause has only to lock them in that order. However, it is hard to consider such an invariant order, because two variables may be unified at any time and after that the unified two variables must have the same order. All the above considerations lead us to the conclusion that the first alternative is unacceptable.

One may think we could detect ununifiability of local and global information earlier than commitment. This is enabled by checking unifiability of the local and the global values of a variable whenever new global or local binding for that variable is created. However, this 'eager' checking never eliminates the unification upon commitment, and this unification must still satisfy all the requirements we stated above.

3.4.3.2. The Second Alternative: Permission Is Eternal

Let us then consider the other alternative that the permission of commitment is of an eternal nature. We no longer need locking operations because it is now impossible for the other clauses to export bindings later. Unification may be done just in a usual manner. This alternative increases the chance of failure of a program in which two or more conjunctive goals try to instantiate the same variable upon commitment. However, it does not cause so much inconvenience. Most predicates we have written are effectively (possibly nondeterministic) functions each of which returns only one result. Such a result may be prepared in a guard and exported upon commitment, but we usually receive it by a variable, in which case no failure can happen.

However, there still remain some semantical problems. Although it is unnecessary to return a permission of commitment once it is obtained, we have to define when a 'unifier' (in terms of Paragraph [9]) can attempt to gain such a permission. In other words, we have to define what kind of global information supplied from goal

arguments must be respected when we regard head unification and the execution of a guard as successful. Some information which may come later than the attempt of commitment could be ignored, but other information which is evidently available must be considered for clause selection. Consider the following goal:

```
:- p(a).                                                    (3.4.3–1)
```

This goal says two things: call the unary predicate '`p`' and set its argument to '`a`'. The question is whether or not the argument setting must be completed before the call. This is not an absurd question; we are examining Concurrent Prolog as a truly parallel language. If the argument setting *must* precede the call, the above goal never fails when the predicate '`p`' is defined as follows:

```
p(a).                                                       (3.4.3–2)
p(b).                                                       (3.4.3–3)
```

Only Clause (3.4.3–2) succeeds in head unification.

However, if the argument value may be determined arbitrarily late, the above goal may fail. Clause (3.4.3–3) may be tested earlier, and the value '`b`' in the head may be recorded locally for later unification if the corresponding goal argument has not been determined. In this case ununifiability of '`a`' and '`b`' will be detected after Clause (3.4.3–3) has gained a permission of commitment, and the original goal finally fails. To generalize, any program clause can be selected for a given goal as long as its guard succeeds, and can make the whole system fail. This should be extremely inconvenient, because we cannot write a predicate like '`p`' intended to check the values of its arguments.

Therefore, at least any information specified textually in a goal must be considered for clause selection, i.e., any inconsistency with local information must be detected before attempting a commitment operation. This means that Clause (3.4.3–3) in the above example must detect inconsistency between '`a`' and '`b`' at head unification and must never create a local copy of its argument as long as the permission of commitment is not revocable.

Note that the above conclusion applies also to our first alternative on the semantics of commitment that the permission of commitment is revocable. For if we allow delay of the determination of an immediate argument value, it may be determined too late—after commitment has completed.

The question of allowed delay of information is related to the semantics of the unification of Concurrent Prolog, and it will be discussed further in Section 3.4.4.

3.4.3.3. Access to Local/Global Information

Another problem which arises by allowing local and global copies of a variable is to which copies we must have access in each of the following cases:

(1) A variable in a goal is textually marked read-only.

(2) A variable in a goal is not textually marked read-only and its local copy has been made.

(3) A variable in a goal is not textually marked read-only and its local copy has not been made.

The first possibility for a goal variable is that it is marked read-only textually. Then candidate clauses must watch its global value. This is necessary for making suspended unification succeed and putting computation forward. For such a variable, local copies are not created (with the exceptional case shown below), since no bindings can be given directly to read-only variables in a goal from a clause head or a guard.

However, there also exists a rather pathological case where a local copy must be created for a read-only variable. Consider the following example:

Goal: `:- p(X?, X), X = a.` (3.4.3–4)
Program: `p(a, A) :- A = a | true.` (3.4.3–5)

The predicate '`=`' unifies its two arguments. The first argument may be instantiated in two ways:

(a) global instantiation by the goal '`X=a`' running in parallel, and

(b) local instantiation by the goal '`A=a`'.

In the second case, a local copy of `X`, which appears with (and without) read-only annotation, must be created. Note that the goal '`A=a`' must locally instantiate the first argument: The goal textually specifies that its two arguments be identical (except for the annotation), so this identity must be respected for clause selection as we concluded in Section 3.4.3.2. This example illustrates also that suspension of unification due to a goal variable marked read-only may be resolved in two ways, globally and locally. Hence it is generally inadequate to wait only for global instantiation of a read-only variable; we have to implement multiple waits.

The second possibility for a variable contained in a goal is that it is not marked read-only textually and that some clause has created its local copy. In this case, that clause should see the local copy. Ignoring it and seeing only its global counterpart might suspend some unification which would otherwise succeed. However, it is not clear whether the clause should be allowed to see also the global value which may get instantiated after the local copy is created. Paragraph [7] seems to disallow it, but it is possible that the clause can solve its guard only by using the global value of some variable for which a local copy has been made.

The third possibility is that a variable contained in a goal is not textually marked read-only and that its local copy has not been created either. This is further divided into two subcases:

(a) The variable is uninstantiated when head unification starts.

(b) The variable has become non-variable or read-only when head unification starts.

For each case, there are three possible interpretations:

(1) The clause should watch its global instantiation.

(2) The clause can ignore it.

(3) The clause *should* ignore it.

Paragraph [7] seems to support Alternative (3) for Case (a) and Alternative (1) for Case (b).

Let us consider Case (a) first. Alternative (3) implies that the value of a variable in a goal which has become read-only or become non-variable *after* the goal is called should be ignored. For instance, the following program

```
      :- p(X), X = a.                    (3.4.3–6)
p(X) :- X? = a | true.                   (3.4.3–7)
```

should never succeed as long as `p(X)` is called before '`X=a`' is executed.

However, we cannot adopt the same alternative for Case (b). In this case, the value of the variable in the goal should not be ignored. Otherwise, the 'protected data' technique (Hellerstein and Shapiro [1984]) (Takeuchi and Furukawa [1985]) would not work correctly.

Typical use of read-only annotation is to attach it to the argument variables of goals which consume (or decompose) the value of those variables. However, the protection against instantiation by the consumer can be achieved also by making the generator of a data structure protect its uninstantiated part, and 'protected data' mean such uninstantiated but protected variables created by the generator. When we use this technique, read-only annotations do not appear textually in the consumer goal (except for the top level) but it is sent from the generator goal. If such dynamic protection should be ignored as Alternative (3) says, this technique could not be used.

Cases (a) and (b) can occur depending on the relative timing of head unification and the instantiation of a variable in the goal (by some other goal). Moreover, head unification is not an instantaneous operation, so it is undesirable to assign the mutually exclusive behaviors (1) and (3) to these two cases. Alternative (1) or (2) should be a better choice for Case (a).

Our claim that Alternative (3) is undesirable could be understood and justified also from the following observation. If Clause (3.4.3–6) were given as

```
:- p(a).                                 (3.4.3–8)
```

it should succeed because the textually specified argument value must not be ignored. This means that partial evaluation of '`X=a`' in Clause (3.4.3–6) to get Clause (3.4.3–7) undesirably changes the semantics of the program from suspension to success.

3.4.4. Atomic Operations in Unification

The semantics of unification must clearly state what are atomic operations. Consider the following unification:

```
:- ..., X = f(a), ...                                    (3.4.4–1)
```

How can this unification be performed, assuming that `X` are uninstantiated? Possible solutions may be:

(1) Create a term `f(a)` (in any manner). Then set `X` to this term.

(2) Create a term `f(A?)` where `A` is a fresh variable. Then set `X` to this term, and in parallel with this set `A` to '`a`'.

(3) Create a most general unary term with the principal function symbol '`f`'. Call its argument `A` (this is not part of the operation). Then set `X` to this term, and in parallel with this set `A` to '`a`'.

Alternative (1) regards the unification of a variable and a non-variable term as an indivisible operation. Alternative (2) tries to allow the principal function symbol and its arguments to be determined in parallel, but it protects non-variable arguments by read-only annotation. Alternative (3) states that the above unification can be done as if it were specified as follows:

```
:- ..., X = f(A), A = a, ...                             (3.4.4–2)
```

where `A` is a variable not appearing elsewhere.

Let us explain the differences among these alternatives in other words. Alternative (1) says that when the principal function symbol of the value of some variable has been determined, the values of its arguments have also been determined as long as they are specified texually at the same place as the principal function symbol. Alternative (2) says that the argument values may come later, but that the uninstantiated fresh variables appearing in the transient state are protected. Alternative (3) says that the arguments may come later and that the variables are not protected.

Note that for unification between two non-variable terms also, we can conceive three alternatives corresponding to the above ones.

The granularity of atomic operations is smallest in Alternative (3) and largest in Alternative (1). So, Alternative (3) seems to be the best under the 'high-parallelism' criterion. We will see in Section 3.4.4.1, however, that Concurrent Prolog cannot adopt Alternative (3).

One remark on parallelism in unification must be made here. It is known that the unification problem has a sequential nature in general, that is, parallelism cannot significantly help (Dwork, Kanellakis and Mitchell [1984])(Yasuura [1984]). However, this result should not discourage finding a good formulation of unification in parallel logic programming languages.

3.4.4.1. The Smallest Granularity Alternative

We have seen in Section 3.4.3.2 that any information textually specified in a goal must be available when head unification commences. To put it differently, a goal can be called only after all its arguments have been loaded. Thus the clause

```
:- p(a).                                                    (3.4.4–3)
```

and the clause

```
:- p(X), X=a.                                               (3.4.4–4)
```

should be defined as different. The word 'different' may be too strong, but at least we can say that Clause (3.4.4–3) is more restrictive than Clause (3.4.4–4). The above difference is related to the semantics of the unification of a goal and a clause head, so the conclusion of Section 3.4.3.2 should be regarded as claiming also that the following two goals be different:

```
:- Head = p(a).                                             (3.4.4–5)
:- Head = p(X), X=a.                                        (3.4.4–6)
```

This claim clearly rejects Alternative (3).

The following example would better show the importance of whether information is textually specified in a goal or not:

```
:- ..., p(A?), ...                                          (3.4.4–7)
```

We have attached a read-only annotation to `A` not simply because we do not want `A` to be instantiated by '`p`' but because we want the predicate '`p`' to wait and respect the value of `A` for clause selection unless the clause selection can be done without any reference to the argument value. Therefore, we can never rewrite Clause (3.4.4–7) to

```
:- ..., p(X), X=A?, ...                                     (3.4.4–8)
```

even though the 'read-only' property is inherited by unification.

Among important concepts in programming languages is referential transparency, which means that if expressions E_1 and E_2 denote the same value in the same context, we can textually replace E_1 in a program by E_2. Referential transparency in logic programming languages, if any, would require the following property: Whenever a term E appears in some goal, we can replace E by a fresh

variable X and in conjunction with that goal put a new goal which unifies X and E. The above example, however, shows that this property does not hold in Concurrent Prolog.

It has long been claimed that logic programming languages provide a good framework for mechanical handling of programs such as program synthesis and program transformation. So, properties such as referential transparency should be respected as much as possible. Losing it would make the language rules complex and mechanical handling of programs more difficult.

3.4.4.2. Other Alternatives

What Alternative (2) means is as follows. Determination of a principal functor and the setting of its arguments can be done in parallel, but when the principal functor is determined and its arguments become accessible, they must be 'protected' if necessary, that is, if they are to be instantiated further. This solution looks consistent with the requirement that any information textually specified in a goal must be respected for clause selection, while retaining parallelism inherent in unification.

However, we will examine in more detail. The problem is that this solution is rather ad hoc and that it exploits only a limited part of parallelism lost in Alternative (1). Arguments to be filled with non-variable terms can be protected by read-only annotations. However, there seem to be no means to protect the two arguments of 'q' in the following example.

```
:- q(Y, Y).                                    (3.4.4–9)
```

This should not be defined as identical to

```
:- q(A, B), A = B.                             (3.4.4–10)
```

because Clause (3.4.4–9) textually specifies that the two arguments of 'q' be identical, while Clause (3.4.4–10) has moved this information out of the goal. Clause (3.4.4–9) cannot select the clause

```
q(a, b).                                       (3.4.4–11)
```

to reduce itself while Clause (3.4.4–10) can.

Clause (3.4.4–9) is not identical to the following one either:

```
:- q(A?, B?), A = B.                           (3.4.4–12)
```

This is too protective. Clause (3.4.4–9) can select the clause

```
q(a, a).                                       (3.4.4–13)
```

while Clause (3.4.4–12) cannot.

Considering all the above problems, Alternative (1) seems to be the best solution in Concurrent Prolog. When the value of the principal functor is available,

any textually specified information on its arguments should be available also. This means that some sequentiality must be assumed for unification.

The above result urges us to examine the semantics of so-called 'metacall', i.e., a feature for 'call'ing some term (say T) as a goal. The goal T will be possibly incrementally generated by some other goal. So we must have some means to guarantee that all the argument information which should be assumed to be textually specified in the goal has been set up to T. Serialization by the commit operator will have to be used for this purpose. The following examples illustrate this.

Goal (a): `:- p.` (3.4.4–14)

Program (a): `p :- G=p(X), X=5 | call(G).` (3.4.4–15)

Goal (b): `:- p(G), call(G?).` (3.4.4–16)

Program (b): `p(G2) :- G2=p(X), X=5 | true.` (3.4.4–17)

In both of the above examples, it is possible to regard `G` as having the value `p(5)` right after commitment. So the goals `call(G)` and `call(G?)` can be defined to work as if they were specified textually as `p(5)`.

To generalize, the value of a variable which is guaranteed to exist right after commitment can and should be treated as a textually specified value after that. The value of a variable which is guaranteed to exist is the value formed by the bindings made by unifications which is guaranteed to be finished by the language rules. The readers may think that the above discussion is too obvious, but it is never the case. In Clause (3.4.4–16) above, the value of `G` is determined upon commitment by unification. However, as we have seen so far, it is not at all clear whether the principal functor '`p`' and the argument value '`5`' arrives at the goal '`call(G?)`' at the same time or not. This depends on the semantics of commitment and the semantics of unification, both of which are most delicate.

So far we have defined the semantics of unification involving non-variable terms. We must further define the property of logical variables.

We may well be tempted to define the semantics of the goal

```
:- p(X), q(X?).
```
(3.4.4–18)

as equivalent to

```
:- p(X), X=Y, q(Y?).
```
(3.4.4–19)

because they are 'logically' identical. Defining these two goals as identical means that communication by shared logical variables may have potential delay. This delay is allowed in Guarded Horn Clauses described in Chapter 4, but in Concurrent Prolog this cannot be allowed.

Firstly, (Shapiro [1983a]) seems to assume no delay for shared variables, since it contains the specification of binary merger as follows (only recursive clauses are shown):

Goal: `:- merge(As?,Bs?,Cs).` (3.4.4–20)

Program:
```
merge([X|Xs],Ys,[Z|Zs]) :- merge(Ys,Xs?,Zs).   (3.4.4–21)
merge(Xs,[Y|Ys],[Z|Zs]) :- merge(Ys?,Xs,Zs).   (3.4.4–22)
```

If we allow delay, the first argument `Ys` in the body goal of Clause (3.4.4–21) will be instantiated by the head argument '`[X|Xs]`' of the same clause upon recursive call. If there is no delay, it cannot be instantiated because `Ys` has been unified with `Bs?` in the head unification. Secondly, the 'protected data' technique also assumes that there is no delay between two or more occurrences of a shared variable. For otherwise the information of protection would be delayed and hence might be violated.

Therefore, we must not assume any delay for shared variables: All occurrences of the same variable must denote the same value at the same time. We consider that allowing delay for shared variables, even though possible, would considerably change the rules of Concurrent Prolog, which would amount to designing another language.

3.4.5. Processing Heads and Guards

It must be clearly specified what kind of parallelism should be allowed for processing heads and guards. In this regard, the semantics of Concurrent Prolog shown in Section 3.4.2.2 and (Shapiro and Takeuchi [1983]) has the following problems.

3.4.5.1. Head Unification

The rules of Concurrent Prolog do not mention the order of unification of head arguments at all. At least four solutions seem to be candidates:

(1) Head unification is performed in parallel. A pseudo-parallel implementation is allowed, but no sequentiality is assumed conceptually.

(2) Head unification is performed sequentially in some order not defined by the language. The implementation can arbitrarily choose one of the possible orders. A program that depends on a particular order is erroneous.

(3) The mixture of (1) and (2) above. That is, the head unification is performed sequentially in some order, or in parallel. A program that depends on a particular strategy is erroneous.

(4) The head unification is performed sequentially, from left to right.

Solution (1) is preferred because sequentiality in the language rule should be minimized according to the principles stated in Section 3.4.1. The set of possible results of the execution of a program should remain unchanged when we systematically change the order of arguments of some predicate throughout the program and a goal clause. The 'result' mentioned above may include at least the following:

(a) Whether the computation terminates or not,

(b) If it terminates, whether it succeeds or not, and

(c) If it succeeds, what bindings are made to the variables in the goal clause.

Solution (1) influences allowable implementations: Sequential unification from left to right becomes inadequate. Consider the following example:

```
Goal:          :- p(a,  a).                    (3.4.5–1)
Program:          p(A?, A).                    (3.4.5–2)
```

The left-to-right head unification suspends while the rule states it must succeed.

In fact, any implementation of head unification which assumes a specific order is inadequate. For example, there is no specific order in which both of the following two goals succeed:

```
Goal (a):      :- p(A,  A?).                   (3.4.5–3)
Goal (b):      :- p(A?, A ).                   (3.4.5–4)
Program:          p(a,  a ).                   (3.4.5–5)
```

Note that the above arguments apply to any unification performed in computation, for example unification upon commitment (see Section 3.4.2).

3.4.5.2. Head Unification and Guard Execution

The rules of Concurrent Prolog specify that the execution of a guard start after head unification has succeeded (Section 3.4.2.2, Paragraphs [2], [6] and [9]). We propose a different solution: Head unification and the execution of a guard are done in parallel. The reasons follow.

Consider the following example:

```
Goal:          :- p(a,  a).                                (3.4.5–6)
Program (a):      p(A,  B) :- A=X?, B=X | true.            (3.4.5–7)
Program (b):      p(X?, B) :- B=X       | true.            (3.4.5–8)
Program (c):      p(X?, X).                                (3.4.5–9)
```

According to (Shapiro [1983a]), Program (a) succeeds and Program (b) suspends. Program (b) suspends because the unification `B=X` is performed only after the unification `a=X?` has succeeded. The result of (c) is not specified. If the unification of arguments is allowed to be performed sequentially, Program (c) may suspend; if the unification must be performed in parallel (as we recommended in Section 3.4.5.1), it succeeds. However, why should these three programs be not identical?

We propose to define (a) as a standard form of (b) and (c), and to make all the above programs succeed. The standard form of a clause must have a head whose arguments are distinct simple variables. Clauses (b) and (c) are considered

as shorthand of (a). One justification of this proposal is that all these clauses are 'logically' identical. Defining the semantics of a clause in terms of its standard form will simplify the description of the semantics. The similar approach has been adopted in PARLOG (Clark and Gregory [1984a]).

Our proposal could be justified also from a practical point of view. Each clause performs head unification and executes its guard to determine whether it can be selected. Looking back our programming style, we usually write in the head what we can write either in the head or the guard, and we write in the guard only what we cannot write in the head. An example of the former is the syntactic check of an input argument, and an example of the latter is the arithmetic comparison of two input values. As far as we see, this choice has been made solely by the content of the check. There seems to be no program that cannot be written without using the fact that head unification precedes the execution of a guard. The only reason why we use head unification for more than argument passing is that the use of head unification is good for concise description.

3.4.6. Unification of Two Read-Only Variables

Paragraph [3] does not explicitly state the semantics of unification of two read-only variables. Kusalik [1984] took up this subject and argued that if a clause such as

```
p(X?) :- guard( ... X ... ) | body.                    (3.4.6–1)
```

is to be allowed, the head unification invoked by the call

```
:- p(A?).                                              (3.4.6–2)
```

should succeed. Then he proposed two possible revisions:

(a) Let the unification of two read-only variables $X?$ and $Y?$ succeed, and make X and Y (identical) non-read-only variables.

(b) Disallow read-only variables appearing in a head.

However, neither of these solutions is desirable:

(a) Assume that

```
:- X?=Y?, X=a.                                         (3.4.6–3)
```

is executed. If `X?=Y?` is executed first, `X` and `Y` become an identical non-read-only variable. This means that the annotated variable `Y?` becomes instantiated by the partner of the unification by the execution of `X=a`, which is inconsistent with the general property of annotated variables.

(b) As Kusalik himself says, a read-only variable in a head has useful applications (Hellerstein and Shapiro [1984])(Takeuchi and Furukawa [1985]) and should not

be prohibited. Moreover, disallowing read-only variables in a head destroys the symmetric nature of unification.

An alternative solution to (a) might be to let the unification of two read-only variables succeed and then to make them identical read-only variables. This preserves the propagative nature of read-only annotations. However, consider the following goal:

```
:- X?=Y?, X=a, Y=a.
```
(3.4.6–4)

This goal suspends if `X?=Y?` is executed first, and succeeds otherwise. This is a new, undesirable kind of nondeterminism which arises from the order of unification: Besides this, the only source of nondeterminism is the commitment operation.

Shapiro's original interpreter makes the unification of two read-only variables suspend. This solution looks better than any of the above alternatives, but Aida [unpublished] and Tanaka [unpublished] claimed that the unification of two identical read-only variables should succeed. Tanaka implemented Concurrent Prolog (Tanaka, Miyazaki and Takeuchi [1984]) and found that it is inconvenient that the goal

```
:- X=Y?, X=Y?.
```
(3.4.6–5)

suspends while the goal

```
:- X=Y?.
```
(3.4.6–6)

succeeds. So the best solution will be to let the unification of two read-only variables suspend unless these read-only variables are identical, in which case it should succeed.

One consequence of this slight revision is that the implementation might become complicated a little bit. Without the revision, suspension is released only when the read-only variable which caused the suspension gets instantiated to some non-variable term. Now suspension can be released also when the read-only variable that caused the suspension is unified with some other variable, as long as the suspension is caused by unification between two read-only variables.

Let us consider the goal '`X?=Y?`' for example. This goal suspends if `X` and `Y` are both uninstantiated and not identical. One way to release this suspension is to instantiate both `X` and `Y` to non-variable terms. Our claim is that now there is another way to release the suspension, that is, to unify `X` and `Y`. This means that the suspended goal '`X?=Y?`' must watch bindings of `X` and `Y` to uninstantiated variables as well as to non-variable terms.

3.4.7. The Predicate '`otherwise`'

The predicate '`otherwise`' was first introduced in (Shapiro and Takeuchi [1983]). An '`otherwise`' goal that occurs in a guard succeeds if and when all other

parallel-Or guards fail. The commonest use of this construct will be to use only one '`otherwise`' as the sole guard goal of the last clause which handles the 'default' or 'exceptional' case. However, the above simple rule is not so restrictive; it implies the following:

(1) '`otherwise`' can appear in a clause other than the last clause: There is no order among clauses constituting a program.

(2) '`otherwise`' need not appear as the sole goal in a guard. If the guard of a clause containing '`otherwise`' contains other goals, that clause may not be selected even if all the other clauses have failed. However, the clause with '`otherwise`' cannot anyhow cover all exceptional cases without restricting the clause head to the most general one, and there seem to be no reasons to restrict the form of a clause with '`otherwise`'. The only possible restriction might be to prohibit two or more '`otherwise`' goals in one guard, since this is harmless but useless.

(3) '`otherwise`' can appear in more than one guard. Assume that the following clauses are contained in the definition of '`p`'

```
p(nil,       ...) :- otherwise | ...        (3.4.7–1)
p(cons(A,B), ...) :- otherwise | ...        (3.4.7–2)
```

and that all other clauses are proved to be unselectable. Then, if the first argument of '`p`' is '`nil`', Clause (3.4.7–1) will be selected since Clause (3.4.7–2) is unselectable. If the first argument of '`p`' has the form '`cons(X,Y)`', Clause (3.4.7–2) will be selected. Of course, if two clauses which are not mutually exclusive have '`otherwise`' goals in their guards, deadlock may result:

```
p(X, Y) :- otherwise | ...                  (3.4.7–3)
p(X, Y) :- otherwise | ...                  (3.4.7–4)
```

It is, however, the responsibility of a programmer to avoid such deadlock.

Although the original definition may look too general, it is recommended for its simplicity. It is easy to implement: '`otherwise`' needs only to monitor the number of failing clauses and succeeds when it reaches the number of the other clauses. We know that this construct is useful for writing non-trivial programs, but we must examine further where the generality of '`otherwise`' stated above is really useful.

3.4.8. Summary

We have discussed some subtle points on Concurrent Prolog. In Section 3.4.3, we examined the semantics of multiple environments and a commitment operation. We showed that we have to further define at least the following things:

(1) Timing of unification of local and global information,

(2) Availability of the values of goal arguments for clause heads and guards, and

(3) Access rule to local and global copies of variables

In Section 3.4.4, we examined the semantics of unification and the allowed communication delay between two or more occurrences of a shared variable. We saw that a non-variable term (say T) specified in a source text must have an 'indivisible' nature: The unification of T with some variable must be done as an atomic action. To put it differently, we must assume some sequentiality for unification. Another conclusion is that all the occurrences of the same variable must denote the same value at the same time. We must not assume delay for a shared variable which is instantiated by some goal and whose value is referenced by another goal. These conclusions mean that some transformation of a program clause allowed in the declarative reading may change its semantics.

In Section 3.4.5, we considered how to execute head unification and the corresponding guard. In Section 3.4.6, we examined the semantics of unification of two read-only variables. In Section 3.4.7, we examined the semantics of the predicate '`otherwise`'.

It must be noted that the arguments in this section apply also to Flat Concurrent Prolog (Mierowsky, Taylor, Shapiro, Levy and Safra [1985]). Flat Concurrent Prolog is different from Concurrent Prolog in that only predefined test predicates are allowed in guards. Management of nested environments is no longer necessary, and implementation on a sequential machine is greatly simplified by doing clause selection as an indivisible operation. However, Flat Concurrent Prolog never removes the need of multiple environments conceptually, since head unification may instantiate global variables. Therefore, if Flat Concurrent Prolog is intended to be a language for a parallel machine, it must resolve the problems discussed in this section.

The results of this chapter should be helpful for defining the precise semantics of Concurrent Prolog. However, the resulting semantics would be more complex than we thought even if it is defined informally, and it would require considerable efforts to have a formal semantics. An alternative research direction will be to revise the language. The language Guarded Horn Clauses described in Chapter 4 was designed in this direction. It abolished the multiple environment mechanism and read-only annotations at the same time, and has become a much simpler language with slight loss of the expressive power.

Chapter 4

GUARDED HORN CLAUSES

As we examined in Section 2.1, a set of Horn clauses allows procedural interpretation (Kowalski [1974]). It was given a semantics as a sequential programming language by Prolog (Roussel [1975]), and Prolog has proved to be a simple, powerful, and efficient sequential programming language (Warren, Pereira and Pereira [1977]).

As Kowalski [1974] points out and we discussed in Section 3.2, a Horn-clause program allows parallel or concurrent execution as well as sequential execution. However, the discussions in Sections 3.2 and 3.3 revealed that a set of Horn clauses as it stands is inadequate for a parallel programming language which is capable of expressing important concepts such as processes, communication, synchronization, and input/output. We need to control AND-parallelism by some mechanism to express these concepts. Several mechanisms have been proposed as we surveyed in Section 3.3.2, but this chapter shows that only one construct, *guard*, is adequate for our purposes.

In this chapter, we introduce guarded Horn clauses. The name Guarded Horn Clauses (abbreviated to GHC) will be used also as the name of the language. We describe its design principles, syntax, semantics, program examples, and justification of the language design. Moreover, we mention possible extensions and implementation issues. Comparison of GHC with other logic/parallel programming languages will be made in Chapter 5.

4.1. Principle-Oriented Approach to Language Design

When we design a programming language, there will be at least three possible approaches. One is an implementation-oriented approach. A practical language must allow efficient implementation on an existing hardware. Therefore, it must not contain any feature whose implementation is very hard or impossible on an intended hardware, which should be one of the most important constraints on the design of an efficient language. Another is an application-oriented approach. When we have a specific application domain at hand, we desire a good language in which problems to be solved can be elegantly formulated.

The third approach could be called a principle-oriented approach. In a principle-oriented approach, we first establish several principles or constraints on language features to be designed. Of course, these principles could include implementability and/or applicability; the point is that a language design must also have many external and internal constraints which are important solely from a programming language point of view. For example, a theoretical language must have a rigorous mathematical foundation; a general-purpose language should provide expressive power in general; a language for huge software will have to provide facilities

for declaring inter-module specification and for checking it; and any language must be self-consistent—it must include no features whose effects and the effects of whose interaction are not well defined. Principles provide a language designer with a basis on which to design, review, and compare languages. They also help a language user understand the language—why some feature has been designed so and why another is missing.

4.2. Design Principles

Our goal is to obtain a programming language that expresses parallelism and allows parallel execution wherever it is possible. We base our language on Horn-clause logic and a resolution principle in the hope that this assumption leads us to a good solution.

Our language is expected to fulfill the following requirements:

(1) Parallelism. *It must be a parallel programming language 'by nature'. It must not be a sequential language augmented with primitives for parallelism. That is, the language must assume as little sequentiality among primitive operations as possible, in order to preserve parallelism inherent in Horn-clause logic. This would lead to a clearer formal semantics, as well as to an efficient implementation on a novel architecture in the future.*

Note that formulation of the underlying concepts as described in theoretical books may not consider parallelism very well. For example, Robinson's unification algorithm (Robinson [1965]) is sequential and books on mechanical theorem proving do not usually say when and how resolution can be done in parallel. This is why we examined parallel execution of logic programs in Section 3.2.

Two remarks related to implementation must be made. Firstly, the above requirement on parallelism is a requirement at the level of a programming language, and it does not require all the parallelism to be exploited in implementation. It is even desirable that a compiler for a sequential computer generates a sequence of codes for a set of operations which need not be executed in pseudo-parallel. Such sequentiality in implementation is considered an optimization in the framework of parallel languages (see Section 3.1).

The other remark is that a truly parallel language may have to allow possibly useless computation. This is because it seems impossible to distinguish between useful and useless computation without assuming a specific implementation once we allow unrestricted parallelism. An example is the prefetching of instructions done by a processor unit. Although prefetched instructions may be just discarded when a branch instruction is executed, they are usually a source of efficiency and hence prefetching is considered generally useful. If we disallow any useless computation, computation becomes highly sequential and we lose efficiency which is the original purpose of parallelism. Therefore, any computation which *may* be effective must

be allowed. Moreover, since it is unlikely that we can generally judge whether some computation may or cannot be utilized in the future, we have to allow any computation as long as it does no harm.

We go to the next principle.

(2) General-Purpose Language. *It must be an expressive, general-purpose parallel programming language. In particular, it must be able to express important concepts in parallel programming—processes, communication, and synchronization. Moreover, a program must be able to communicate with the outside world, especially with human beings.*

As explained in Section 2.4.9, human-computer communication is important as well as communication within a computer, and it is desirable to treat them uniformly from the following principle.

(3) Simplicity. *It must be a simple parallel programming language. We do not have much experience with either theoretical or pragmatic aspects of parallel programming. Therefore, we must first establish a foundation of parallel programming on a simple language.*

We are not so accustomed to parallel programming or languages for parallel programming. The author's short experience with programming in Concurrent Prolog has shown that parallel programming is not so easy even for those experienced with sequential programming. It is hard to expect what is implied by the interaction of complex language rules.

(4) Architecture Independence and Efficiency. *Our language does not commit itself to any of existing computer architectures. We leave undefined any control mechanism relevant only to efficiency. Nevertheless, our language must allow efficient implementation on a computer of the current technology. We have a lot of simple, typical problems to be described in the language as well as complex ones. It is very important that such programs run as efficiently as the comparable ones written in existing parallel programming languages.*

Efficiency for simple programs is an important property for a general programming language. However, the requirement of simplicity and generality may possibly interfere with the efficiency: Many useful programs may share the important pragmatics that could be implemented more efficiently under some linguistic support. Therefore, it is meaningful to identify an interesting subclass of the language within which such pragmatics is efficiently supported and to get a more specific but more efficient language. One reservation is that such subsetting must be done only for efficiency purposes—a program in the subset must also be a program in the fullset. Subsetting is often said to be undesirable. However, if we anyhow need both an expressive language and a (very) efficient language, the better choice should be to design an expressive language first and to appropriately restrict it to get an efficient

language. It would be very difficult to design an efficient one first and extend it naturally.

Moreover, our language specification should satisfy the following general principles.

(5) Implementability. *There should be no language features that are not implementable or are very hard to implement. If such rules were to exist, the blame should be laid on the language specification.*

(6) Compelling Force of Language Rules. *There should be as few rules as possible whose violation cannot be easily detected at compile time or at run time. These would promote erroneous programs to circulate.*

The following are more specific guidelines.

(7) *As we stated in the beginning of this section, we base our language on Horn-clause logic and the resolution principle and try to utilize the previous results on logic programming and parallel logic programming. Moreover, we try to preserve the properties of the original framework of logic programming as much as permitted by the other guidelines.*

For example, 'logical' transformation of a program clause must preserve its semantics to the maximal extent. Preserving the underlying formalism is desirable also in the sense that it meets the requirement of simplicity.

(8) *We consider input and output as a boundary condition of language design. More specifically,*

(8a) *we apply the basic idea of logic programming that a predicate defines a relation between data to the whole program as well as internal procedures, and*

(8b) *we use the same mechanism for interprocess communication and for human-computer communication.*

These points were discussed in Sections 2.3.3 and 3.3.2. Item (8a) claims that input and output must be performed by means of streams. Item (8b) claims that all communication data must be determinate.

Finally, we note that we are proposing basic concepts rather than a complete programming language for practical use; our language has a tinge of a computational model in this sense. Nevertheless, we believe that our proposal will provide a basis for a programming language for describing large software. Issues of user-oriented language features will be discussed in Section 8.3.2.

4.3. Syntax

A GHC program is a set of guarded Horn clauses of the following form:

$$H \ :\!- \ G_1, \ \ldots, G_m \mid B_1, \ \ldots, B_n . \qquad (m > 0, \ n > 0).$$

where H, G_i's, and B_i's are atomic formulas as defined in Section 2.1. H is called a clause head, G_i's are called guard goals, and B_i's are called body goals. The connectives '`:-`' and '`,`' are common to ordinary Horn clauses. The only difference from Horn clauses is that one of the conjunctive operators is replaced by a commitment operator '`|`'. The part of a clause before '`|`' is called a guard, and the part after '`|`' is called a body. Note that *a clause head is included in a guard.* Declaratively, the commitment operator denotes conjunction, and the above guarded Horn clause is read as "H is implied by $G_1, \ldots, G_m$ and $B_1, \ldots, B_n$".

To start a GHC program, we use a goal clause of the following form

$$\texttt{:-}\ B_1\texttt{,}\ \ldots\texttt{,}\ B_n\texttt{.} \qquad (n > 0)$$

as to which there are no changes from the original framework.

The nullary predicate '`true`' is used for denoting an empty set of guard or body goals explicitly in a program. This is used only for notational convenience and need not be considered as primitive. Actually it could be defined as follows:

```
true :- 1=1 | 1=1.
```

One binary predicate, '`=`', is predefined by the language. The predicate '`=`' is used for unifying two terms. This predicate should be considered as predefined, since it cannot be defined in the language for a syntactical reason.

4.4. Semantics

The semantics of GHC is quite simple. Roughly speaking, to execute a program is to refute (Section 2.1.3) a given goal clause by means of parallel input resolution (Section 3.2.2) using the clauses in the program. Of course, some adaptations must be made according to the syntactical changes. Firstly, a proof tree constructed by parallel input resolution must have an arc corresponding to each guard goal, as well as each body goal, of the clause represented by the parent node. Moreover, we must define the treatment of the predefined predicate '`=`': It is treated as if it were defined by the following ordinary Horn clause:

```
X = X.
```

Then, we give the semantics of the guard by restricting the above parallel input resolution.

As the simplest way to meet Requirement (8) of Section 4.2, we treat the construction of all proof trees in a single environment rather than independently. That is, we disallow unification in two proof trees to instantiate the same variable to different values. Any substitution applied to a variable must be common or global throughout the execution of the program.

To achieve the above goal while letting the language be still useful, we impose the following rules of suspension:

- *Rules of Suspension*

  (a) Unification invoked directly or indirectly in the guard of a clause C called by a goal G (i.e., unification of G and the head of C and any unification invoked by executing guard goals of C) cannot instantiate (see Section 2.1.2) the goal G.

  (b) Unification invoked directly or indirectly in the body of a clause C cannot instantiate the guard of C *until that clause is selected for commitment* (see below).

  A piece of unification that can succeed only by making such bindings is suspended until it can succeed without making such bindings *(end of the rules of suspension)*.

Another rule we must have is *the rule of commitment*. When some clause C called by a goal G succeeds in solving (see below) its guard, the clause C tries to be selected for subsequent execution (i.e., proof) of G. To be selected, C must first confirm that no other clauses in the program have been selected for G. If confirmed, C is selected indivisibly, and the execution of G is said to be committed to the clause C.

We say that a set of goals *succeeds* (or is *solved*) if the proof procedure succeeds in constructing for them a non-ordered refutation whose non-root nodes all represent selected or ordinary clauses. We do not say that it is solved even if the proof procedure constructs a non-ordered refutation with a non-root node representing a non-selected clause: We are interested in a proof in which only selected clauses are involved. We do not have the notion of failure, though we could introduce it. This point will be discussed in Section 4.10.

It must be stressed that under the rules stated above, anything can be done in parallel: Conjunctive goals can be solved in parallel; candidate clauses called by a goal can compete in parallel for commitment; unification of a goal and the head of a candidate clause can be done in parallel, both internally and with the execution of guard goals.

However, it must be even more stressed that we can also execute a set of operations in a predetermined order as long as it does not change the meaning of the program. The only possible difference between sequential and parallel execution is that sequential execution may fail to solve a set of goals which can be solved by parallel execution. Therefore, the serialization is allowed only if there is no such possibility.

Note that in spite of the above notes on parallelism, instantiation of a variable must of course be done as an indivisible operation, because otherwise the properties of unification would no longer hold. The selection of a clause upon commitment must also be done as a indivisible operation.

The rules of suspension could be restated as follows:

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

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

Rule (a) is used for synchronization, so it could be called *the rule of synchronization*. Rule (b) is rather tricky; it states that we can execute the body of a non-selected clause, whose result may prove to be useless afterwards. However, the above restrictions guarantee that this look-ahead computation never affects the selection of candidate clauses nor the other goals running in parallel with the caller of the clause. So Rule (b) is effectively *the rule of sequencing* between the guard and the body of a clause.

Let us compare the above rules of GHC with those of Concurrent Prolog discussed in Section 3.4. In Concurrent Prolog, unification which is performed in a guard (including a head) and which would export bindings pretends to succeed by recording the resultant bindings locally. In GHC, such unification simply suspends. Suspension of unification may, but does not always, be released when the variable that caused suspension is bound to some variable or some non-variable term by the unification invoked in the body of some selected clause.

An example may be helpful in understanding the rules of suspension. Let us consider the following program:

Goal clause:	`:- p(X), q(X).`	(4.4–1)
Program clauses:	`p(ok) :- true | ... .`	(4.4–2)
	`q(Z) :- true | Z=ok.`	(4.4–3)

Two goals in Clause (4.4–1) can be executed in parallel, but we assume that `p(X)` is executed first. Then the unification is attempted between the goal `p(X)` and the head `p(ok)` of Clause (4.4–2), but this unification cannot instantiate `X` to the constant '`ok`', since it is invoked in the guard: Clause (4.4–2) must wait until `X` is instantiated to '`ok`' by the goal `q(X)`. On the other hand, the goal `q(X)` can be unified with the head `q(Z)` of Clause (4.4–3) without instantiating `X`. So Clause (4.4–3) can be selected and the goal '`Z=ok`' instantiates `Z` and hence `X` to '`ok`'. Then the head unification of Clause (4.4–2) succeeds and Clause (4.4–2) is selected.

In short, Clause (4.4–2) can be selected only after the goal '`Z=ok`' is executed no matter which of the two goals in Clause (4.4–1) starts first. The goal `q(X)` acts as the producer of the value of `X` and the goal `p(X)` acts as the consumer. Thus we have introduced causality into logic programming.

The semantics of the following program should be more carefully understood:

Goal clause: `:- p(X), q(X).` (4.4–4)

Program clauses: `p(Y) :- q(Y) | ... .` (4.4–5)

`q(Z) :- true | Z=ok.` (4.4–6)

To solve the guard of Clause (4.4–5), we have to do two things in parallel: unifying `p(X)` and `p(Y)` (i.e., parameter passing), and solving `q(Y)`. Let us first assume that parameter passing is executed first. Then the goal '`Z=ok`' tries to instantiate `Z`, `Y` and `X`, which are now identical, to '`ok`'. However, it must suspend because it is indirectly invoked in the guard of Clause (4.4–5). Let us then consider the other case where the goal `q(Y)` is executed prior to parameter passing. The variable `Y` is successfully bound to '`ok`' because this does not export binding to the goal `p(X)`. However, this binding makes the subsequent parameter passing suspend because it would bind `X` to '`ok`'. Hence, no matter which case actually happens, Clause (4.4–5) behaves equivalently to Clause (4.4–2) as for the binding given to the variable `X`.

4.5. Important Properties

In this section, we list the important consequences of the above semantics.

Consequence 1. *Any unification intended to 'export' bindings to the caller of a clause through its head arguments must be specified in the body of the clause. Such unification must be specified by using the predefined predicate '='. The programming style of GHC differs from that of Prolog in this point.*

Consequence 2. *Unification between a clause head and its caller may, but need not, be executed in parallel. It can be executed sequentially in any predetermined order.*

This would require justification. It suffices to show that when the unification of a clause head H and its caller G suspends trying to make a binding to a variable X in sequential execution, it does not succeed in parallel execution at least until X is instantiated to some term. We consider a *frozen counterpart* of G. A frozen counterpart G' of an expression G with respect to a variable X is an expression obtained by instantiating X to a fresh constant. Then, sequential unification (without dataflow restriction) between G' and H *fails* obviously. However, since success or failure of ordinary unification should not depend on the specific algorithm employed, any correct parallel unification algorithm must fail to unify G' and H. Any parallel unification with dataflow restriction between G and H has an obvious counterpart in ordinary parallel unification between G' and H, and the former detects suspension or failure where the latter detects failure. This means that the former never succeeds at least while X is left as it is.

Consequence 3. *Unification between the head of a clause C and its caller and the execution of the guard goals of C can be executed in parallel. That is, the execution of the guard goals can start before the head unification has completed. However, the usual way of execution that solves the guard goals only after the head unification is also allowed.*

The first half is obvious from the definition of parallel input resolution. The second half can be shown in the same manner as Consequence 2 as outlined below. A proof tree to solve each guard goal defines a set of equations which must be solved in conjunction with head unification without instantiating the caller. Therefore, the only difference from the proof of Consequence 2 is that a set of equations to be solved is expanded by the guard goals. We can consider a frozen counterpart of the caller of the clause as before, and can justify sequentiality between the head and the guard goals.

CONSEQUENCE 4. *The execution of the body of a clause may, but need not, start before that clause is selected for commitment.*

This was explained in Section 4.4.

CONSEQUENCE 5. *We need not implement a multiple environment mechanism, a mechanism for binding a variable with more than one value.*

This mechanism is in general necessary when more than one candidate clause for a goal is tried in parallel. GHC, however, avoids it by allowing at most one clause, a selected clause, to export bindings for each caller. This restriction is one of the reasons why the operational semantics of GHC is incomplete as a theorem prover, as will be discussed in Section 4.9.

Unfortunately, Consequences (2) and (3) do not hold if we introduce the concept of failure; see Section 4.10.

4.6. Program Examples

This section illustrates some example programs. Besides this section, Chapter 7 shows how to program—possibly automatically—search problems in GHC.

4.6.1. Binary Merge

```
merge([A|Xs],Ys,    Zs) :- true | Zs=[A|Zs1], merge(Xs,Ys,Zs1).
merge(Xs,    [A|Ys],Zs) :- true | Zs=[A|Zs1], merge(Xs,Ys,Zs1).
merge([],    Ys,    Zs) :- true | Zs=Ys.
merge(Xs,    [],    Zs) :- true | Zs=Xs.
```

The goal '`merge(Xs,Ys,Zs)`' merges two streams `Xs` and `Ys` (implemented as lists) into one stream `Zs`. This is typical of nondeterministic programs: When both `Xs` and `Ys` have been instantiated, the element of either stream may appear first in `Zs`. The language rules of GHC do not state that the selection of clauses must be fair. That is, an implementation may always choose the first clause when both the first and the second clauses are selectable. In a good implementation, however, the elements of `Xs` and `Ys` is expected to appear in `Zs` almost in the order of arrival. The above treatment of the fairness problem is the same as in CSP (Hoare [1978]).

Note that the declarative reading of the above program gives the usual, logical specification of the nondeterministic merge—arbitrary interleaving of the two input streams makes the output stream. A difference between the GHC program and the logical specification is that a GHC program expresses the direction of computation or causality while the logical specification does not. Another difference is that the execution mechanism of GHC chooses as the value of `Zs` arbitrary one of the possible interleavings of `Xs` and `Ys` and does not compute the others.

4.6.2. Generating Primes

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

gen(N,Max,Ns) :- N=<Max | Ns=[N|Ns1], N1:=N+1, gen(N1,Max,Ns1).
gen(N,Max,Ns) :- N> Max | Ns=[].

sift([P|Xs],Zs) :- true | Zs=[P|Zs1], filter(P,Xs,Ys), sift(Ys,Zs1).
sift([],    Zs) :- true | Zs=[].

filter(P,[X|Xs],Ys) :- X mod P=:=0 |             filter(P,Xs,Ys).
filter(P,[X|Xs],Ys) :- X mod P=\=0 | Ys=[X|Ys1], filter(P,Xs,Ys1).
filter(P,[],    Ys) :- true        | Ys=[].
```

The goal '`primes(Max,Ps)`' returns through `Ps` a stream of primes up to `Max`. The stream of primes is generated from the stream of integers generated by '`gen(2,Max,Ns)`' by filtering out multiples of primes. For this purpose, a goal '`filter(P,Xs,Ys)`' is generated for each prime `P`, which filters out multiples of `P` from the stream `Xs` and yields `Ys`. Since all integers in `Xs` are guaranteed to be non-multiples of any prime less than `P`, the first element of `Ys` is a prime succeeding `P`. When it is obtained, a new filter goal is created and cascaded.

Note that the above program is simple-minded. For example, the goal '`filter(101,Xs,Ys)`' need not filter out anything until it finds in the input stream `Xs` an integer greater than 10201 ($= 101^2$).

The binary predicate '`:=`' evaluates its right-hand side operand as an integer expression and unifies the result with the left-hand side operand. The binary predicate '`=:=`' evaluates its two operands as integer expressions and succeeds iff the results are the same. These predicates cannot be replaced by the predicate '`=`' because '`=`' never evaluates its arguments. The predicates '`=\=`', '`=<`', '`>=`', '`<`' and '`>`' are also used for comparison, whose meanings should be obvious.

Although these predicates will be provided as system predicates, we need not think of them as illogical or something not definable in the language. For example, we could enumerate all the clauses expressing possible input-output relations of '`:=`' which include the following:

```
X:=0 :- true | X=0.  X:=0+0 :- true | X=0. ...
X:=1 :- true | X=1.  X:=0+1 :- true | X=1. ...
X:=-1 :- true | X=-1.  ...
...
```

Also, the predicate '=\=' could be defined as

```
X=\=Y :- true | A:=X, B:=Y, A<>B.
```

where the predicate '<>' could be defined by enumerating all clauses of the form

α <> β :- true | true.

where α and β denote non-identical integers.

4.6.3. Generating Primes by Demand-Driven Computation

```
primes(Ps) :- true | gen(1,Ns), sift(Ns,Ps).

gen(M,[N|Ns1]) :- true | M1:=M+1, N=M1, gen(M1,Ns1).
gen(M,[]     ) :- true | true.

sift(Xs,[Z|Zs1]) :- true |
        Xs=[Z|Xs1], filter(Z,Xs1,Ys), sift(Ys,Zs1).
sift(Xs,[]     ) :- true | Xs=[].

filter(P,Xs,[Y|Ys1]) :- true | Xs=[X|Xs1], filter2(P,X,Xs1,Y,Ys1).
filter(P,Xs,[]     ) :- true | Xs=[].
filter2(P,X,Xs1,Y,Ys1) :- X mod P=:=0 |      filter(P,Xs1,[Y|Ys1]).
filter2(P,X,Xs1,Y,Ys1) :- X mod P=\=0 | Y=X, filter(P,Xs1,Ys1    ).

ask(IOs,Ps) :- true | IOs=[read(M)|IOs1], ask2(IOs1,Ps,M).
ask2(IOs1,Ps,M) :- M > 0 | Ps=[P|Ps1], ask3(IOs1,Ps1,M,P).
ask2(IOs1,Ps,M) :- M=:=0 | ask(IOs1,Ps).
ask2(IOs1,Ps,M) :- M < 0 | IOs1=[], Ps=[].
ask3(IOs1,Ps1,M,P) :- wait(P) |
        IOs1=[write(P),nl|IOs2], M1:=M-1, ask2(IOs2,Ps1,M1).

test :- true | instream(IOs), ask(IOs,Ps), primes(Ps).
```

This program illustrates the general statement that demand-driven computation can be implemented by means of data-driven computation. It is the demand-driven version of the prime generator program in Section 4.6.2 with a user interface. The program in Section 4.6.2 is data-driven and there is no means to control the generation of `Ps` once '`prime(Max,Ps)`' is called. Moreover, it requires the upper bound of primes to be generated prior to execution. On the other hand, the predicate '`primes`' in the demand-driven version generates primes on demand.

Let us examine the program. When the goal '`primes(Ps)`' is executed, `Ps` is passed to the second argument of the goal '`sift(Ns,Ps)`'. However, the predicate

'`sift`' examines the second argument `Ps` in its guard, so the head unification suspends until `Ps` is instantiated by the goal '`ask(IOs,Ps)`'. The goal '`gen(1,Ns)`' also suspends for the second argument `Ns`.

Assume here that `Ps` is instantiated to the form '`[_|_]`'. Then the first clause of '`sift`' is selected for the goal '`sift(Ns,Ps)`' and instantiates `Xs` and hence `Ns`. Then the first clause of '`gen`' is selected for the goal '`gen(1,Ns)`' and it instantiates `N` to 2, which also becomes the value of the first element of `Ps`.

Seen from outside, the goal '`prime(Ps)`' fills the undetermined element of a given list structure with a new prime, but it does not create the skeleton of the list structure by itself. Thus instantiation of (the sublist of) `Ps` to the form `[_|_]` is regarded as a demand to the goal '`prime(Ps)`'.

The above program provides a user interface goal '`ask(IOs,Ps)`' to run the prime generator explained above. The goal '`ask(IOs,Ps)`' reads an integer M from the terminal, sends M demands to the '`prime(Ps)`' goal displaying each obtained prime before sending the next demand, again reads a new integer, and so on. Here we have used the declarative input/output predicate '`instream`' provided by the compiler described in Section 6.2. The predicate '`instream`' takes a stream of input/output commands as its argument, of which we used '`read(N)`' for reading an integer, '`write(N)`' for displaying an integer, and '`nl`' for beginning a new line. These commands are processed in the order in which they appear in the stream, so we have precise control over input and output.

When the goal '`ask(IOs,Ps)`' reads a negative integer, it closes its argument streams `IOs` and `Ps` and terminates itself.

An important notice follows: Although this program has full control over the generation of primes, it is never a good program from a viewpoint of efficiency since it has lost most of the parallelism inherent in the data-driven version. For efficiency, we must allow possibly useless computation to some extent. One way to do this will be to use the bounded buffer technique described in Section 4.6.4. Another way will be to give up fully demand-driven computation and to incorporate into the data-driven program a mechanism for controlling the generator of the integer stream.

4.6.4. Bounded Buffer Stream Communication

```
test(N) :- true | buffer(N,Hs,Ts), ints(0,100,Hs), consume(Hs,Ts).

buffer(N,Hs,Ts) :- N > 0 | Hs=[_|Hs1], N1:=N-1, buffer(N1,Hs1,Ts).
buffer(N,Hs,Ts) :- N=:=0 | Ts=Hs.

ints(M,Max,[H|Hs]) :- M< Max | H=M, M1:=M+1, ints(M1,Max,Hs).
ints(M,Max,[H|_ ]) :- M>=Max | H=eos.

consume([H|Hs],Ts) :- H\=eos | Ts=[_|Ts1], consume(Hs,Ts1).
consume([H|Hs],Ts) :- H =eos | Ts=[].
```

This program shows that we can program a bounded buffer in GHC, an idea first shown by Takeuchi and Furukawa [1983] [1985] in a logic programming framework. The goal '`ints(0,100,Hs)`' is a demand-driven generator of a stream `Hs` of integers. The demands are generated by the goal '`consume(Hs,Ts)`', which tries to keep `N` unanswered requests issued. For this purpose, we use the notion of a difference list. The goal '`buffer(N,Hs,Ts)`' initially generates a difference list of length `N` whose head and tail are `Hs` and `Ts`, respectively. The goal `consume(Hs,Ts)` receives both its head and tail, and instantiates the current tail, i.e., the uninstantiated part of the stream, whenever a new element in `Hs` is obtained and examined in the guard of `consume`.

The binary predicate '`\=`' is the negation of the predicate '`=`'. It succeeds when its two arguments turn out to be ununifiable; it suspends until then. This predicate, again, need not be thought of as predefined. To check whether some term is ununifiable with the constant '`eos`', we can prepare the following clause

X`\=eos :- true | true.`

for every most general term X whose principal function symbol is distinct from '`eos`'.

4.6.5. Meta-Interpreter of GHC

```
call(true ) :- true | true.
call((A,B)) :- true | call(A), call(B).
call(A=B  ) :- true | A = B.
call(A    ) :- A\=true, A\=(_,_), A\=(_=_) |
     clauses(A,Clauses), resolve(A,Clauses,Body), call(Body).

resolve(A,[C|Cs],B) :- melt_new(C,(A:-G|B2)), call(G) | B=B2.
resolve(A,[C|Cs],B) :- resolve(A,Cs,B2) | B=B2.
```

This program is basically a GHC version of the Concurrent Prolog meta-interpreter by Shapiro [1984]. The predicate '`clauses`' is a system predicate which returns in a *frozen* form (Nakashima, Ueda and Tomura [1984]) a list of all clauses whose heads are potentially unifiable with the given goal; it can omit those program clauses which have turned to be unselectable forever, though a simple implementation may return *all* program clauses. Each frozen clause is a ground term in which original variables are indicated by fresh constant symbols called a frozen variable and in which two connectives '`:-`' and '`,`' are represented by corresponding function symbols; it is *melted* in the guard of the first clause of '`resolve`' by '`melt_new`'. The goal '`melt_new(C,(A:-G|B2))`' creates a new term (say T) from the frozen term `C` by giving a new variable for each frozen variable in `C`, and tries to unify T with '`(A:-G|B2)`'.

The predicate '`resolve`' tests the candidate clauses and returns the body of arbitrary one of the clauses whose guards have been successfully solved. This many-

to-one arbitration is realized by the nest of binary clause selection performed in the predicate '`resolve`'.

It is essential that each candidate clause is melted after it has been brought into the guard of the first clause of '`resolve`'. If it were melted before passed into the guard, all variables in it would be protected against instantiation from the guard.

One minor problem with the above meta-interpreter is that the above meta-interpreter does not allow the body of a candidate clause to be executed before that clause is selected, contrary to the semantics in Section 4.4 (Levy [1985]). Only the body of the selected clause is '`call`'ed in the body of the fourth clause of '`call`'. The discrepancy will be due to the syntactically restrictive way in which we provided the function of commitment. Of course, practically, there should be no problem at all.

4.7. Primitive Operations of GHC

This section makes some remarks on the primitive operations of GHC.

4.7.1. Resolution

GHC uses parallel input resolution (Section 3.2.2) as its basic computational mechanism. Resolution in the usual sense (Robinson [1965]) contains selection of a program clause, head unification and goal rewriting, but parallel input resolution reveals that they need not form an indivisible operation. Head unification can be executed in parallel with the newly created goals.

4.7.2. Unification and Anti-Substitutability

GHC allows all equations appearing in computation to be executed in parallel, though an actual implementation may exploit sequentiality. We will examine what must be considered as primitive operations in unification.

Given a set of equations S, we consider a new set of equations $S' \cup \{X = T\}$ where S' is obtained by replacing some occurrence of a term T by a fresh variable X. The reverse operation of this, which may appear in any unification algorithm, may be called *substitution*; therefore we will call our transformation *anti-substitution*. Clearly, anti-substitution does not change the solution of S (i.e., the most general substitution of all the substitutions which simultaneously solves the equations in S) except that the solution of $S' \cup \{X = T\}$ binds X to T.

We define anti-substitution of a guarded Horn clause in a similar way. Given a program clause, anti-substitution makes a new clause by replacing an occurrence of some term T in the guard/body by a fresh variable X and adding the goal $X{=}T$ in that guard/body, respectively. This transformation does not change the result of a program; that is, it does not change the answer substitution or causality of bindings.

It suffices to show that the anti-substitution does not affect the restriction to the parallel input resolution imposed by GHC, since obviously it does not affect the result of the original parallel input resolution except for the binding to X. Anti-substitution can be divided into two cases: One is the case where a term in a clause head is replaced, and the other is the case where a term in a guard goal or a body goal is replaced. Firstly, we assume that the head H of a clause C is rewritten to H' and the goal $X{=}T$ is added in the guard, yielding a new clause C'. For the clause C to be selected for the caller G, we must solve $G{=}H$ in conjunction with the guard goals without instantiating G. For the clause C' to be selected for the caller G, we must solve $G{=}H'$ and $X{=}T$ in conjunction with the original guard goals without instantiating G. By considering a frozen counterpart of G (see Section 4.5) for the case of suspension, it is understood that the clause C is selectable if and only if the clause C' is selectable.

Secondly, we assume that some goal G is rewritten to the conjunction of goals "$G', X = T$". Then the only difference between G and G', if any, is that the latter explicitly states that the bindings necessary to select some clause for $G^{(\prime)}$ may be postponed; G' eventually becomes G and during this transition candidate clauses for G' never instantiates G' for the rules of suspension. Therefore, this rewriting does not affect the selection of a clause.

We note that anti-substitutability holds for a goal clause as well as program clauses. This property well meets Requirement (7) of Section 4.2. It can be considered as a kind of referential transparency in a logic programming framework, as we discussed in Section 3.4.4.

Anti-substitutability is helpful for considering the primitive operations of unification in GHC. It seems natural to admit anti-substitutability *operationally* as well as *declaratively*, since the computational mechanism should be left to implementation as long as it brings the correct result. If we admit it, the following consequences follow:

(1) The goal `X=f(a)` is equivalent to `X=f(Y),Y=a`. Therefore, `X` might not be instantiated to a ground term instantaneously.

(2) A conjunction of communicating goals '`p(X),q(X)`' is equivalent to the conjunction '`p(X),X=Y,q(Y)`'. Therefore, the occurrences of `X` in the original conjunction may not have the same value at the same time (of any observer). In other words, communication by a shared variable may have a potential delay.

Item (1) is pointed out also by Hagiya [1983].

These consequences suggest freedom in implementation. Of course, the *actual* set of primitive operations is determined by the unification algorithm employed. These consequences also provide a good reason why extralogical predicates such as '`var`' and '`==`' of Prolog do not make any sense in GHC.

Under anti-substitutability, the metacall facilities proposed by Clark and Gregory [1984b] have a semantical problem. Their two-argument metacall '`call(Goal, Result)`' tries to solve `Goal` possibly generating output bindings, and it unifies `Result` with '`succeeded`' upon success and with '`failed`' upon failure. We have not introduced the notion of failure, but there is no problem in introducing it for unification goals.

Consider the following example (Sato and Sakurai [1984]):

```
:- call(X=0, _), X=1.
```

If the first goal is executed first, `X` becomes `0`. Then the unification `X=1` fails and so does the whole clause. If the second goal is executed first, `X` becomes `1`. But since the first goal never fails, the whole clause succeeds. This is a new kind of nondeterminism resulting from the order of unification; without this feature, all nondeterminism would result from the arbitrary choice of selectable clauses.

Let us consider another example:

```
:- call(X=0, _), call(X=1, _).
```

Using anti-substitutability, we can rewrite the above goal clause to:

```
:- call(X=0, _), X = Y, call(Y=1, _).
```

However, this rewriting shows that the failure of unification cannot be confined in either '`call`'. The failure can creep out and topple the whole goal. This means that the metacall facilities as proposed by Clark and Gregory cannot protect a system program from unpredictable behavior of a user program.

4.7.3. Commitment

The body of a clause selected for commitment becomes free from the dataflow restriction by the rules of suspension. However, the restriction need not be released indivisibly with the selection of the clause. It may be released after commitment and in parallel with the execution of the body goals.

4.8. Process Interpretation of GHC

Although GHC is considered as a simplification of previous parallel logic programming languages, it also has close relationship to programming languages under other categories. For example, it can be viewed as a generalization of nondeterministic dataflow languages and Communicating Sequential Processes (Hoare [1978]). This section shows that GHC provides a flexible framework for describing processes, communication, and synchronization. Detailed comparisons with other programming languages will be made in Chapter 5.

A program clause of GHC can be viewed as a rewrite rule of the goal clause as in Prolog. For example, let us consider the first clause of '`merge`' shown in Section 4.6.1:

```
merge([A|Xs],Ys,Zs) :- true | Zs=[A|Zs1], merge(Xs,Ys,Zs1).
```

This clause claims that it can rewrite its caller by its two body goals if the caller satisfies a certain condition, or more specifically, if the first argument of the caller is instantiated to the form '`[_|_]`'. Generally speaking, a goal, given sufficient bindings from other conjunctive goals, reduces itself to a set of other goals, and by doing so generates new bindings. Thus conjunctive goals can be viewed as processes interacting with one another by means of the bindings to shared variables.

The process interpretation of GHC is given by the following correspondence:

A system of processes	$\longleftrightarrow$	*Conjunctive goals*
A process	$\longleftrightarrow$	*A goal*
Process state	$\longleftrightarrow$	*The set of arguments of the goal*
Computation	$\longleftrightarrow$	*Goal rewriting and unification*
Communication	$\longleftrightarrow$	*Instantiation of a shared variable by a unification goal in a body and observation of the generated binding by unification in a guard*
Synchronization	$\longleftrightarrow$	*The rule of synchronization*
Choice nondeterminism	$\longleftrightarrow$	*The rule of commitment*
The (possibly recursive) definition of a process	$\longleftrightarrow$	*A predicate*

Note that when a process G is reduced to other processes, we need not think of G as disappeared; the process G has only committed its remaining tasks to those processes and it continues to exist conceptually until all its subprocesses have terminated.

The process interpretation of logic was first presented by van Emden and de Lucena [1982], and it was presented by Shapiro [1983a] in the context of a concrete programming language. GHC allows a process interpretation most naturally, since it has the clearest notion of causality among other logic programming languages.

GHC as a process description language has the unique feature of flexibility. It allows dynamic generation and deletion of processes, including recursive process generation. It allows dynamic creation of data structures. It allows demand-driven computation as well as data-driven computation. Moreover, we can 'declaratively' handle mutable objects such as arrays and databases by implementing them as processes and by using transaction streams as the interface (see Section 6.1).

GHC uses shared variables for communication; however, this communication scheme is quite different from the shared-variable communication in procedural languages. A procedural language requires complex mechanisms and protocols for synchronization and communication because variables do not have a single-assignment property. GHC variables, on the other hand, have a single-assignment property; their values are determined possibly gradually as computation proceeds, but they never *change* non-monotonically. Therefore, we can make necessary synchronization based on how much information is known on the value of a variable. Synchronization based on the availability of data is the basic idea of dataflow languages. However, availability is treated as binary information in early dataflow languages, while GHC provides a more general and uniform information structure, namely a term, which naturally contains the notion of the availability of data.

Although GHC uses shared variables for communication, we can also use the message-passing paradigm in writing a GHC program; GHC is expressive enough to model message passing thanks to the single-assignment property of variables. Thus, GHC and its ancestors could be viewed as proposing a new computation and communicational model which is hard to be categorized into any of the existing models. It is hoped that the simplicity of GHC will make it accepted as a parallel computational model as well as a programming language.

4.9. Justification of the Language Design

GHC is designed by adapting Horn-clause logic and the resolution principle for a parallel programming language. Although GHC can be given an interpretation of its own apart from the original framework of logic programming as we showed in Section 4.8, it should be meaningful to try to justify our adaptation:

(1) While logical formulae express no causality, computation obviously connotes direction. Therefore, it is not unnatural for a programming language to have a notion of direction or causality. Since the result of computation is constructed by unification in logic programming, a natural way of introducing causality is to introduce it among unification operations.

(2) Bindings to variables must be determinate for the reason shown in Section 4.2. Therefore, candidate clauses for a goal must not instantiate the goal while there is more than one such candidate. It might be possible to employ a more generous method than the rules of suspension to guarantee determinacy; however, the author is quite doubtful whether the increase of the expressive power makes up the loss of simplicity.

(3) In addition to the rules of suspension, we must have a rule for allowing at most one clause to export bindings in order to construct the result of computation. Thus we need a mechanism to select one clause *reasonably*. We employed the rule of commitment which involves nondeterministic choice for this purpose. Nondeterministic choice is necessary to interface with the outside world which

we are obliged to treat as nondeterministic whether or not it is actually so (Clinger [1981], p. 52). Even without interaction with the outside world, nondeterminism is necessary at the front end of a shared service process (e.g., a manager of internal symbols) which serves its clients in the order of the arrival of requests. Use of success and choice nondeterminism rather than success and failure (as in Qute; see Section 5.3) is also closely related to the fact that the semantics of GHC is defined without using any notion of failure. Computation proceeds based only on positive information.

(4) Because of the adaptation, the operational semantics of GHC has lost completeness as the theorem prover for Horn-clause logic, though of course it retains soundness. The incompleteness has been introduced to satisfy the requirement of determinate bindings, not for the reason of efficiency as Shapiro [1983a] and Gregory [1985b] state. Note that although GHC is incomplete in general, it must be complete with respect to a nontrivial class of programs and goals, which is yet to be identified.

The semantics given to the guard construct is powerful enough to express synchronization, conditional branching and choice nondeterminism, and contributes very much to the simplicity of the language. The Relational Language (Clark and Gregory [1981]) was the first to introduce the guard concept to logic programming for the purpose similar to ours (IC-Prolog (Clark and McCabe [1980]) was the first to introduce the guard concept to logic programming, but the purpose was rather different). GHC has removed the restrictions on the guard of the Relational Language together with mode declarations and annotations.

Note, however, that the notation '`|`' we introduced for expressing guards is rather arbitrary. We could employ *any* notation as long as it can distinguish between restricted and unrestricted unification directly or indirectly invoked by a clause. For example, we could specify first clause of '`merge`' as

$$merge([\mathbf{A} \;/\; \mathbf{Xs}], \mathbf{Ys}, [\mathbf{A} \mid \mathbf{Zs1}]) \;\; :\!- \; \mathbf{merge(Xs,\ Ys,\ Zs1)}.$$

and the predefine predicate '`=`' as

$$\mathbf{X} = \mathbf{X}.$$

where the italicized part indicates unification to be performed before commitment.

Finally, we justify the lack of AND- and OR-sequentiality in GHC. We deliberately excluded AND-sequentiality, because our programming experience with Concurrent Prolog has never called for this construct. One may think that AND-sequentiality could be used for the specification of scheduling and for synchronization. However, the primitives for scheduling should be introduced at a different level from that of GHC, and AND-sequentiality as a synchronization primitive is of no use in the computational model of GHC which allows anti-substitutability.

4.10. Possible Extensions—Treatment of Failure

In this section, we consider the treatment of finite failure in GHC.

The semantics of GHC as described in Section 4.4 does not introduce the concept of failure. Any meaningful result of computation is constructed only of successful subcomputations, though no such result may exist. Moreover, GHC does not commit itself to Closed World Assumption (Gallaire and Minker [1978]). We assume that a program is open-ended: If there is no selectable clause for a given goal, we regard it as not provided yet. This idea coincides with the idea of incomplete data structures, so it should be convenient when we handle programs as data. This idea coincides also with the idea of regarding system predicates as defined by possibly infinite program clauses, as we did in Sections 4.6.2 and 4.6.4. Note that this idea is effectively used in the query-the-user facilities (Sergot [1983]).

Nonetheless, it may sometimes be convenient to assume a closed world for a whole program or a specific predicate. Only by doing so we can introduce the predicate '`otherwise`' discussed in Section 3.4.7 or OR-sequentiality of PARLOG (Clark and Gregory [1984a]), both of which are used for expressing *default* clauses. The predicate '`otherwise`' can appear only as a guard goal. A goal '`otherwise`' succeeds when all the other program clauses turn out to be non-candidates (see below).

Finite failure can be recursively defined as follows:

(1) The goal of the form S=T fails

- if the principal function symbols of S and T are different, or
- if S is of the form $f(S_1, \ldots, S_n)$ and T is of the form $f(T_1, \ldots, T_n)$, where f is some function symbol, and the goal S_i=T_i fails for some i.

(2) The goal of the other form fails if all the clauses in the (closed) program turns out to be non-candidates.

Here, a clause

$$H \texttt{ :- } G_1\texttt{, } \ldots\texttt{, } G_m \texttt{ | } B_1\texttt{, } \ldots\texttt{, } B_n\texttt{.}$$

is said to be a non-candidate for a goal G if some of $G_1, \ldots, G_m$ or G=H fail.

Unfortunately, introduction of the notion of failure restricts the exploitation of sequentiality mentioned in Section 4.5. Let us consider the following example due to Gregory [1985a]:

```
Goal clause:                          :- and(X, false).
Program clause:  and(true, true) :- true | true.
```

The head unification fails if the arguments are unified in (pseudo-) parallel, but suspends forever if they are unified sequentially from left to right.

An even more awkward example due to Chikayama and Miyazaki [unpublished] is shown below:

Goal clause: `:- p(A, false, true).`
Program clause: `p(X, X, X) :- true | true.`

Let us assume that `A` and `X` have been unified. Then the head unification suspends at the second argument because if `X` were bound to '`false`', so would `A`. We go to the third argument if we employ pseudo-parallelism. However, the unification of the third argument suspends also for the same reason. This is an erroneous execution scheme, because `A`, '`false`' and '`true`' cannot be unified no matter what `A` is bound to. The ununifiability would be detected if the unification were done from right to left.

To generalize, when a variable X and ununifiable two terms T_1 and T_2 are unified in a guard, failure must be detected whether X can be instantiated or not. We need a mechanism capable of detecting the ununifiability of T_1 and T_2 without instantiating X.

4.11. Implementation of the Synchronization Mechanism

This section considers the implementation of the rules of suspension, or more specifically, the rule of synchronization. Of course, there are many other implementation issues to be considered, particularly on parallel implementation. However, parallel implementation would still have to be included in the future works because of their difficulty recognized by many researchers. As regards sequential implementation, Section 6.2 describes an efficient compiler, but that compiler does not allow calling user-defined predicates from a guard. So the purpose of this section is to show some ideas on the implementation of full GHC.

We will first show an easy-to-understand but possibly inefficient method: pointer coloring. When a term in a goal and a variable in the guard of a clause C are unified, we color the reference pointer which indicates the binding. We cannot make a binding to a term dereferenced using one or more colored pointers. When the clause C is selected, colored pointers created in its guard are uncolored. For this purpose, the guard of a clause must record all pointers colored for that guard. Uncoloring can be done in parallel with the other operations in the body, as we stated in Section 4.7.3.

Care must be taken when a term in a goal to be unified with a variable in a guard is itself dereferenced using colored pointers. Consider the following example:

Goal clause: `:- p(f(A)).` (4.11–1)
Program clauses: `p(X) :- q(X) | ... .` (4.11–2)
`q(Y) :- true | Y=f(b).` (4.11–3)

If `Y` should directly point to the term '`f(A)`' by a colored pointer and uncolor it upon selection of Clause (4.11–3), `A` would be erroneously instantiated to the constant '`b`'. There are a couple of possible remedies:

(1) To disallow pointers which go directly out of nested guards and to use instead a chain of pointers.

(2) To let each pointer know how many guards it goes up through.

(3) (Miyazaki [1985a]) Pointers may go directly through nested guards. However, we let each colored pointer know for what guard it is colored. When directly pointing to a term obtained by dereferencing colored pointers, the new pointer must be recorded in the guard which records the last colored pointer in the dereferencing chain.

In many cases, however, we can statically analyze suspension and need not use colored pointers. The simplest case is the following clause:

```
p(true) :- ... | ... .
```

The clause head claims that the argument of the caller must have been instantiated to '`true`' to select this clause. We can statically generate the code for this check, and need not use colored pointers.

In general, if a guard calls only system predicates for simple checking such as integer comparison, compile-time analysis is easy because no consideration is needed on other clauses. On the other hand, if it calls a user-defined predicate, global analysis is necessary to determine which unification may suspend and which unification cannot. There will be no general method for static analysis, but in many useful cases, static analysis like PARLOG's compile-time mode analysis (Clark and Gregory [1984c]) will be effective.

4.12. Summary and Future Works

We have proposed a parallel logic programming language Guarded Horn Clauses. Its design principles, syntax, semantics, important properties, programming examples, primitive operations, interpretation as a process description language, justification of the design, possible extensions, and implementation of the rule of synchronization have been described.

We will review the language design along the general design principles in Section 4.2. Firstly, we tried to rule out sequentiality from the semantics as much as possible. Even the rule of sequencing is described as part of the rules of suspension, which contributes to the uniformity. However, at the same time, we tried to retain the possibilities of exploiting sequentiality in an actual implementation. This is important for efficient implementation on a sequential computer or on communicating sequential computers. Secondly, we demonstrated in Section 4.6 that GHC is expressive enough to describe data-driven and demand-driven computation, bounded-buffer communication, and a meta-interpreter of GHC itself. In addition, we showed that GHC can be viewed as a general and powerful process description language in Section 4.8. Thirdly, we tried to keep the language as simple as possible by relating

all the semantical essences to the guard construct. We defined the semantics in terms of the original logic programming framework to keep the language independent of specific architectures and implementations and to avoid inessential details to creep in.

It cannot be immediately concluded that GHC can be efficiently implemented on parallel computers. The efficiency of GHC owes very much to the future research on the language itself and its implementation. However, we can say that GHC is more favorable than Concurrent Prolog for implementation, as will be discussed in Chapter 5. For applications in which efficiency is the primary issue but little flexibility is needed, we could design a restricted version of GHC which allows only a subclass of GHC and/or introduces declarations which help optimization. As stated in Section 4.2, such a variant should have the properties that additional constructs are used only for efficiency purposes and that a program in that variant is readable as a GHC program once the additional constructs are removed from the source program.

There are many future works on the language itself as well as its implementation, many of which are related to control issues and system programming. Those include notations for helping an implementation perform efficiently in time and space, facilities for writing programming systems such as tracers and debuggers, notations for exception handling, better treatment of the fairness of nondeterministic choice and choice with priority. These issues will be discussed in more detail in Section 8.3.3.

Chapter 5

COMPARISON OF GHC WITH OTHER PROGRAMMING LANGUAGES AND MODELS

In this chapter, we compare Guarded Horn Clauses with other programming languages and computational models.

5.1. Concurrent Prolog

GHC was designed in an effort to refine Concurrent Prolog keeping its programming paradigm as much as possible. As a result, these languages have many features in common from the viewpoint of programming—process interpretation, communication by streams, and committed-choice nondeterminism. If one turns his eyes to semantical details, however, he or she will find that GHC is simpler than Concurrent Prolog.

Firstly, unlike Concurrent Prolog, GHC has no read-only annotations. In GHC, the semantics of guards enables process synchronization. Thus the synchronization mechanism has become more static and easier to analyze at compile time. Nevertheless, we can use incomplete messages and specify demand-driven computation. The only technique abandoned is the 'protected data' technique explained in Section 3.4.3. GHC has excluded it because protected data make sense only if the language assumes sequentiality in the transmission of terms (see Section 3.4.4). However, GHC does not have such sequentiality because it has adopted anti-substitutability (Section 4.7.2), which states that a complex term may not be treated as an atomic entity when some variable is instantiated to it.

Secondly, Concurrent Prolog needs a multiple environment mechanism while GHC does not. In Concurrent Prolog, bindings generated in each guard are recorded locally until commitment and are exported into the global environment upon commitment (Miyazaki, Takeuchi and Chikayama, 1985). However, the language rules on this mechanism proved to contain semantical problems whose solution would require an additional set of language rules, as discussed in Section 3.4.3. More importantly, we have not had any evidence that we need multiple environments in stream-AND-parallel programming. The multiple environment mechanism of Concurrent Prolog may seem helpful in detecting the failure of output unification prior to commitment. However, this checking does not seem so important since we always write programs so that output unification succeeds. Moreover, this checking is incomplete unless we adopt the harder alternative on the semantics of commitment described in Section 3.4.3.1.

The third difference is that GHC enjoys anti-substitutability as mentioned above, while Concurrent Prolog does not because of its synchronization mechanism. Thus a complex term is not treated as atomic in GHC, and communication by a shared variable may have potential delay (see Section 4.7.2).

5.2. PARLOG

GHC has many similarities to PARLOG as well as to Concurrent Prolog from the viewpoint of programming. However, unlike PARLOG, GHC requires no mode declaration for each predicate. Mode declaration of PARLOG is nothing but a guide for translating a PARLOG program into Kernel PARLOG (Clark and Gregory, 1984c). It is a kind of macro and we can do without modes. In fact, GHC is more similar to Kernel PARLOG than to PARLOG. However, unlike Kernel PARLOG, we have only one kind of unification. Although each unification operation appearing in a GHC program might be compiled into one of several specialized unification procedures, GHC itself needs—and has—only one.

Another difference from (Kernel) PARLOG is that a (Kernel) PARLOG program requires compile-time analysis in order to guarantee that it is legal, i.e., it contains no unsafe guard which may bind variables in the caller of the guard (Clark and Gregory, 1984c). Although Gregory (1985b) states that the safety check of guard could be done at run time, it should not be the intended use of PARLOG. On the other hand, a GHC program is legal if and only if it is syntactically legal; it can be executed without any semantic analysis. This is the main source of semantical simplicity of GHC compared with PARLOG. However, we must note that a GHC program can be executed *more efficiently* with compile-time analysis. It is true that some GHC programs require nontrivial run-time analysis of whether or not each piece of unification generates a disallowed binding, but we expect that such programs are seldom written. It is expected that the class of GHC programs whose PARLOG counterparts have no unsafe guard accepts compile-time analysis. For example, the GHC-to-Prolog compiler described in Section 6.2 disallows user-defined goals in guards, but it generates the code for suspension checking as a sequence of Prolog goals.

From a semantical point of view, GHC is much nearer to PARLOG than to Concurrent Prolog. As a result, many of the implementation techniques of PARLOG can be used for GHC and vice versa. Kernel PARLOG could be used as an intermediate language of a GHC compiler. Output unification specified in a clause body must be implemented by full unification in principle, but this unification could be made more efficient by compile-time analysis. In a word, PARLOG can be viewed as a realistic approximation to, or a specialization of, Guarded Horn Clauses.

PARLOG has a couple of features not in GHC. One is the notation of set constructors for eager and lazy exhaustive search for solutions of Horn-clause programs. It is possible to incorporate a similar notation into a user language of GHC. A program in the user language can be compiled into GHC by using the technique described in Chapter 7. Other features include sequential control and some extralogical predicates such as '`var`'. These features are out of the scope of the definition of GHC. Features for mere control which do not affect the result of computation could be introduced in a control metalanguage. Other extralogical features are yet to be studied.

5.3. Qute

Qute (Sato and Sakurai, 1984) is a functional language based on unification. Qute allows parallel evaluation which corresponds to AND-parallelism in logic programming languages, but the result of evaluation is guaranteed to be the same irrespective of the particular order of evaluation. There is no committed-choice nondeterminism.

Although Qute and GHC were independently developed and look differently at a glance, their suspension mechanisms, introduced with different motivations, are essentially the same. The Qute counterpart of GHC's guard is the condition part of the *if-then-else* construct, from where no bindings can be exported. Both the guard and the condition part can be used for conditional branching and synchronization. An important difference is that Qute uses the notion of failure as well as the notion of success for conditional branching, while GHC uses the latter only. As we discussed in Section 4.10, failure is not always easy to implement correctly at least on a sequential machine, though those difficult cases may seem rather pathological.

Another difference related to the above one is that Qute has no committed-choice nondeterminism while GHC has one. Qute does not have committed-choice nondeterminism (though Sato and Sakurai (1984) suggest it could) because it pursues the Church-Rosser property of the evaluation algorithm. GHC has one because our applications include a system which interfaces with the real world (e.g., peripheral devices).

GHC provides a simpler and disciplined user interface than Qute in the following point. Qute has two ways of returning the result of computation: One is by returning it as the value of an evaluable expression, and the other is by returning it through an argument variable of the expression. GHC has the latter only. It could be argued that Qute is a stronger language than GHC in the senses that it has the notion of failure and that it allows functional abstraction, i.e., it allows us to dynamically construct a function, pass it around, and apply it. However, it is yet to be studied where these features are essential in writing a program.

5.4. Oc

Oc recently proposed by Hirata (1985) is thought of as a programming language which simplified GHC further. Oc has thrown away any guard goals; thus a guard always consists only of a clause head. One may wonder how, say, comparison of integers can be specified. However, all system predicates have been changed to return a constant indicating success or failure, so we can do conditional branching according to that constant. Disregarding the above issue of system predicates, Oc is a strict subset of GHC.

5.5. Communicating Sequential Processes (CSP)

We compare GHC with CSP (Communicating Sequential Processes) as de-

scribed in (Hoare, 1978). We did not choose CSP described in (Hoare, 1985); it is more of a computational model while CSP in (Hoare, 1978) is a proposal of a programming language.

GHC is similar to CSP in the following points:

(1) Both encourage programming based on the concept of communicating processes.

(2) Input and output are regarded as fundamental in the design of a language.

(3) The guard mechanism plays an important role for conditional branching, nondeterminism and synchronization.

(4) Both pursue simplicity.

The major difference is that CSP tries to rule out any dynamic constructs—dynamic process creation, dynamic memory allocation, recursive call, etc.—while GHC does not. Another major difference is that CSP has a concept of sequential processes while GHC does not. A GHC program can be read as specifying communicating *parallel* processes. These differences come from the difference in design philosophy: The design of CSP is affected by the current computer architecture, while GHC is designed keeping independence of, but never ignoring, implementation on existing computers.

As a result, GHC is more abstract and has a smaller set of primitives: It uses unification instead of input, output, and assignment commands, and it uses recursive call instead of a repetitive command. Both use the guard mechanism, but the guard of GHC has a capability of synchronization while CSP provides the synchronization primitives separately. Finally, CSP and GHC are different in the manner of communication: CSP employs synchronized communication while GHC employs buffered communication. However, this is not an essential difference, since each communication scheme can simulate the other.

5.6. Sequential Prolog

Comparison with sequential Prolog must be made from the viewpoint of logic programming languages rather than the viewpoint of parallel programming languages.

First of all, GHC has no concepts of the order of program clauses or the order of goals in a clause. GHC is undoubtedly nearer to Horn-clause logic in this point. The semantics of Prolog must explain its sequentiality; without it, we cannot discuss some properties of a program such as termination.

GHC has lost completeness as a theorem prover of Horn-clause logic deliberately, not as a result of compromise (see Section 4.9). On the other hand, Prolog

has lost completeness rather unconsciously because of its depth-first search strategy. It will be hard to straightforwardly describe the semantics of GHC within the framework of first-order logic. This situation, however, is common also to Prolog because of the notorious but important cut operator as well as the search strategy. The commitment operator of GHC is the parallel of the cut operator, but it must be a simpler construct from a formal point of view, since it has been introduced in a more disciplined way.

Prolog features backtracking, which is a sequential implementation of the multiple environment mechanism. On the other hand, GHC is a single-environment language. This means that an implementation of GHC needs no 'trail stack' used for unbinding and the operations on it.

One problem with Prolog is that the use of '`read`' and '`write`' predicates prevents the declarative reading of a program (see Section 2.3.3). In GHC, we no longer need imperative predicates because the concept of streams can be well adapted to input and output. Large data structures such as mutable arrays and databases can also be declaratively and efficiently handled by using transaction streams as the interface, as we will discuss in Section 6.1.

5.7. Delta-Prolog

Delta-Prolog (Pereira and Nasr, 1984) is an extension of Prolog which allows multiple processes. Communication and synchronization are realized using the notion of an *event*. The underlying logic which explains the meaning of events is called Distributed Logic, which is a kind of modal logic.

One of the differences between Delta-Prolog and GHC is that Delta-Prolog retains the concept of sequentiality and the cut operator of Prolog. Sequentiality is essential in Delta-Prolog because it is utilized for guaranteeing the order of events. GHC did not stick to those features of Prolog since they seemed to be no more than peculiarities of Prolog. A parallel program in Delta-Prolog may look quite different from the comparable sequential programs in Delta-Prolog itself and in Prolog. On the other hand, a class of GHC programs which have only unidirectional information flow (like pipelining) can easily be rewritten in Prolog by replacing commitment operators by cuts, and a class of Prolog programs which use no deep backtracking and each of whose predicates has only one intended input/output mode can be easily rewritten in GHC also.

5.8. Other Computational Models

In this section, we compare the paradigms provided by GHC with those provided by other computational models. Shapiro (1983a) also made detailed comparison between Concurrent Prolog and other models.

5.8.1. Dataflow Computation

As we mentioned in Section 4.8, GHC can be viewed as a flexible process description language and as a generalization of dataflow languages. Here we compare GHC in detail with the conventional dataflow computational model based on dataflow graphs, or more specifically, with the model described in (Amamiya, 1984).

GHC can be called a dataflow language in that computation proceeds based on the availability of data. We can translate a dataflow graph into a GHC program straightforwardly, as long as the dataflow graph does not use awkward notions as mentioned later. A node in the dataflow model is a kind of a process, and its function can be realized by a GHC goal. An arc carries data and can be represented by a variable possibly shared between goals. Note that such a variable may not necessarily represent a stream of data: A stream must be used only when it is necessary to process a sequence of data. Stream merging is not considered as primitive in GHC but can be defined in the language.

However, GHC can be viewed as providing a more general and flexible dataflow model than the conventional dataflow model. A major difference between the conventional dataflow model and GHC is that the former does poor distinction between a computational model and implementation issues, and between static program structures and run-time process structures. For example, the idea of sharing nodes 'for the sake of efficiency' called for the notion of colored tokens, which seems to be too concrete a notion to appear in a computational model. GHC, on the other hand, well separates source programs and run-time process structures, and naturally expresses dynamic process reconfiguration. Nevertheless, it allows efficient implementation: We use recursion in defining iterative processes, but recursive creation of almost the same process is very cheap, as we will show in Section 6.1.

A dataflow graph is too restrictive for expressing flexible data and control structures. In GHC, demand-driven computation, communication using incomplete messages, 'lenientcons', and mutable objects can be specified elegantly in a uniform framework, while their dataflow graph representations tend to be awkward. To sum up, a dataflow graph seems to be appropriate only for expressing a simple class of dataflow computation.

5.8.2. Actor Model

The actor model (Hewitt, 1977; Yonezawa, 1979; Yonezawa, 1984) is a computational model based on objects and message passing. It is considered as giving the foundation of object-oriented programming.

The process interpretation of GHC shows major similarity to and minor difference from the actor model. Roughly speaking, an object (called an actor) in the actor model corresponds to a goal in GHC, and message passing and receiving are done by unification in a body and a guard, respectively. A GHC goal can represent

either *pure* or *impure* actor; that is, it can effectively hold its internal state. A goal representing an object 'identifies' another goal by a shared variable (usually used as a stream) leading to it rather than by its name. Like the actor model, a goal can directly send a message only to its direct *acquaintances* (Hewitt and Baker, 1977), that is, it can make bindings only to those variables to which it has access. Both models are completely distributed models. There are no global or central notions; computation is subject only to the local causality specified by program clauses. Lastly, both models assume potential delay in message passing.

However, there also exist several differences. First we mention the differences in the concept of an object. In GHC, the notion of state change is nothing but a pragmatic one, and at the language level it is described without side effects. That is, an object with internal states can be implemented as a goal with 'state arguments' which hold the current set of state values, and the state change is done by reducing itself to a new goal with different state arguments. Another difference is that not all objects may be implemented as goals, but those only denoting values such as individual integers may be implemented as terms. This is a kind of optimization: We can alternatively implement individual integers as goals which accept messages such as 'plus' and 'factorial', though it does not seem to be a natural programming style. Two more differences follow. While an actor is activated only by receiving a message, a GHC goal is always active and receives a message by itself by unification in guards. Moreover, there is no one-to-one relationship between arrival of a message and reduction of a goal. A GHC goal may be reduced to other goals without receiving any messages, or it may be reduced by receiving two messages from the same or different goals.

Next, we mention the differences in message passing. The most important difference is that many-to-one communication in GHC must be specified explicitly by means of choice nondeterminism in the commitment operation, while the merging of messages is implicit in the actor model. Assume, for example, that a goal is to directly send (by stream communication) a message to another goal which is not a direct acquaintance. Then it must ask some goal to set up a new stream and make it *merged* with existing streams leading to the destination goal. Here, the merge predicate provides the necessary choice nondeterminism. Note that explicit specification of choice nondeterminism does not necessarily cause inconvenience in programming or inefficiency of implementation if we have an appropriate user language and an optimizing compiler; see Section 6.1 for an optimization technique for the merge predicate.

The actor model assumes that there is a total order in the arrivals of messages at some object (Hewitt and Baker, 1977). On the other hand, GHC has no corresponding notion; we assume no order between the instantiation of two arguments of a goal, which is physically a natural decision. As Clinger (1981) suggests, arrival ordering is a strong assumption which needs some form of arbitration to make it realistic. Thanks to the arbitration implicit in the formalism, the actor model can

discuss fairness, while the current computational model provided by GHC is too liberal to handle fairness. Finally, we note that two messages on the same stream are ordered even in GHC, of course.

5.8.3. Process Network Model by Kahn

Kahn (1974) presented a simple language to express networks of parallel processes, which is conceptually very similar to GHC with the process interpretation. Although procedural, his language can be counted as one of the ancestors of GHC. The language is not so static as CSP: It allows recursive call and it employs buffered communication. A channel used for one-to-one communication is almost exactly the counterpart of a stream of GHC, except that communication is not performed by unification but by the imperative commands. Kahn (1974) did not explicitly state that a network allows dynamic reconfiguration, but he and MacQueen (1977) showed a prime generator program similar to the one in Section 4.6.2 using a slightly modified language.

A small but important difference between his language and GHC is that the former has excluded choice nondeterminism.

5.8.4. Functional/Applicative Programming

There are many proposals of functional or applicative languages including FP (Backus, 1978), HOPE (Burstall, MacQueen and Sannella, 1980), KRC (Turner, 1981), FEL (Keller, 1982) and Valid (Amamiya, Hasegawa and Mikami, 1983), and some of them allow for concurrent evaluation. We compare GHC with those languages as a whole.

GHC is similar to those functional languages in that it is a *declarative* language satisfying a form of referential transparency which we called anti-substitutability in Section 4.7.2. All these languages are free from side effects. Another point is that many of those functional languages as well as GHC allow for demand-driven computation and non-strict data structures. The major difference of GHC from those functional languages is in how elegantly these features are introduced. As we showed in Section 4.6.3, demand-driven computation can be elegantly specified in GHC without any additional computation rules by using the incomplete message technique. Furthermore, the data structure of GHC naturally includes incomplete or non-strict terms, while in usual functional languages non-strict data structures must be realized by means of lazy evaluation (call by need). Thus it can be said that GHC has simpler and more refined concepts both on data structures and computation rules, the latter of which owes much to parallelism.

Moreover, GHC can express quite complex information flow by using the incomplete message technique while the functional languages can express only simple information flow. This is why GHC can be used for describing parallel systems

based on message passing. Note that the above comparison does not apply to the 'functional' language Qute (Section 5.3); however, it is questionable whether Qute can be called a functional language in the usual sense because the evaluation of a Qute expression may instantiate variables in the expression.

Many functional programming languages have type systems while GHC does not. However, a user language of GHC could have a type system to make programs more readable and reliable by providing programming systems and compilers with more information (Bruynooghe, 1982).

A point which is advantageous to functional languages over logic programming languages including GHC is the capability of handling higher-order entities. The higher-order features provided by the current logic programming languages are less expressive and less elegant.

5.8.5. Object-Oriented Languages

Since we have already discussed the computational aspects of object-oriented languages in Section 5.8.2, here we deal with linguistic aspects. The most important linguistic aspect common to most of the object-oriented languages is the inheritance mechanism (Goldberg, 1983; Weinreb, 1981; Chikayama, 1984), though it is independent of the message-passing aspect and hence is not a prerequisite for being object-oriented. Nonetheless, the inheritance mechanism is important because it enables differential programming, a programming methodology for obtaining a desired program by specifying the differences from the existing ones.

Shapiro and Takeuchi (1983) showed how to implement inheritance in Concurrent Prolog, and the same technique applies also to GHC. In their method, each object (i.e., goal) delegates the processing of a message it has received to its parent object if it cannot process the message by itself. Thus the hierarchy of objects is implemented by cascading them by streams. This is a good solution since relational programming provides a suitable framework for differential programming as we discussed in Section 3.3.2.

A problem in this solution is that the message delegation occurs dynamically whereas the hierarchical structure is usually static. It is necessary to consider a technique for efficient implementation. A process fusion technique (Furukawa and Ueda, 1985) will help efficient implementation and thus will encourage differential programming and modularization. Moreover, an appropriate syntactical support for hierarchical programming may be useful in a user language of GHC. Note, however, that such a syntactic support does not affect the underlying computational model which we believe should be as simple as possible.

Chapter 6

IMPLEMENTATION OF PARALLEL LOGIC PROGRAMMING LANGUAGES

This chapter deals with two topics on the implementation of Guarded Horn Clauses and other parallel logic programming languages. One is an efficient implementation of many-to-one and one-to-many stream communication among processes; the other is an efficient compiler of Guarded Horn Clauses and Concurrent Prolog on top of sequential Prolog. The technique for the former can be applied also to the implementation of arrays which allow constant-time access.

This chapter focuses on implementation on conventional sequential computers; implementation on parallel computers is not discussed here. Of course, to demonstrate the viability of parallel logic programming languages on parallel computers, we cannot limit the scope of discussion to sequential computers. Nevertheless, implementation on a sequential computer is very important.

It is most likely that our first step towards a general-purpose parallel computer is communicating sequential computers. Then, even on a parallel architecture, it is very likely for each *processor* to deal with multiple *processes* for the following two reasons. First, it is unrealistic to limit the number of processes a user can create to the number of processors available. Second, even if a lot of processors are available, the best way to allocate two processes which communicate intensively with each other and have little portions executable in parallel will be to allocate them on the same processor. Therefore, the technique of running multiple processes efficiently on a single processor is crucial for the efficiency of the whole system.

Moreover, parallelism in a source language and parallelism in implementation can be different as we argued in Section 3.1. If for some problem we can write a better program in a parallel language than in a sequential language, then it is meaningful to have an efficient implementation of the parallel language on a sequential computer.

6.1. Stream and Array Processing

In this section, the GHC predicate for merging n input streams is investigated, and a compilation technique for getting its efficient code is presented. Using the technique, data on input streams are transferred with a delay independent of n. Furthermore, it is shown that the average time for the addition and the removal of an input stream is independent of n. The predicate for distributing each item on an input stream to one of n output streams can also be realized as efficiently as n-ary *merge*. The compilation technique for the *distribute* predicate is applicable also to the implementation of mutable arrays that allow constant-time access and updating. Although the efficiency stated above could be achieved by a sophisticated compiler,

the codes should be provided directly by the system to get rid of the bulk of source programs and the time required to compile them. The implementation technique described in this section was first proposed in (Ueda and Chikayama [1984a]) using Concurrent Prolog as a base language. By altering the base language to GHC, the technique described below has been slightly simplified.

6.1.1. Introduction

When we implement a large-scale distributed system in parallel logic programming languages such as Concurrent Prolog, PARLOG and GHC, the performance of the system will be influenced significantly by how efficiently streams as interprocess communication channels can be merged and distributed. So we discuss how to implement predicates for merging many input streams and those for distributing data on a single input stream into many output streams. We use GHC for the following discussions, but the results obtained are in principle applicable also to Concurrent Prolog and PARLOG.

6.1.1.1. Importance of Streams in GHC-like Languages

As we stated in Section 4.8, GHC allows process interpretation: Processes are expressed by goals which are executed in AND-parallel, and interprocess communication is expressed by means of shared variables appearing as arguments of the goals. Statically, the shared variables express lists of data or messages flowing among (usually two) goals: As computation proceeds, the values of the lists are gradually instantiated to the end. We use the term 'stream' to refer to shared variables used in this manner (Section 3.3.2).

Since interprocess communication is done by instantiating and checking streams which have been laid among processes in advance, the efficiency of stream operations—sending, receiving, merging, and distributing—are of crucial importance.

6.1.1.2. Necessity of Dynamic, Multiway Stream Merging and Distribution

Streams need not be merged or distributed if several processes are linearly connected by shared variables to perform pipeline processing. However, if there is a process that needs to receive data or messages from many other processes—e.g., a process that manages a shared resource—a merging process must be put at the front-end:

$$:\!- p_1(C_1), p_2(C_2), \ldots, p_n(C_n),$$
$$\quad \texttt{merge}(\texttt{C}, C_1, C_2, \ldots, C_n), \mathit{shared\_resource}(\texttt{C}).$$

In order to accept messages from an indefinite number of processes, it must further be possible to dynamically vary the number of input streams to be merged.

Suppose that a newly created process needs to communicate with a shared process. Then it must first issue a request to the front-end merging process (by using other input streams or a 'request' stream) to set up a new input stream. A new stream could alternatively be laid by attaching binary *merge* to one of the existing input streams. However, communication delay will be proportional to the number of communicating processes if this method is repeatedly used.

The purposes of message distribution can be classified into *broadcasting* and *routing*. Broadcasting is easy at least at the source language level: Receiver processes need only share the broadcast stream. On the other hand, routing is somewhat complex. Routing is necessary to implement *public* communication channels for *private* communication. It may often happen that a process, say A, wants to communicate with another process B to which no direct communication channel has been laid. In such a case, there are two possible ways. One is to send a message through an indirect and 'public' path; the other is to send a request message (through an indirect path) to establish a direct and private communication channel and to use it afterwards. In either case, the process A must have indirect access to the process B; the relay processes on the way must be able to appropriately transfer messages according to the destinations attached to the messages.

In message distribution also, it must be possible to dynamically change the number of processes to be managed.

6.1.1.3. Related Works

Shapiro and Mierowsky [1984] dealt with the problem of merging an indefinite number of streams (henceforth the number of input streams will be denoted by n). They demonstrated

(1) a method to ensure n-bounded waiting and the maximum delay of $O(n)$ by using an unbalanced tree consisting of binary *merge*, and

(2) a method to ensure n-bounded waiting and a maximum delay of $O(\log n)$ by using a 2-3 tree (Aho, Hopcroft and Ullman [1974]) consisting of binary and ternary *merge*.

The term 'n-bounded waiting' was defined by them to mean that any message arriving at the merging process will be overtaken by no more than n input messages from other streams.

The delay of $O(n)$ in Method (1) will be unacceptable when n is large and the traffic is heavy. This method may be practical, however, in the case of essentially costly communication such as interprocess communication in multi-processor environments.

Method (2) is a major improvement over Method (1) in terms of delay. In procedural languages, however, the delay of interprocess communication does not

depend on the number of senders as long as it is simulated on a sequential computer. Therefore, it is desirable to achieve a constant-time delay also in logic programming languages.

Kusalik [1984] also dealt with bounded-wait merging of n streams. He showed a method to ensure bounded-wait merging without resort to the operational characteristics of the underlying machine or interpreter but to the extralogical primitive '`varbl`'. One of his solutions has $O(\log n)$ delay, but the number of input streams cannot be changed. The other solutions can merge indefinite number of streams, but they are inefficient.

The above two papers concentrate on how to *program* n-ary *merge* having the desired properties. On the other hand, we study how to *compile* a naive n-ary *merge* program.

After our method was published, Shapiro and Safra [1985] showed that multiway merge can be programmed in Concurrent Prolog by introducing destructive assignment. The difference between their and our approaches is that we did not introduce any extralogical primitives into the base language. They claim that their approach can realize smaller delay, but there should be no essential difference in delay. If our approach is specialized, housekeeping data and the code for manipulating them can be simplified and made more efficient.

Gelernter [1984] discussed the suitability of Concurrent Prolog for the description of multi-process systems. He concludes that interprocess communication using merge networks is 'not only bulky but unduly constricting'. However, this criticism is not from the viewpoint of descriptive power or efficiency.

6.1.2. Objectives

We have the following two objectives:

(1) To realize on a sequential computer n-ary *merge* and *distribute* with a maximum delay of $O(1)$.

(2) To extend the solution to (1) to the case where n varies dynamically.

It is clear that (1) cannot be accomplished through the combination of binary and ternary *merge* or *distribute*. The predicates must process all messages directly at the top level:

```
merge(Ys,X1, ...,[X|Xk],..., Xn) :-  true | Ys=[X|Zs],
merge(Zs,X1, ...,  Xk,   ..., Xn).

distribute([(k,X)|Xs],Y1, ...,Yk,..., Yn)  :- true | Yk=[X|Zk],
distribute(    Xs,     Y1, ...,Zk,..., Yn).
```

If these predicates are *interpreted*, the cost for tail recursion can be proportional to the size of each clause (= $O(n)$). However, if *compiled*, these predicates promise

to yield higher efficiency, as will be discussed in Section 6.1.3.1. We should no longer define 'delay' as the depth of a tree. We will define 'delay' as

- the time passed from the arrival of a message at a goal in a waiting state until the original waiting state is restored by tail-recursion, during which the message is transferred to the output stream.

The delay is calculated by the number of primitive operations which are regarded as executable within a unit time on a sequential computer in usual complexity analysis. Speaking more precisely, we adopt the RAM model under the uniform cost criterion (Aho, Hopcroft and Ullman [1974]).

6.1.2.1. Outline of Sequential Implementation of GHC

We assume that our implementation adopts the following process management technique proposed by Shapiro [1983b].

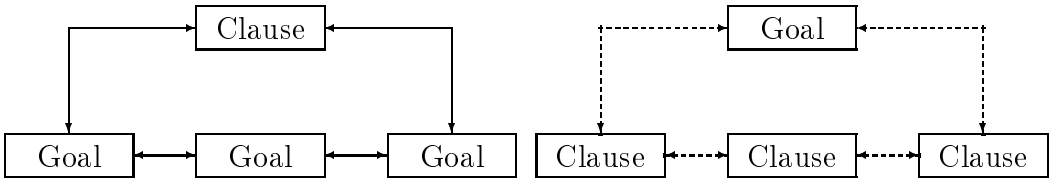
The descriptors of conjunctive goals make up a circular list called an AND-loop, and the descriptors of candidate clauses composing a predicate make up a circular list called an OR-loop (Figure 6.1).

Each element goal of an AND-loop is the parent of an OR-loop comprising candidate clauses until it is committed to one of those clauses; after commitment, it is replaced by a doubly-linked list representing goals of the body (Figure 6.3(a)). If the body is empty, the element of the original AND-loop disappears (Figure 6.3(b)). The parent of an AND-loop, having lost all elements, is considered a success (Figure 6.3(d)). On the other hand, failure of any AND-loop element is the failure of the parent (Figure 6.3(e)), assuming that we adopt Closed World Assumption.

Each element of an OR-loop represents a candidate clause which has not yet been selected for commitment, and is the parent of the AND-loop whose elements represent guard goals. The success of an OR-loop element implies that the parent goal can be committed to the clause corresponding to that element (Figure 6.3 (a), (b)). On the contrary, when some element of an OR-loop fails, that element simply disappears. The parent goal of an OR-loop, having lost all elements, is considered a failure (Figure 6.3(c)).

The system has a queue called *Process Queue* in which leaf elements of a tree formed by AND/OR-loops (i.e., elements which are not parents of other loops; see Figure 6.2) await execution. A clause suspended due to the dataflow restriction waits in the waiting list attached to the variable that caused suspension instead of waiting in *Process Queue*. That clause is re-scheduled when the variable is instantiated.

One possible optimization of the above method is to execute head unification and simple guard goals as an indivisible sequence of operations. We call it *immediate check*. If an immediate check succeeds, we need not create an OR-loop. Otherwise,



(a) AND-loop (b) OR-loop

Fig. 6.1. AND-loop and OR-loop

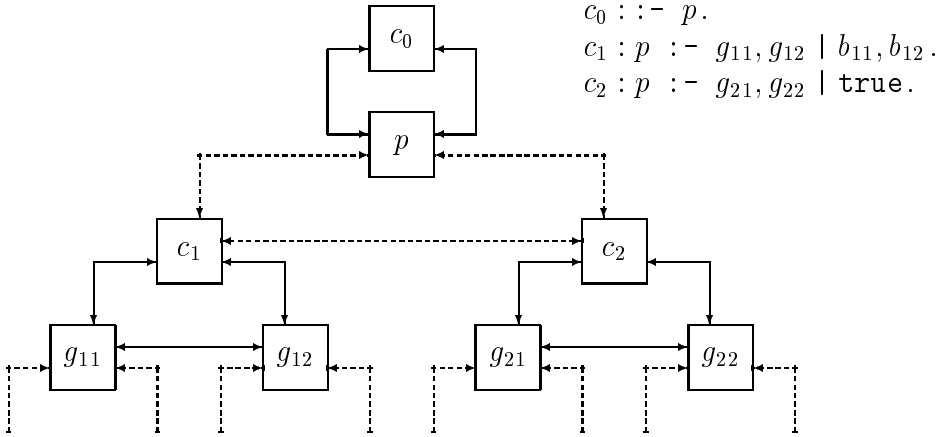


Fig. 6.2. Tree Structure Constructed by AND/OR-loops

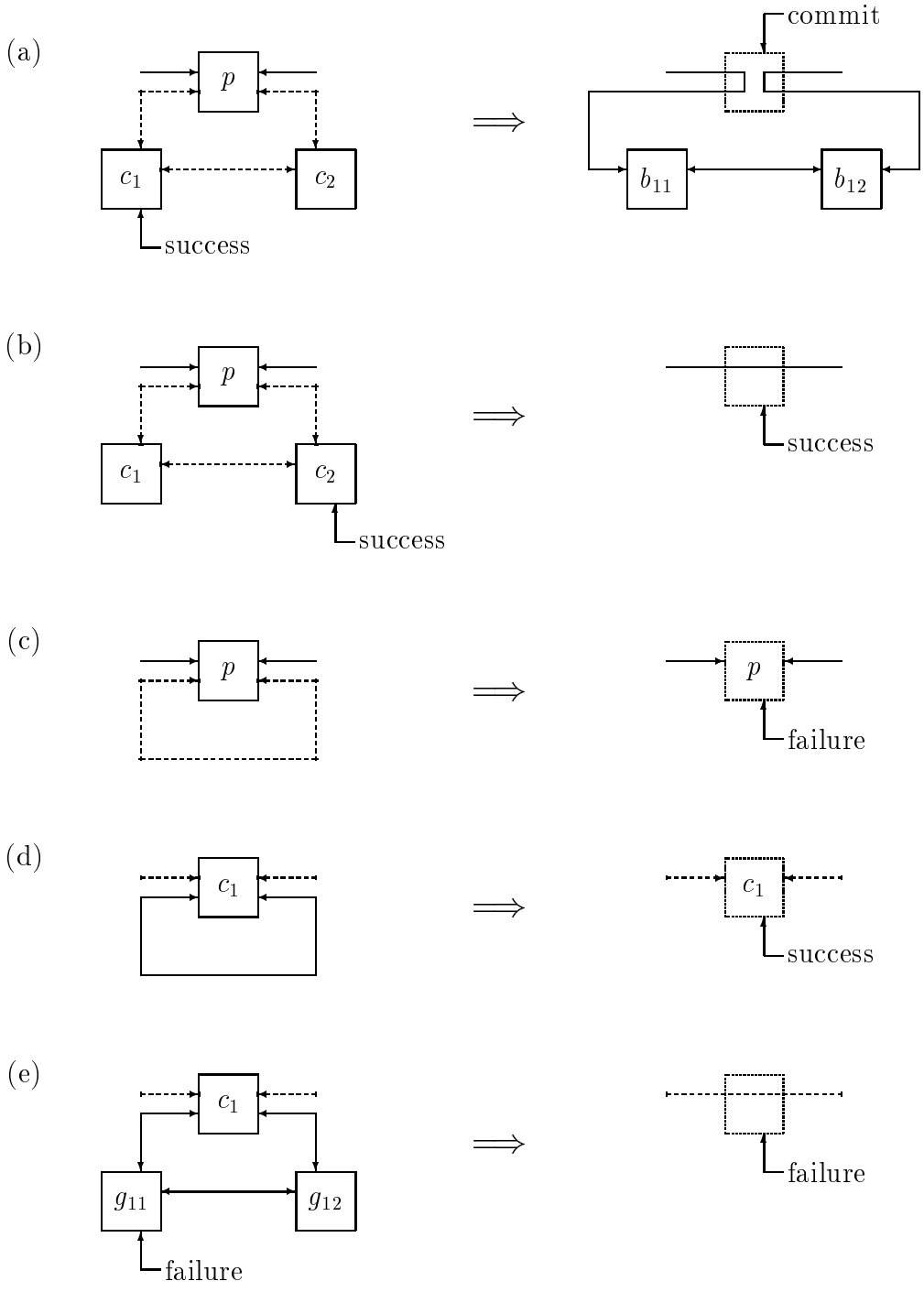


Fig. 6.3. Changes and Their Propagation of AND/OR Loops

an OR-loop is created for those clauses which have suspended during immediate check and which have succeeded in immediate check but have complex guards, and they wait for instantiation of variables.

6.1.3. Implementation of the Merge Predicate

6.1.3.1. Examination of *n*-ary *merge*

The n-ary *merge* predicate is defined by n clauses of the following form if we ignore the base cases for termination which will be discussed in Section 6.1.3.3.5:

• The kth clause ($k = 1, \ldots, n$):

```
merge(Ys,X1, ...,[X|Xk],..., Xn) :-  true | Ys=[X|Zs],
merge(Zs,X1, ...,  Xk,   ..., Xn).
```

This predicate has the following characteristics:

(1) To see if the cth clause is selectable, one need only examine the cth argument (henceforth we number the arguments starting with 0).

(2) Upon the tail recursion employing the cth clause, only the 0th and the cth arguments change compared with the original goal. Therefore, the argument list of the tail-recursive goal can be obtained by slightly modifying that of the original goal.

(3) When all clauses are in a wait state and one of the argument variables is instantiated, only one clause (or two, including the base case) needs to be re-examined.

Now we will consider tail recursion. Suppose that a goal G selects a clause C and a recursive body goal G' is generated whose kth argument is the same as that of G. Then the wait condition of each program clause with respect to G' is the same as that with respect to G as regards the kth argument; that is, if the unification of G with the head of some clause suspends at the kth argument, then it should suspend also for G' unless other body goals in the clause C instantiate the kth argument of G.

Stating the above property in terms of n-ary *merge*, we get the following.

(4) After tail recursion employing the cth clause, we have to examine the following clauses until some selectable clause is found or all clauses to be examined have been examined:

  (a) the cth clause itself,

  (b) clauses which must have been but have not been examined in the previous call due to the selection of some other clause, and

(c) clauses which must be examined again as the result of the unification invoked in the body of the cth clause.

Possibility (c) is not directly related to tail recursion, so we do not count it as the cost of tail recursion. Possibility (b) refers to the clauses that have been 'carried over': Once they are examined, they will suspend or become non-candidates due to failure of unification, or they will be selected and again become candidates after tail recursion. Therefore, the average number of clauses to be checked after each tail recursion does not depend on the total number of clauses.

From the above considerations, we can conjecture that n-ary *merge* can process each message within a constant time. Note that Warren [1980] proposed an implementation technique of sequential Prolog that takes advantage of the characteristics (1) and (2).

6.1.3.2. Implementation Technique for Fixed-Arity *merge*

To efficiently implement n-ary *merge*, we have to consider the following:

(1) OR-loop (with $O(n)$ elements) must not be created and discarded upon each recursion even if all clauses suspend. In order to prevent examination of clauses not worth examining, it is best to manage candidate clauses within the descriptor of a goal.

(2) The argument list must be re-utilized.

In the following sections, we show a general compilation technique which achieves a constant-time delay when applied to n-ary *merge*. The technique is applicable to other predicates as long as they have no user-defined goals in their guards. However, while our technique efficiently processes multiple waits of a goal, most programs we write do not wait for the values of two or more variables simultaneously; for such programs a simpler compilation technique could be used.

We adopt Closed World Assumption throughout Section 6.1. Hereafter, the number of clauses composing the predicate will be denoted by M, and the number of arguments by N.

6.1.3.2.1. Configuration of a Process Descriptor

A process descriptor, or the descriptor of a goal, has the following items.

(1) *AND Brothers:* Two pointers for constructing an AND-loop.

(2) *Process Queue Pointer:* A pointer for designating the next element in a *Process Queue*.

(3) *Candidate Queue:* A queue of candidate clauses of the goal managed by the process descriptor. At most M elements.

(4) *Clause States:* An array indicating whether each clause is in the *candidate*, *suspend*, or *fail* state. M elements.

(5) *Clause Backward Pointers:* An array of pointers designating entries in the waiting lists attached to uninstantiated variables that suspended candidate clauses. One element for each clause. Each pointer is meaningful if and only if the corresponding *Clause State* is *suspend*.

(6) *Suspend/Fail Table:* The reasons why a particular clause was not selected for commitment can be attributed to some of the arguments of the goal. Thus, if these arguments change upon tail recursion, that clause may become selectable. Therefore, a table of pairs $(c,\ k)$, where c is the number of the suspended or failing clause and k is the number of the argument that may be the cause, is maintained. This table must enable

- efficient sequential retrieval of elements containing c, and
- efficient deletion of elements containing k.

For example, the structure shown in Figure 6.4 fulfills this condition. The maximum number of elements depends on the predicate; in the case of n-ary *merge*, it is $O(N + M)$ and hence $O(n)$.

(7) *Fail Count:* The total number of clauses that cannot be selected for the current goal due to failure of unification.

(8) *Program Code:* A pointer to the predicate's code.

(9) *Argument List:* Arguments of the goal. N elements.

6.1.3.2.2. Operations

A. Creation of a Process Descriptor

When some predicate is newly called (i.e., not as tail recursion), the area for the process descriptor is allocated and its entries are set up as follows:

- all clauses are entered in *Candidate Queue* (3),
- all *Clause States* (4) are set to *candidate*,
- all *Clause Backward Pointers* (5) are left undefined,
- *Suspend/Fail Table* (6) is cleared,
- *Fail Count* (7) is set to 0, and

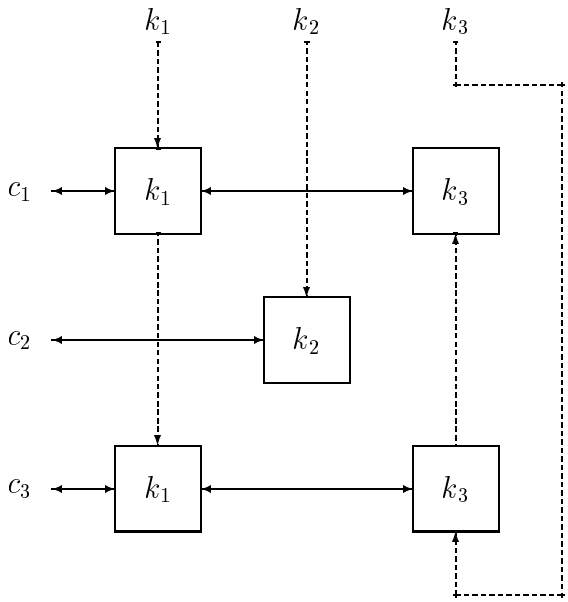


Fig. 6.4. Sample Data Structure of Suspend/Fail Table

- *Program Code* (8) and *Argument List* (9) are set up.

The completed process descriptor is entered in the AND-loop by appropriately modifying *AND Brothers* (1) of this and neighboring goals. It is also entered in *Process Queue* by making it designated by *Process Queue Pointer* (2) of the last element.

B. Selection of a Clause

B-1. If *Candidate Queue* is not empty, instructions for unifying the arguments of the first candidate (say the cth clause) and those of the goal (*Argument List* of the process descriptor) are executed. In the case of n-ary $merge$, only the instructions for the cth argument are executed.

- If this succeeds, the body goals are executed (see D).
- If this fails,
  (1) *Fail Count* is incremented by 1,
  (2) *Clause State* of the cth clause is set to $fail$,
  (3) *Suspend/Fail Table* is updated (cf. Section 6.1.3.2.3), and
  (4) other candidate clauses in *Candidate Queue* are tested.
- If this suspends,
  (1) *Clause State* of the cth clause is set to *suspend*,
  (2) *Suspend/Fail Table* is updated,
  (3) the pair $(p,\ c)$, where p is the pointer to the process descriptor and c is the number of the clause, is entered in the waiting list of the variable that appears in the goal and caused the suspension,
  (4) *Clause Backward Pointer* for the cth clause is made to point to the pair entered in (3), and
  (5) other candidate clauses in *Candidate Queue* are tested.

B-2. If *Candidate Queue* is empty and *Fail Count* is equal to M(= the number of clauses), the goal ends with failure since we have adopted Closed World Assumption here. Otherwise, execution of the current goal is suspended.

C. Instantiation of Variables

When a variable that has suspended unification invoked in guards is instantiated by unification invoked in some clause body, the following is done for each entry $(p,\ c)$ in the waiting list of that variable.

- The following is done for the process descriptor designated by p.

  (1) *Clause State* of the cth clause is set to *candidate*, and c is entered in *Candidate Queue*.

  (2) All elements of the form $(c,\ -)$ ('$-$' stands for 'don't care') are deleted from *Suspend/Fail Table*.

  (3) This process descriptor is entered in *Process Queue*.

D. Execution of the Body

If the body of a clause selected for commitment (say the cth clause) has a recursive call, the following tasks are done.

(1) Assume that the arguments of the head and the arguments of the recursive call differ in the k_1th, k_2th, ..., k_lth arguments. For each k_i $(i = 1, \ldots, l)$, the following are done.

  - Elements of the form $(d,\ k_i)$ are searched from *Suspend/Fail Table*, and for each d, the following are done.

    - If *Clause State* of the dth clause is $fail$, *Fail Count* is decremented by 1. If it is *suspend*, the entry of the waiting list pointed to by *Clause Backward Pointer* is eliminated.

    - *Clause State* of the dth clause is set to *candidate*, and d is entered in *Candidate Queue*.

    - Elements of the form $(d,\ -)$ are deleted from *Suspend/Fail Table*.

  - The k_ith element of *Argument List* is rewritten.

(2) The cth clause is entered in *Candidate Queue*.

(3) Clause selection (cf. *B*) is performed.

If goals other than a recursive call are contained, we generate new process descriptors for them. However, short-lived goals such as unification can be executed immediately after commitment without making process descriptors. In particular, a unification goal in a clause body never suspends unless it is indirectly called from some guard, and therefore it need not prepare for suspension if we disallow any user-defined goals in guards. Thus the output unification in n-ary *merge* can be efficiently executed.

If there is no recursive call, the area for the original process descriptor can be released after all the pairs $(p,\ c)$ entered in the waiting lists of uninstantiated variables in *B-1* are eliminated. However, there are cases in which this area can be re-utilized for optimization (cf. Section 6.1.3.4).

6.1.3.2.3. Management of *Suspend/Fail Table*

The purpose of managing *Suspend/Fail Table* is to minimize the number of suspended clauses which must be woken up when some variable gets instantiated. We must carefully design the management scheme of clauses so that all the suspended clauses which must be woken up will be correctly woken up.

If n-ary *merge* is called as follows,

```
                          0th        cth
:- merge( Ys , ..., Xs , ... ).
```

the cth clause suspends. In this case, the cause of suspension lies only in the cth argument of the goal; even if another clause were selected and tail recursion took place, tail recursion itself would not remove the cause since the cth argument remains the same. However, there are more complex cases where

- the ith clause suspends or fails due to the kth argument,
- the jth clause is selected and its tail-recursive goal does not change the kth argument, but
- the ith clause must be woken up upon tail recursion.

Consider the following example:

```
Goal:                    :- p([a], []).
Program:  p(X,      X) :- true | true.
          p([A|X], Y) :- true | p(X, Y).
```

If head unification is done from the left to right, head unification of the first program clause fails at the second argument. However, we should attribute the cause of failure both to the first and the second arguments. Actually, if the second clause is selected and tail recursion takes place, the first clause becomes selectable though the second argument remains unchanged.

To generalize, when the unification of the kth argument of the cth clause suspends or fails, it suffices to enter in *Suspend/Fail Table* the pair $(c,\ i)$ for each i such that the ith argument is 'related to' the kth argument. Here, the term A is said to be 'related to' (henceforth denoted by R_t) the term B if some variable appearing in A is 'related to' some variable in B; and a variable V_1 is said to be related to a variable V_2 if V_1 and V_2 are related by the reflexive transitive closure of the following relation R_v.

- *Relation* R_v: both variables appear together in a goal of the guard (if the guard is empty, R_v is the sameness of the variables).

  Example : For the cth clause of n-ary *merge*, the quotient A/R_t of the set of arguments A by R_t is

  $$\{\{i\} \mid i = 1, \ldots, n\}.$$

For the clause

```
p(I,J,K,L,M)  :- a(I,J), b(J,K), c(L,M) | true.
```

we get

$$\{\{0,1,2\},\{3,4\}\}.$$

The number of elements that are simultaneously entered in *Suspend/Fail Table* does not exceed

$$\sum_{clauses} \begin{pmatrix}\text{maximum size of the elements of} \\ \text{`(set of arguments)}/R_t\text{'}\end{pmatrix}.$$

In the case of n-ary *merge*, this value is $O(n)$.

Note that for n-ary *merge*, we can specialize or simplify the data structure of *Suspend/Fail Table* and the operations for it, since the only possible entries are of the form $(c,\ c)$.

6.1.3.3. Properties of the Fixed-Arity Merge

We will now examine the properties of n-ary *merge* compiled using the technique presented in Section 6.1.3.2. The existence of base-case clauses will not be considered here. It will be discussed later in Section 6.1.3.3.5.

6.1.3.3.1. Space Efficiency

The size of each item of a process descriptor other than *Suspend/Fail Table* is obviously no greater than $O(n)$, and the size of *Suspend/Fail Table* is $O(n)$, as stated in Section 6.1.3.2.3. Therefore, the size of each process descriptor is $O(n)$. The size of the program code will be discussed in Section 6.1.3.3.4.

6.1.3.3.2. Time Efficiency

We consider the time required for processing a message arriving at n-ary *merge* waiting for some input.

A. *Creation of Process Descriptors:* $O(n)$, but we can ignore it since this must be done only once.

B. *Head Unification:* Only one clause is woken up when a message arrives. Then the cost for the head unification is $O(1)$, because unification must be attempted for only one argument. It is $O(1)$ even when the unification suspends or fails, because the cost accompanying the suspension or the failure (i.e., updating of *Suspend/Fail Table* and the waiting lists of variables) is $O(1)$ by using an appropriate data structure for *Suspend/Fail Table* (Section 6.1.3.2.1).

C. *Instantiation of a Variable:* As stated above, only one clause is woken up and each task shown in Section 6.1.3.2.2-C takes constant time.

D. *Tail Recursion:* When the cth clause is selected for commitment, the only clause to be checked again is the cth clause. Furthermore, only two entries of *Argument List* need be rewritten. Therefore, the time required for the recursive call is $O(1)$.

Therefore, the delay of a message arriving at n-ary *merge* in an input-wait state does not depend on n.

When the merge process is busy processing other messages, a newly arriving message may not appear in the output stream within a constant time. However, once the processing of that message starts, it ends within a constant time.

6.1.3.3.3. Order of Clause Checking

Individual clauses of n-ary *merge* are checked in the order they are entered in *Candidate Queue*. Since a selected clause is reentered at the tail of the queue, n-bounded waiting is achieved. Moreover, clauses which have suspended or failed are not in *Candidate Queue*, so they do not influence the efficiency.

6.1.3.3.4. Program Size

The codes for operations A and C in Section 6.1.3.2.2 is common to all predicates. The code for operations B and D are prepared for each predicate. In the case of n-ary *merge*, its code size is $O(n)$, because both B and D are constant-time tasks.

However, since the codes for individual clauses are almost the same, they can be parameterized with respect to the clause numbers. If this is done, the code size of the whole predicate is reduced to $O(1)$.

This parameterization could be accomplished by a sophisticated compiler capable of detecting similarities among the clauses. However, even if such a compiler were employed, it would not reduce the size of the source program ($O(n^2)$) and the time required for compilation. Furthermore, there may be only a few programs which can benefit from this optimization. Considering all these things, the most realistic approach is to let the system provide the code for n-ary *merge*.

Now we have shown that the code size for n-ary *merge* can be made independent of n. This is still unsatisfactory, however, since the system will have to provide n-ary *merge* for various n's. If these were provided as separate codes, the amount of codes would be $O(n_{max})$, where n_{max} is the maximum value of n.

However, here again, drastic optimization is possible. Because the code for n-ary *merge* remain almost the same even if n varies, it can be parameterized with

respect to n. This parameterization realizes a family of predicates for merging any number of inputs with a single code whose size is independent of n_{max}. Note that it is mandatory that these codes be provided by the system, because the size of the corresponding source program is $O({n_{max}}^3)$.

6.1.3.3.5. Base Case

To terminate n-ary *merge*, a clause describing the base case must be carefully supplied. The clause

```
merge(Ys, [], ..., []) :- true | Ys=[].
```

is logically correct, but it cannot be efficiently processed by the above implementation technique. Suppose that all input streams except the last one have been terminated, that is, the original goal

`merge(`Ys`,` X_1`, ...,` X_{n-1}`,` X_n`)`

has been reduced to the following goal:

`merge(`Ys'`, [], ..., [],` Xs'`)`

If we use the technique shown in Section 6.1.3.2.2, the delay of the message in Xs' may no longer be small, since we may check all the input streams when we execute the base-case clause. One way to avoid it is to record (in the process descriptor) which input clauses have been terminated and to use this information to optimize the code for the base case. An alternative, much simpler solution uses the '`otherwise`' construct proposed by Shapiro and Takeuchi [1983]:

```
merge(Ys, [], ..., []) :- otherwise | Ys=[].
```

As explained in Section 4.10, an '`otherwise`' goal in a guard succeeds if and when all other guards fail. A clause containing '`otherwise`' in its guard should be put into *Candidate Queue* only after *Fail Count* reaches the number of clauses not containing '`otherwise`'. With '`otherwise`', the base-case clause does not increase the delay of message transfer or the size of the process descriptor, since it is never scheduled until all input streams are terminated. When the base-case clause is scheduled, all input streams must be instantiated (usually to `[]`) and therefore that clause turns out to succeed or fail without suspension.

6.1.3.4. Dynamic Change of the Number of Input Streams

A merge predicate with a fixed arity is useful only when the number of inputs is statically known. We will now expand this to allow the addition of new streams and the removal of terminated streams. The program shown below has an additional (say the (-1)th) argument for accepting requests for new input streams.

- The kth clause (transfer) $(k = 1, \ldots, n)$

```
merge(S, Ys, X1, ...,[X|Xk], ..., Xn) :-  true | Ys=[X|Zs],
merge(S, Zs, X1, ...,  Xk,   ..., Xn).
```

- The 0th clause (addition)

```
merge([Xn+1|S], Ys, X1, ..., Xn) :- true |
merge(   S,     Ys, X1, ..., Xn, Xn+1).
```

- The $(-k)$th clause (removal) $(k = 1, \ldots, n)$

```
merge(S, Xs, X1, ..., [],  ..., Xn-1, Xn)  :- true |
merge(S, Xs, X1, ..., Xn, ..., Xn-1).
```

- Base Case

```
merge([], Xs) :- true | Xs=[].
```

The clauses for adding or removing streams are not tail recursive, but in those clauses the clause heads and the body goals are very similar. Therefore, if the process descriptors for those body goals can be constructed by modifying the original ones, it is much more efficient than to create them from scratch.

In GHC, process descriptors cannot be managed by a simple stack scheme but by a general memory management technique. Here we will suppose that the Buddy system (Knuth [1968]) is employed. The size of each partitioned area will then be a power of two, and each process descriptor is created in one of these areas. When it is created, each item must be placed depending on the size of the area allocated so that the cost of relocation with the addition and removal of streams can be minimized. Then, even when the number of inputs changes, most of the existing information need not be moved as long as the area is large enough to accommodate the new descriptor.

We will show the operations to be performed when the $(-n)$th to 0th clauses are selected and the process descriptor is reusable. We use the value *unused* as one of the possible values that *Clause State* can take, and when the area for *Clause States* is allocated, we fill its unutilized portion with *unused*'s.

For simplicity, the following description is specialized to the merge predicate.

A. When the 0th Clause is Selected and a New Stream is Added

(1) (Operations accompanying the addition of the $\pm(n+1)$th clauses) If *Clause States* of the $\pm(n+1)$th clauses are not *candidate*, they are set to *candidate* and those clauses are entered in *Candidate Queue*.

(2) The 0th clause is entered in *Candidate Queue*.

(3) The (-1)th and the $(n+1)$th arguments of *Argument List* are updated.

(4) The program code is replaced (If the program is parameterized with respect to n, only the parameter value is replaced).

We must note that *Clause States* of the $\pm(n+1)$th clauses may already be *candidate*, in which case these clauses are already in *Candidate Queue*. This happens in case the process descriptor area had been used by some predicate with more clauses.

B. When the $(-c)$th Clause $(c > 0)$ is Selected and an Empty Stream is Removed

(1) (Operations accompanying the change of the cth argument) Elements of the form $(c',\ c)$ (only $(c,\ c)$ can exist, if any) are searched from *Suspend/Fail Table*. For each c', the following is done.

- If *Clause State* of the c'th clause is $fail$, *Fail Count* is decremented by 1. If it is *suspend*, the entry in the waiting list pointed to by *Clause Backward Pointer* for the c'th clause is deleted.
- *Clause State* of the c'th clause is set to *candidate*, and c' is entered in *Candidate Queue*.
- Elements of the form $(c',\ -)$ (only $(c,\ c)$ can exist, if any) are deleted from *Suspend/Fail Table*.

(2) (Operations accompanying disappearance of the $\pm n$th clauses)

- If *Clause State* of the nth clause is $fail$, *Fail Count* is decremented by 1. The same operation is done also for the $(-n)$th clause.
- Elements of the form $(\pm n,\ -)$ (only $(n,\ n)$ and $(-n,\ n)$ can exist, if any) are deleted from *Suspend/Fail Table*.
- Nothing is done with the $\pm n$th clauses in *Candidate Queue*. When those clauses, which have disappeared, are dequeued, their *Clause States* are changed to $undefined$.

(3) The $(-c)$th clause is entered in *Candidate Queue*.

(4) The cth argument of *Argument List* is updated.

(5) The program code is replaced.

Step (1) is performed in case the $(-c)$th clause is selected after the cth clause is entered in *Suspend/Fail Table* due to the failure of the unification of the cth argument.

It is clear that both A and B can be performed within a constant time.

When the area for the current process descriptor cannot be reused to add a new stream, it is necessary to allocate a new area of twice the size and to create a new descriptor in that area. On the contrary, when it becomes possible to express the process descriptor with half the size of the current area (by the repeated removal of streams), the process descriptor can be packed and the unused area can be freed. These operations are shown below.

A′. Addition of Streams Entailing Moving to a New Area

(1) An area twice as large as the current process descriptor area is allocated.

(2) All items of the original process descriptor are copied.

(3) The entries designated by all meaningful *Clause Backward Pointers* (i.e., ones for suspended clauses) are made to point to the new area.

(4) The operations described above in A are performed.

B′. Deletion of Streams Entailing Compaction

(1) The operations described above in B are performed.

(2) *Candidate Queue* is examined and the all the clauses other than the $-(n-1)$th to the $(n-1)$th clauses are deleted, if any.

(3) The original process descriptor is packed in the top half of the current area.

(4) The bottom half of the area is freed.

Step (2) is necessary in this case because we have to reduce the number of elements in *Candidate Queue* to free the area.

We will now consider the time complexity of A' and B'. If the time needed for memory allocation and release is ignored, both A' and B' can be done within a time of $O(n)$. The time complexity of memory allocation and release by the Buddy system is

$$O(\log(\text{size of the whole area managed by the Buddy system})).$$

This value, however, is determined only by the execution environment of the program, which is independent of n. Therefore, if the execution environment is fixed, the time needed for A' and B' is $O(n)$.

In order to add and remove streams within an average time of $O(1)$, it must be guaranteed that the frequency of the operations A' and B' is at most once every $O(n)$ changes of the number of streams. This is easily achieved by doing B' only when it becomes possible to represent the process descriptor with (for example) one-fourth of the current area.

6.1.4. Implementation of the Distribute Predicate

This section describes the outline of the implementation technique of the predicate for message distribution. We do not go into details since the implementation is much simpler: We need not implement multiple waits, which is the main source of the complexity of stream merging.

6.1.4.1. Distribution to a Fixed Number of Output Streams

The predicate *distribute* with n output streams is expressed by $n+1$ clauses of the following form:

- The kth clause ($k = 1, \ldots, n$)

```
distribute([(k,X)|Xs], Y1, ...,Yk, ..., Yn)  :- true | Yk=[X|Zk],
distribute(    Xs,     Y1, ...,Zk, ..., Yn).
```

- The 0th clause

```
distribute([], Y1, ..., Yk) :- true | Y1=[], ..., Yk=[].
```

First, we will consider the situation where there is no wait. We must enable random access to clauses because, if the first to the nth clauses were individually checked, the time complexity would be $O(n)$. The DEC-10 Prolog compiler (Warren [1977]) generates a code that selects clauses using the hash value of the principal functor of the first argument. In the case of *distribute*, hashing by the *tertiary* functor (a functor of the third level) of the first argument is necessary to select a clause within a constant time.

Next, we will consider how to achieve the code size of $O(1)$. Parameterization of the code of each clause is of course necessary. In the case of *distribute*, we should further make use of the fact that clauses can be selected by simple *indexing* rather than hashing; a hash table requires an area of $O(n)$.

What if there is a wait? The first to the kth clauses all wait for the 0th argument. If they individually wait, the desired efficiency cannot be achieved. Those clauses should be managed together also while waiting. In other words, they should be entered in the waiting list of a variable as a cluster of clauses. When the 0th argument is instantiated, the appropriate clause must be selected by indexing.

6.1.4.2. Dynamic Change of the Number of Output Streams

As in the case of *merge*, it is an important function to dynamically change of the number of output streams. This can be implemented by adding the following clauses:

- Addition

```
distribute([grow(Y_{n+1})|Xs],Y_1, ..., Y_n) :- true |
distribute(      Xs,       Y_1, ..., Y_n, Y_{n+1}).
```

- Deletion

```
distribute([shrink| Xs],Y_1, ..., Y_{n-1}, Y_n) :- true |
distribute(     Xs,      Y_1, ..., Y_{n-1}).
```

In order to efficiently change the number of output streams, a technique similar to the one described in Section 6.1.3.4 can be applied.

6.1.5. Applying Implementation Technique of Distribution Predicates to Mutable Arrays

The lack of mutable arrays (arrays of rewritable elements) is often counted as one of the problems of Prolog. Of course, arrays can be simulated by *assert* and *retract*, but such arrays are not *logical* arrays. One direction to realize logical arrays is to make a correspondence

- Arrays: Data of the array type
- Operations on arrays: Predicates that manipulate array arguments

and to gain efficiency by a dedicated data structure. However, it is also possible to make the following correspondence

- Arrays: Goals (i.e., processes)
- Operations on arrays: Streams of messages

by the program

```
array(n, S) :- array(S, X_1, ..., X_n).
array([read(k,Y_k)|S], X_1, ..., X_k, ..., X_n)  :- true | Y_k=X_k,
array(       S,        X_1, ..., X_k, ..., X_n).       (for k = 1,...,n)
array([write(k,Y_k)|S],X_1, ..., X_k, ..., X_n)  :- true |
array(       S,        X_1, ..., Y_k, ..., X_n).       (for k = 1,...,n).
```

This is a rather natural solution if we regard arrays as mutable *objects*. Eriksson and Rayner [1984] also have independently proposed this solution. This program is very similar to *distribute*, and constant-time accessing and updating is realized by applying the implementation technique for *distribute*. It is also possible to add clauses for inquiring and/or changing the number of elements. Note that all transactions with an array object are done through the argument `S` of the binary '`array`' predicate; a programmer does not have direct access to any of the array elements.

6.1.6. Summary

We investigated the properties of n-ary *merge* written in GHC and presented its implementation which transfers each message with a delay independent of n. Furthermore, we showed that an input stream can be added and removed within an average time of $O(1)$. With respect to n-ary *distribute* also, we gave the outlines of implementation as efficient as *merge*. Mutable arrays that allow constant-time accessing and updating were shown to be realizable by the same implementation technique as that for *distribute*.

However, it was concluded that these predicates should be supported directly by the system. If the system provides them, *merge* and *distribute* for all arities can be realized with the constant-size code. It is unrealistic to obtain the code by compiling a source program, not for the reason of the efficiency of the code obtained, but for the reason of the bulk of the source program and the time needed for compilation. Nevertheless, it is favorable in many respects (e.g., for the construction of programming systems) that the semantics of the system-supplied code is expressible as a GHC program.

An alternative to the system predicate approach is to enhance the descriptive power of source programs. The fact that the above predicates allow compact representation at the object-code level indicates sparseness or redundancy of their source codes. Therefore, if we introduce new source-level notations, they will enable more concise representation of those predicates and also will make the compiler approach more realistic.

6.2. Concurrent Prolog Compiler and GHC Compiler on Top of Prolog

This section describes compilers of Concurrent Prolog and GHC which use (sequential) Prolog as target and implementation languages. Object programs obtained can further be compiled into machine codes by a Prolog compiler. Due to the similarity among the source, target and implementation languages, the compilers and the runtime supports were small and very rapidly developed. Benchmark tests showed that (twice) compiled Concurrent Prolog programs ran 2.7 to 4.4 times faster and 2.7 to 5.3 times slower than comparable Prolog programs running on the interpreter and compiler, respectively, of the same Prolog system. GHC programs ran almost as fast as the comparable Concurrent Prolog programs with mode declarations. The contents of this section is based on (Ueda and Chikayama [1984b]) and (Ueda and Chikayama [1985]).

6.2.1. Introduction

To evaluate a programming language in terms of efficiency, one must try to have a serious implementation of the language. It is unfair to underrate a language by its unserious implementation for the purpose of rapid prototyping. In the case of Concurrent Prolog, the interpreter in the original paper (Shapiro [1983a]) must be regarded as such a prototype. Although the availability of the interpreter was quite important for the popularization of the language, it was useful only for the experiments of small programs because the the slowdown from the bare Prolog system on which the interpreter ran amounted to two orders of magnitude. This motivated us to develop a Concurrent Prolog compiler. We made a compiler of Concurrent Prolog first, because Guarded Horn Clauses had not been invented at that time. The compiler of GHC, based on the Concurrent Prolog compiler, was developed very rapidly after GHC was invented.

The development of these compilers has the following two major purposes:

(1) To provide a programming environment in which one can write and test programs of considerable size.

(2) To know how fast Concurrent Prolog and GHC programs run.

We chose (sequential) Prolog for the target and the description languages for the following reasons.

(1) A Prolog program can be compiled into efficient machine codes (Warren [1977]).

(2) Similarity among the source, target, and description languages enables rapid development.

(3) We can obtain a portable implementation.

(4) The laborious work of writing system predicates is greatly reduced by interfacing between Concurrent Prolog/GHC and Prolog.

Our approach is similar to the approach taken by Gregory [1984] when he wrote a PARLOG system on top of Prolog. However, it did not optimize unification, and its performance on a compiler-based Prolog implementation has not been reported. We tried to get maximum efficiency on a compiler-based Prolog implementation. The difference of underlying implementations is never trivial, because accumulation of small hacks may greatly improve the efficiency under an optimizing compiler, while the improvement should not be so drastic under an interpreter.

6.2.2. Linguistic and Non-Linguistic Features

Our implementation of Concurrent Prolog is basically a compiler version of the original interpreter by Shapiro [1983a]. Some linguistic extensions we have made are listed below. These features are in common with the GHC compiler, since the GHC compiler is based on the Concurrent Prolog compiler.

(1) Input and output have been made declarative. There is no '`read`' or '`write`' predicate à la Prolog. Instead, we have '`instream`' and '`outstream`' predicates which take one argument: a stream of request messages. Each request message must be an appropriate Prolog I/O goal. For example, if the goal

```
outstream([write(ok), nl | _])
```

is executed, the message '`ok`' followed by a newline is output. In order to guarantee the uniqueness of the input and output streams, neither '`instream`' nor '`outstream`' can be called twice. While '`outstream`' accepts messages related to output only, '`instream`' accepts messages related to output as well as to input. This feature of '`instream`' is used for synchronizing output to the terminal and input from it. In fact, we can perform all I/O operations using '`instream`' only; '`outstream`' has been provided for mere convenience.

(2) Mode declaration facilities similar to those of DEC-10 Prolog (Bowen, Byrd, Pereira, Pereira and Warren [1983]) have been provided. The purpose is to get smaller and more efficient codes.

On the other hand, our compilers inherit the following linguistic and non-linguistic restrictions from the original interpreter.

(1) The scheduling of candidate clauses is depth-first and hence unfair. Moreover, when some clause suspends, the partial result of computation is not retained for subsequent execution but just discarded.

(2) There is no distinction between suspension and failure. A goal for which there are no immediately selectable clauses may be re-scheduled, whether the cause is finite failure or suspension.

(3) Suspended goals do busy-waiting.

These restrictions greatly simplify implementation on Prolog and are very effective for performance. Although they might look true restrictions at a glance, they cause little inconvenience to the execution of most Concurrent Prolog and GHC programs we have written:

(1) Most guards are used just for synchronization and conditional branching rather than 'OR-parallel problem solving'. There have been few programs that require the (pseudo-) parallel execution of two or more guards.

(2) Typical Concurrent Prolog and GHC programs are written so that all goals may succeed except for small ones in guards. Failure of a whole program is a rather exceptional situation in Concurrent Prolog and GHC.

(3) By employing bounded depth-first scheduling (see below), the frequency of suspension can be made small in most applications.

The GHC compiler has another restriction: It does not allow any user-defined goals in a guard. The purpose of this restriction is to enable simple static analysis of suspension. Note that this restriction is very similar to the restriction adopted by Flat Concurrent Prolog (Mierowsky, Taylor, Shapiro, Levy and Safra [1985]).

Non-linguistic features include scheduling strategies and trace facilities.

Since conjunctive goals must be solved in (pseudo-) parallel, we have to implement a scheduler of goals. We decided to use one goal queue, and employed 100-bounded depth-first scheduling as the default strategy. *N-bounded depth-first scheduling* means that each newly-scheduled goal is n-reducible. A *newly-scheduled goal* is a goal which was enqueued at the rear previously and is now dequeued at the front. That a goal G is $n(>0)$-*reducible* means that when G is reduced to B_1, ..., B_m by the following clause,

$$H \ \texttt{:-}\ G_1\texttt{,}\ \ldots\texttt{,}\ G_k \mid B_1\texttt{,}\ \ldots\texttt{,}\ B_m\texttt{.} \qquad (k>0, m>0).$$

each B_i $(i=1,\ldots,m)$ is $(n-1)$-reducible prior to the execution of the other goals in the queue. That a goal G is 0-*reducible* means that G must be pushed at the rear of the goal queue and the goal at the front must be executed next.

It is easy to see that n-bounded depth-first scheduling is so general as to interpolate between breadth-first and depth-first scheduling. Bounded depth-first scheduling has both the fairness of breadth-first scheduling and the efficiency of depth-first scheduling, as we will see later.

The initial bound value for each goal can be specified at run time. When bounded depth-first scheduling is unnecessary, one can tell the compilers to generate a simpler code that uses (unbounded) depth-first scheduling and will gain more efficiency. We can also mix these two strategies in a program. In stream-oriented programs, the check of a bound value is important only for a producer goal which may generate arbitrarily many data autonomously: Other goals are suspended by

the synchronization mechanism of Concurrent Prolog and GHC irrespective of the scheduling strategy. Therefore, we can restrict the bounded depth-first scheduling to the producer predicates while retaining the fairness of scheduling.

Execution trace is enabled by compiling a source program with the 'trace' option.

6.2.3. Compilation Technique

A general advantage of a compiler approach is that we can statically determine parts of what we must determine at run time in an interpreter approach. In the case of Concurrent Prolog and GHC, such parts include scheduling and unification. These two aspects are discussed in the following.

6.2.3.1. Scheduling

Our compilers have inherited the following notions from Shapiro's original interpreter:

(1) a queue (represented as a difference list) of goals to be solved

(2) a flag showing whether computation may deadlock or not

(3) a cycle marker for detecting termination and deadlock of computation.

However, our compilers have not inherited a scheduler predicate. The compilers generate one Prolog predicate (object code) for each Concurrent Prolog/GHC predicate. When such a Prolog predicate is called, it is given as arguments a goal queue, which will be called a *continuation* hereafter. The called predicate must perform scheduling tasks by itself, that is, it must appropriately handle the given continuation and start another (Prolog) goal when it has finished the tasks for itself or when it swaps out its caller. The goal to be started is either the first goal in the given continuation or a goal provided by the predicate itself. In the former case, we must pop the first goal, provide it with the rest of the continuation and call it. In the latter case, the given continuation is passed to the new goal. A Prolog predicate generated by the compilers never fails unless it calls a nonexistent predicate.

Each Prolog predicate generated by the compilers consists of the following three parts:

(1) (optional) *Prelude part* for tracing and clause indexing,

(2) *Clause-by-clause part* (one Prolog clause for each Concurrent Prolog/GHC clause) for goal reduction, and

(3) *Postlude part* for re-scheduling itself when all clauses fail to be selected or the bound value reaches zero.

Figure 6.5 shows the source and the object programs of quicksort in Concurrent Prolog. Figure 6.6 shows the source and the object programs of quicksort in GHC. Note that candidate clauses are tested sequentially in our implementation.

Each compiled clause has five additional arguments:

(1) A counter maintaining the current bound value used for bounded depth-first scheduling.

(2), (3) The head and the tail of the difference list representing a continuation.

(4) A deadlock flag showing whether or not some goal has been reduced since the last occurrence of the cycle marker. This flag is checked by the next cycle marker for deadlock detection.

(5) The initial bound value given at run time.

Each element of a continuation has the following form:

$$\texttt{\$(}\mathit{Goal}\texttt{, }\mathit{Qh}\texttt{, }\mathit{Qt}\texttt{, }\mathit{Deadlock\_flag}\texttt{).}$$

Goal is a Prolog goal corresponding to some Concurrent Prolog/GHC goal. When *Goal* is called, its additional arguments must be given appropriate values. Qh, Qt, and $Deadlock\_flag$ are 'taps' of *Goal* used for this purpose. That is, Qh, Qt, and $Deadlock\_flag$ have been unified with the second, the third, and the fourth additional arguments of *Goal*, respectively.

A Concurrent Prolog or GHC clause

$$\mathit{Head}\ \texttt{:-}\ \mathit{Guard}\ \texttt{|}\ \mathit{Body}\texttt{.}$$

is transformed into a Prolog clause of the following form:

(receiving arguments) :-
(*Head* unification),
(bound check),
(executing *Guard*), !,
(decrementing bound),
(scheduling *Body*).

The 'bound check' and the 'decrementing bound' parts are not generated if depth-first scheduling is specified.

The last part, 'scheduling *Body*', does the following things.

(1) When no body goals exist (i.e., *Body* is '`true`'), the first goal in the continuation is called.

(2) When just one body goal exists, that goal is called with the same continuation that the current clause has received.

```
qsort([Pivot|Xs],Ys0,Ys2) :-          % Ys0-Ys2:   d-list
  part(Xs?,Pivot,Small,Large),
  qsort(Small?,Ys0,[Pivot|Ys1]),
  qsort(Large?,Ys1,Ys2).

qsort([],Ys,Ys).                      % Ys-Ys:   empty d-list

part([X|Xs],Pivot,Small,[X|Large]) :- Pivot <  X |
  part(Xs?,Pivot,Small,Large).

part([X|Xs],Pivot,[X|Small],Large) :- Pivot >= X |
  part(Xs?,Pivot,Small,Large).

part([],_,[],[]).
```

(a) Concurrent Prolog source program

```
:-fastcode.                           % Compiler option
:-public qsort/8.
:-mode qsort(?,?,?, +,?,-,+,+).
qsort(Arg1,Ys0,Ys2, Bold,H,T,Flag,B0) :-
  ulist(Arg1,Pivot,Xs),               % Arg1=[Pivot|Xs]
  Bold > 0, !,                        % Bound check
  Bnew is Bold-1,                     % Decrement bound
  part(Xs?,Pivot,Small,Large,
    Bnew,
    [$(qsort(Small?,Ys0,[Pivot|Ys1],
             Bnew,H1,T1,Flag1,B0),
                  H1,T1,Flag1   ),    % Push 1st qsort
     $(qsort(Large?,Ys1,Ys2,
             Bnew,H2,T2,Flag2,B0),
                  H2,T2,Flag2   )     % and 2nd qsort
    | H],                             % to the top of the continuation
    T,nd,B0).

qsort(Arg1,Ys0,Ys1, Bold,H,T,Flag,B0) :-
  unil(Arg1), unify(Ys0,Ys1),         % Ys0=Ys1, Arg1=[]
  Bold > 0, !,                        % Bound check
  H=[$(Goal,H1,T,nd)|H1],             % Pop the first goal, give
  incore(Goal).                       % appropriate continuation and
                                      % deadlock flag, and schedule it
```

(b) Object program in DEC-10 Prolog *(continued on the next page)*

Fig. 6.5. Compiling Concurret Prolog into Prolog

```
qsort(Arg1,Arg2,Arg3,                    % Suspension processing
  Bold,
  [$(Goal,H1,T1,Flag)|H1],               % Pop the first goal,
  [$(qsort(Arg1,Arg2,Arg3,
           B0,H2,T2,Flag2,B0),           % push qsort itself,
              H2,T2,Flag2   )|T1],
  Flag,B0) :- incore(Goal).              % and call it

:-public part/9.
:-mode part(?,?,?,?, +,?,-,+,+).
part(Arg1,Pivot,Small,Arg4, Bold,H,T,Flag,B0) :-
  ulist(Arg1,X1,Xs),                     % Arg1=[X1|Xs]
  ulist(Arg4,X2,Large),                  % Arg4=[X2|Large]
  unify(X1,X2),                          % X1=X2
  Bold > 0,                              % Bound check
  cpwait(Pivot,Pivot_w),                 % Wait for Pivot
  cpwait(X1,X1_w),                       % and X1
  Pivot_w < X1_w, !,                     % and compare them
  Bnew is Bold-1,                        % Decrement bound
  part(Xs?,Pivot,Small,Large, Bnew,H,T,nd,B0).

part(Arg1,Pivot,Arg3,Large, Bold,H,T,Flag,B0) :-
  ulist(Arg1,X1,Xs), ulist(Arg3,X2,Small),
  unify(X1,X2), Bold > 0,
  cpwait(Pivot,Pivot_w), cpwait(X1,X1_w),
  Pivot_w >= X1_w, !, Bnew is Bold-1,
  part(Xs?,Pivot,Small,Large, Bnew,H,T,nd,B0).

part(Arg1,_,Arg3,Arg4, Bold,H,T,Flag,B0) :-
  unil(Arg1), unil(Arg3),                % Arg1=[], Arg3=[],
  unil(Arg4),                            % Arg4=[]
  Bold > 0, !,
  H=[$(Goal,H1,T,nd)|H1], incore(Goal).

part(Arg1,Arg2,Arg3,Arg4,
  Bold, [$(Goal,H1,T1,Flag)|H1],
  [$(part(Arg1,Arg2,Arg3,Arg4,
          B0,H2,T2,Flag2,B0),
             H2,T2,Flag2   )|T1],
  Flag,B0) :- incore(Goal).
```

(b) Object program in DEC-10 Prolog *(continued from the previous page)*

Fig. 6.5. Compiling Concurret Prolog into Prolog *(continued)*

```
qsort([Pivot|Xs], Ys0, Ys2) :- true |
  part(Xs, Pivot, Small, Large),
  qsort(Small, Ys0, [Pivot|Ys1]),
  qsort(Large, Ys1, Ys2).

qsort([],         Ys0, Ys1) :- true | Ys0 = Ys1.

part([X|Xs], Pivot, Small, Large) :- Pivot <  X |
  Large = [X|L1], part(Xs, Pivot, Small, L1).

part([X|Xs], Pivot, Small, Large) :- Pivot >= X |
  Small = [X|S1], part(Xs, Pivot, S1, Large).

part([],     _,     Small, Large) :- true       |
  Small = [], Large = [].
```

(a) GHC source program

```
:-fastcode.
:-public qsort/8.
:-mode qsort(?,?,?, +,?,-,+,+).
qsort(Arg1,Ys0,Ys2, Bold,H,T,Flag,B0) :-
  nonvar(Arg1), ulist(Arg1,Pivot,Xs),
  Bold > 0, !, Bnew is Bold-1,
  part(Xs,Pivot,Small,Large,
    Bnew,
    [$(qsort(Small,Ys0,[Pivot|Ys1],
             Bnew,H1,T1,Flag1,B0),
                  H1,T1,Flag1   ),
     $(qsort(Large,Ys1,Ys2,
             Bnew,H2,T2,Flag2,B0),
                  H2,T2,Flag2   )
    | H],
    T, nd, B0).

qsort(Arg1,Ys0,Ys1, Bold,H,T,Flag,B0) :-
  nonvar(Arg1), unil(Arg1),
  Bold > 0,
  Ys0 = Ys1,!,
  H = [$(Goal,H1,T,nd)|H1], incore(Goal).
```

(b) Object program in DEC-10 Prolog *(continued on the next page)*

Fig. 6.6. Compiling GHC into Prolog

```
qsort(Arg1,Arg2,Arg3,
  Bold, [$(Goal,H1,T1,Flag) | H1],
  [$(qsort(Arg1,Arg2,Arg3,
           B0,H2,T2,Flag2,B0),
              H2,T2,Flag2   ) | T1],
  Flag,B0) :- incore(Goal).

:-public part/9.
:-mode part(?,?,?,?, +,?,-,+,+).
part(Arg1,Pivot,Small,Large, Bold,H,T,Flag,B0) :-
  nonvar(Arg1), ulist(Arg1,X,Xs),
  Bold > 0,
  nonvar(Pivot), nonvar(X), Pivot < X,
  Large = [X|L1], !,
  Bnew is Bold-1,
  part(Xs,Pivot,Small,L1, Bnew,H,T,nd,B0).

part(Arg1,Pivot,Small,Large, Bold,H,T,Flag,B0) :-
  nonvar(Arg1), ulist(Arg1,X,Xs),
  Bold > 0,
  nonvar(Pivot), nonvar(X), Pirot >= X,
  Small = [X|S1], !,
  Bnew is Bold-1,
  part(Xs,Pivot,S1,Large, Bnew,H,T,nd,B0).

part(Arg1,_,Small,Large, Bold,H,T,H,I) :-
  nonvar(Arg1), unil(Arg1),
  Bold > 0,
  Small = [], Large = [], !,
  H = [$(Goal,H1,T,nd) | H1], incore(Goal).

part(Arg1,Arg2,Arg3,Arg4,
  Bold, [$(Goal,H1,T1,Flag) | H1],
  [$(part(Arg1,Arg2,Arg3,Arg4,
          B0,H2,T2,Flag2,B0),
             H2,T2,Flag2   ) | T1],
  Flag,B0) :- incore(Goal).
```

(b) Object program in DEC-10 Prolog *(continued from the previous page)*

Fig. 6.6. Compiling GHC into Prolog *(continued)*

(3) When two or more body goals exist, the second and the subsequent goals are put at the front of the continuation, and the first goal is called with the modified continuation.

Upon these calls, the fourth additional argument, the deadlock flag, is set to '`nd`' (for 'no deadlock').

The GHC compiler gives special treatment to unification goals in a clause body. Since we disallow nested guards, such unification goals never suspend. So we need not schedule them normally but we can execute them immediately upon commitment. Thus the treatment of body goals described above applies only to those other than unification.

Of the above three cases, only the first case needs an indirect call; in the other cases, at least one of the body goals is directly called as long as the current bound value is not zero.

Avoiding indirect calls is important for efficiency. A major application of Concurrent Prolog and GHC is to describe a distributed system in which constituent processes, represented as conjunctive goals, communicate with one another using shared variables as streams. In this case, most of the reductions use tail-recursive clauses having just one body goal except for output unification. Our compilers translate such clauses into tail-recursive Prolog clauses. Since advanced Prolog implementations realize tail-recursion optimization which avoids the growth of the local stack and re-utilizes information left on the stack, a tail-recursive Concurrent Prolog/GHC program is expected to have good properties. Assume that 100-bounded depth-first scheduling is used and that much less than 1% of reductions use clauses with no body goals other than unification. Then, 99% of predicate calls are done by efficient direct scheduling.

The clause for handling suspension is included in the postlude part of each predicate. It pushes the current goal at the rear of the given continuation, and calls its first goal.

Deadlock and termination are detected by a cycle marker: a call to the system predicate '`$END`' (Figure 6.7). This predicate receives a continuation and a deadlock flag, and simply terminates if the given continuation is empty. If the continuation is not empty and the deadlock flag has been set to '`nd`' since the last call of '`$END`', it enqueues itself, resets the deadlock flag to '`d`' (for 'deadlock'), and calls the first goal in the continuation. Otherwise, the predicate '`$END`' fails. The goal '`$END`' is given as the continuation of an initial goal which is input from the terminal.

6.2.3.2. Unification

In Concurrent Prolog and GHC, unification of two terms may suspend even when they are unifiable in the ordinary sense. Therefore, its object code in Prolog

```
:- public '$END'/3.
'$END'([],_,_) :- !.                    % Succeeds if no goals remain

'$END'([$(Goal,H1,T1,d)|H1],            % Pop the first goal
       [$('$END'(H2,T2,Dnd2),           % and push itself
                 H2,T2,Dnd2)|T1],
       nd) :-                           % If deadlock flag is 'nd'
   incore(Goal).                        % then call the popped goal
```

Fig. 6.7. System predicate for the detecion of deadlock and termination

```
:- public ulist/3.    :- mode ulist(?,-,-).
ulist([H|T],H,T) :- !.
ulist(X?,H,T) :- nonvar(X), ulist(X,H,T).

:- public unil/1.     :- mode unil(?).
unil([]) :- !.
unil(X?) :- nonvar(X), unil(X).

:- public cpwait/2.   :- mode cpwait(?,?).
cpwait(X?,Y) :- !, nonvar(X), cpwait(X,Y).
% Here, 1st arg is a non-variable.
cpwait(X,X).
```

(a) for Concurrent Prolog

```
:- public ulist/3.   :- mode ulist(+,-,-).
ulist([H|T], H, T).

:- public unil/1.    :- mode unil(+).
unil([]).
```

(b) for GHC

Fig. 6.8. Some system predicates for unification and synchronization

must realize the suspension mechanism. The Concurrent Prolog compiler represents read-only annotations by means of a function symbol '`?`', which is appropriately interpreted in the unification routines. The GHC compiler realizes suspension by using the extralogical predicates '`nonvar`' and '`==`'.

The suspension mechanism makes unification a little bit heavier. However, for unification between a goal and a clause head, specialized unification procedures can be used depending on the form of the head, because the form of the head can be analyzed statically. The use of specialized unification procedures diminishes run-time overhead.

The code for head unification is expanded at the beginning of each clause body in the form of a sequence of goals. Assume that one of the head arguments of some source clause is a list `[`T_1`|`T_2`]`, where T_1 and T_2 are some terms. The corresponding Prolog code first tries to unify the goal argument X with the term `[`Car`|`Cdr`]` where Car and Cdr are variables, and if successful, executes the goals for processing its Car and Cdr according to the forms of T_1 and T_2, which may have been expanded also. This idea is similar to the one employed in the DEC-10 Prolog compiler (Warren [1977]): The only difference is that our compiler can expand a unification procedure to any level, as Warren's new abstract Prolog machine architecture (Warren [1983]) enables.

Note that some unification procedures cannot be expanded at all: In the case of Concurrent Prolog, general unification procedure must be used for a variable which occurs more than once in a head (e.g., the variable `Ys` in '`qsort`' in Figure 6.5). In the case of GHC, two or more occurrences of a variable in a clause head are 'compared literally' by the Prolog predicate '`==`' not to instantiate the caller.

Figure 6.8 shows the definitions of some unification and synchronization procedures used in the program in Figures 6.5 and 6.6. The '`cpwait`' predicate is used for interfacing between Concurrent Prolog and Prolog. Note that the object code for unification fails upon suspension: Suspension processing is done by the last clause of each predicate.

Mode declaration facilities enable us to declare one of the following three modes for each argument of a predicate.

(1) *Index mode* ('`+`'): tells the compiler to generate a code that takes advantage of the clause indexing feature of the underlying Prolog implementation. The object code of a predicate having this mode has a two-stage structure: the first stage for waiting for the arguments of this mode, and the second stage for clause selection. This mode is effective when there are lots of clauses.

(2) *Normal mode* ('`?`'): specifies that the argument be processed in the ordinary way.

(3) *Output mode* ('`-`'): declares that the goal argument is always an uninstantiated non-read-only variable. This mode is effective only in the Concurrent Prolog

compiler. For arguments of the output mode, implicit head unification of Prolog is used instead of explicit unification procedures.

Figure 6.9 shows how object codes generated by the Concurrent Prolog compiler are affected by a mode declaration.

6.2.4. Performance

Detailed performance evaluation was made for the Concurrent Prolog compiler rather than the GHC compiler to make a reasonable comparison between the compiler and the original interpreter. As for the GHC compiler, we only note that compiled GHC programs ran almost as fast as the comparable Concurrent Prolog programs with mode declarations.

Table 6.1 shows some benchmark results. For each program, we obtained four timing data from the Concurrent Prolog compiler: with bounded depth-first/depth-first scheduling and with/without mode declarations. The benchmark programs were timed also on the original interpreter under breadth-first scheduling. Moreover, Prolog programs having the same input-output relations as the benchmark programs were written and timed. The Prolog system we used is DEC-10 Prolog on DEC2060.

Table 6.1 shows that our object codes ran 12 to 220 times as fast as the original interpreter. Moreover, they ran 2.7 to 4.4 times as fast as the comparable Prolog programs processed by the DEC-10 Prolog interpreter. They were, of course, slower than the comparable Prolog programs processed by the compiler, but the slowdown was 1/2.7 to 1/5.3, which we think is quite reasonable.

The '`append`' program ran at more than 11.5kRPS (kilo Reductions Per Second: equivalent to kLIPS if there are no guards).

Mode declaration was effective for all the benchmark programs. The speedup was 19% to 84%. As for the benchmark programs, the source of improvement is the declaration of the output mode. The speedup brought by changing bounded depth-first scheduling to depth-first scheduling was 27% or less.

The third program that performs bounded-buffer communication was inefficient, because process switching took place very often. We can see from Table 6.1 that we can make this program 2.75 times faster only by changing the buffer size to 10.

The column showing the number of suspensions indicates that the bounded depth-first scheduling provides good behavior except for bounded buffer programs. The ill behavior of the one-bounded buffer program is inevitable, because that behavior is just what the program has explicitly specified.

6.2.5. History and Possible Extensions

The Concurrent Prolog system explained above is the second version at ICOT. The first version, written by Chikayama [unpublished], optimized head unification,

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

:- mode append2(+,?,-).
append2([A|X],Y,[A|Z]) :- append2(X,Y,Z).
append2([],Y,Y).
```

(a) Concurrent Prolog source program

```
:-fastcode.
:-public append/8.
:-mode append(?,?,?, +,?,-,+,+).
append(Arg1,Y,Arg3, Bold,H,T,Flag,B0) :-
  ulist(Arg1,A1,X), ulist(Arg3,A2,Z),
  unify(A1,A2),                    % Head unification
  Bold>0, !,                       % Bound check
  Bnew is Bold-1,                  % and updating
  append(X,Y,Z, Bnew,H,T,nd,B0).   % Tail recursion

append(Arg1,Y1,Y2, Bold,H,T,Flag,B0) :-
  unil(Arg1), unify(Y1,Y2),        % Head unification
  Bold>0, !,                       % Bound check
  H=[$(Goal,H1,T,nd)|H1],          % Pop the next goal
  incore(Goal).                    % and call it

append(Arg1,Arg2,Arg3,
  Bold, [$(Goal,H1,T1,Flag)|H1],
  [$(append(Arg1,Arg2,Arg3, B0,H2,T2,Flag2,B0),
                               H2,T2,Flag2   )|T1],
  Flag,B0) :- incore(Goal).        % Suspension processing
```

(b) Object program in DEC-10 Prolog *(continued on the next page)*

Fig. 6.9. The effect of mode declaration

```
:-public append2/8.
:-mode append2(?,?,?, +,?,-,+,+).
append2(Arg1,Arg2,Arg3, Bold,H,T,Flag,B0) :-
  cpwait(Arg1,Arg1_w),              % Wait for Arg1
  Bold > 0, !,                      % Bound check
  '$$$append2'(Arg1_w,Arg2,Arg3,
               Bold,H,T,Flag,B0).   % Clause selection

append2(Arg1,Arg2,Arg3,
  Bold, [$(Goal,H1,T1,Flag)|H1],
  [$(append2(Arg1,Arg2,Arg3, B0,H2,T2,Flag2,B0),
                                 H2,T2,Flag2   )|T1],
  Flag,B0) :- incore(Goal).         % Suspension processing

'$$$append2'([A|X],Y,[A|Z],         % Unification for Arg1
                                    % and Arg3 is embedded
            Bold,H,T,Flag,B0) :- !,
  Bnew is Bold-1,                   % Bound updating
  append2(X,Y,Z, Bnew,H,T,nd,B0).   % Tail recursion

'$$$append2'([],Y,Y,                % Unification for Arg1
                                    % and Arg3 is embedded
            Bold,H,T,Flag,B0) :- !,
  H=[$(Goal,H1,T,nd)|H1],           % Pop the next goal
  incore(Goal).                     % and call it

'$$$append2'(Arg1,Arg2,Arg3,
  Bold, [$(Goal,H1,T1,Flag)|H1],
  [$('$$$append2'(Arg1,Arg2,Arg3,
                  B0,H2,T2,Flag2,B0),
                     H2,T2,Flag2   )|T1],
  Flag,B0) :- incore(Goal).         % Suspension processing
```

(b) Object program in DEC-10 Prolog *(continued from the previous page)*

Fig. 6.9. The effect of mode declaration *(continued)*

Table 6.1. Concurrent Prolog Benchmark on DEC2060

Program	Processing (*1)	Reductions	Suspensions	Time(*2)/RPS(*3) (compiler without mode)	(compiler with mode)	(interpreter)
List concatenation (Append) (500+0 elements)	B	502	0	—	—	2313 / 217
	BD100	502	0	88.7/ 5660	54.8/ 9160	—
	D	502	0	79.0/ 6350	43.0/11700	—
	P			15.8/31800	11.9/42200	188 /2670
Stream merge (100+100 elements)	B	202	0	—	—	1005 / 201
	BD100	202	0	42.9/ 4710	28.7/ 7040	—
	D	202	0	38.4/ 5260	23.6/ 8560	—
	P			8.3/24300	8.0/25300	73.7/2740
Bounded buffer (size=1)(*4)	B	204	0	—	—	1473 / 138
	BD100	204	200	147 / 1390	121 / 1690	—
	D	204	200	143 / 1430	119 / 1710	—
Bounded buffer (size=10)(*4)	B	204	0	—	—	1470 / 139
	BD100	204	20	60.2/ 3390	47.6/ 4290	—
	D	204	20	56.3/ 3620	43.3/ 4710	—
Primes (2 to 300) (without output)	B	2778	8445	—	—	80521 / 35
	BD100	2778	73	966 / 2880	769 / 3610	—
	D	2778	0	886 / 3140	689 / 4030	—
	P			216 /12900	188 /14800	2969 / 936
Quicksort (50 elements)	B	378	2225	—	—	20233 / 19
	BD100	378	0	125 / 3020	96.5/ 3920	—
	D	378	0	119 / 3180	91.3/ 4140	—
	P			21.3/17700	17.3/21800	246 /1540

*1 B—breadth-first scheduling; BD100—bounded depth-first scheduling (bound=100); D—depth-first scheduling; P—DEC-10 Prolog compiler (with 'fastcode' option) and interpreter.

*2 In milliseconds. Overhead for timing has been excluded:

*3 RPS—number of Reductions Per Second. An RPS value does not count reductions in guards. RPS values of Prolog programs were calculated using the number of reductions of the corresponding Concurrent Prolog programs.

*4 A Prolog counterpart does not exist.

but it still employed a centralized scheduler predicate which managed the goal queue. Therefore, the first version can be considered as a step from the original interpreter towards the second version. Although less efficient, the first version had an advantage that detailed trace information could be obtained more easily. This reflects the fact that the degree of compilation was smaller compared with the second version.

Before writing the second version, we made several mock-ups of object codes for simple programs and tested them. Design of object codes must be made very carefully because it is crucial for performance. A slight difference in object codes may greatly affect the performance if the codes have been highly optimized. After determining the object code format, it did not take much effort to write the compiler.

The GHC compiler was made by modifying the second version of the Concurrent Prolog compiler. Due to the similarity of Concurrent Prolog and GHC, all we had to do was to modify the code generator for guards and some runtime support routines including unification.

The restriction of the GHC compiler that no user-defined goals are allowed in a guard is not so severe as it might look, but it can be relaxed. There are two possible approaches to allowing user-defined goals in a guard: a static approach and a dynamic approach. In a static approach, one must analyze all user-defined goals in guards to determine whether each piece of unification can suspend or not. This approach is used in PARLOG for compile-time mode analysis (Clark and Gregory [1984c]). A dynamic approach is used in Miyazaki's compiler (Miyazaki [1985b]). This compiler makes use of the fact that in DEC-10 Prolog, a newer global variable has a larger address than older ones. This fact enables us to distinguish non-writable variables in a caller from writable ones newly created in a guard by using a 'threshold' address. Of course, it is inevitable that a dynamic approach loses efficiency in exchange for flexibility.

6.2.6. Summary

We have implemented fast, portable compilers of Concurrent Prolog and GHC on top of Prolog. If a Prolog system is available, one can immediately get started with parallel logic programming.

Both Concurrent Prolog and GHC systems are less than 800 lines long. It took only a few days to have the first working version of the Concurrent Prolog compiler. Modifying the Concurrent Prolog system to make the GHC system took one and a half days. In other methods, it would take much more efforts to make a system with the same efficiency. It is well known that Prolog is a good tool for rapid prototyping of another logic programming language, but all these facts show that an efficient Prolog implementation is a good tool also for getting an *efficient* implementation of another logic programming language rapidly.

Chapter 7
EXHAUSTIVE SEARCH IN GHC

This chapter presents a technique for compiling a Horn-clause program intended to be used for exhaustive search into a GHC (Guarded Horn Clauses) program. The technique can be viewed also as a transformation technique for Prolog programs which eliminates the '`bagof`' primitive and non-determinate bindings. The class of programs to which our technique is applicable is shown with a static checking algorithm; it is nontrivial and could be extended. An experiment on a compiler-based Prolog system showed that our compilation technique improved the efficiency of exhaustive search by 6 times in the case of a permutation generator program. This compilation technique is important also in that it exploits the AND-parallelism of GHC for parallel search. The contents of this chapter is based on (Ueda [1985c]).

7.1. Motivations

We often use Horn-clause logic, or more specifically the language Prolog, to obtain all solutions of some problem, that is, to obtain all answer substitutions for the variables in a goal to be solved. In this framework, however, it is difficult to *collect* the obtained solutions into a single environment to make further processing such as counting the number of the solutions, comparing them, classifying them, and so on. This is because these solutions correspond to different, independent paths of a search tree. For this reason, many of Prolog implementations support system predicates for creating a list of solutions of a goal given as an argument; examples are '`setof`' and '`bagof`' of DEC-10 Prolog. Naish [1985] made a survey of all-solutions predicates in various Prolog systems. These system predicates, however, internally use some extralogical features to record the obtained solutions. So it should be an interesting question whether it is possible to do exhaustive search without such primitives.

Another motivation is that we may sometimes wish to do exhaustive search in GHC or other parallel logic programming languages which do not directly support exhaustive search. In this case, parallelism inherent in GHC should be effectively used for the search.

One possible way to achieve the above requirements is to write down a first-order relation directly which states, for example, that "S is a list of all solutions of the N-queens problem". It is almost evident that such a relation can be described within the framework of Horn-clause logic. However, in practice, it is much harder to write it manually than to write a program which finds only one solution at a time. A programming tool which automatically generates an exhaustive search program could resolve this situation, and this is the way which we will pursue in this chapter.

7.2. Outlines of the Method

Our method is to compile a Horn-clause program intended to be used for exhaustive search by means of backtracking or OR-parallelism into a GHC program or a deterministic Prolog program which returns the same (multi-)set of solutions in the form of a single list. Here, the word 'deterministic' means that all bindings given to variables are determinate and never undone. Prolog programs in this subclass are interesting from the viewpoint of implementation, since a trail stack need not be prepared to execute them correctly. Furthermore, determinism in this sense has a similarity with the semantical restriction which GHC imposed to Horn clauses to make all activities done in a single environment. This similarity is reflected by the fact that a transformed program can be interpreted both as a GHC program and as a Prolog program by the slight change between the '`|`' (commitment) operator and the '`!`' (cut) operator.

There are two possible views of this transformation technique. One is to regard this as compilation from a Horn-clause program (with no concept of sequentiality) to a guarded-Horn-clause program. By compiling OR-parallelism into AND-parallelism, we eliminate a multiple environment mechanism which is in general necessary for parallel search since each path of a search tree would create its own binding environment. The other view is to regard it as transformation of a Prolog program. This transformation serves as simplification in the sense that the predicate '`bagof`' and the unbinding mechanism can be eliminated. Moreover, this transformation may remarkably improve the efficiency of a search program, as we will see later.

Our technique has another important meaning. By making search performed in a single environment, it becomes possible to introduce a mechanism for 'controlling' the search. That is, our technique may provide a starting point for more intelligent search.

A transformed program, viewed as a GHC program, emulates the OR-parallel and AND-sequential execution of the original program. The original OR-parallelism is compiled into AND-parallelism as stated above, and the sequential execution of conjunctive goals is realized by passing a continuation around. The AND-parallelism of GHC we use is a simple one, since two conjunctive goals solving different paths of a search tree have no interaction except when solutions are collected.

A continuation is a data structure which represents remaining tasks to be done before we get a solution. The notion of a continuation was effectively used also in Concurrent Prolog and GHC compilers on top of Prolog (Section 6.2) to implement a goal queue. The difference is that we use a stack instead of a queue here.

7.3. Previous Research

Implementation technique of exhaustive search in parallel logic programming languages can be found in (Hirakawa, Chikayama and Furukawa [1984]) and (Clark

and Gregory [1985c]). Their approach is to describe an interpreter of Horn-clause programs in Concurrent Prolog or PARLOG, but the following problems could be addressed to this approach:

(1) The interpreter approach loses efficiency.

(2) The multiple environment mechanism is implemented as a run-time creation of variants (Section 2.1.2) of terms.

Problem (1) will not be serious, since it could be resolved by a partial evaluation technique. Alternatively, we could directly write a compiler which corresponds to the original interpreter without much difficulty, as we did in Section 6.1. On the other hand, Problem (2) seems serious.

The reason why we need multiple environments is that different sets of unifiers can be generated when we rewrite a goal in two or more ways by using different program clauses at the same time (Section 3.3.1). Therefore, when we interpret an exhaustive search program, we make a necessary number of variants of the current set of goals and the partially determined solution prior to that simultaneous derivation. The above interpreters made some optimization to reduce the amount of variants to be created, but they did not avoid run-time creation of them.

However, run-time creation of variants is a time-sensitive operation. That is, a goal for creating a variant, say '`copy(`T_1`,`T_2`)`', cannot be rewritten to the conjunction of two goals 'T_1`=`T_3`, copy(`T_3`,`T_2`)`'. Hence the predicate '`copy`' is incompatible with the anti-substitutability of GHC (Section 4.7.2), and GHC cannot give any reasonable semantics to it. In the framework of sequential Prolog also, the predicate '`copy`' should be considered extralogical, because it cannot be defined without the extralogical predicate '`var`' which checks if its argument is currently an uninstantiated variable. The use of extralogical predicates should of course be discouraged, since it introduces semantical complexity and it hinders description of programming systems and support from them.

7.4. A Simple Example

To illustrate the difference between the previous method and ours, let us consider the example of decomposing a list using the '`append`' predicate:

```
:- append(U,     V,[1,2,3]).
   append([],    Z,Z      ).                                    (7.4–1)
   append([A|X],Y,[A|Z]   ) :- append(X,Y,Z).                   (7.4–2)
```

From the head of Clause (7.4–2), we get a partial solution `U=[1|X]`. Then we get three instances for `X`, namely `[]`, `[2]`, and `[2,3]`, by recursive calls. However, these three solutions cannot share the common prefix '`[1|`' as long as the value of a variable is represented by a reference pointer rather than by an association list, and this is why we have to make variants of the partial solution `[1|X]`.

Our method, on the other hand, first rewrites Clause (7.4–2) as follows:

```
append(X2    Y,[A|Z]  ) :- append(X,Y,Z), X2=[A|X].      (7.4–3)
```

The predicate '=' unifies its two arguments. It can be defined by a single unit clause:

```
X = X.
```

We assume that body goals are executed from left to right, following head unification. Then, while Clause (7.4–2) generates answer substitutions in a top-down manner, Clause (7.4–3) generates them in a bottom-up manner by combining ground terms. The first output argument `X2` remains uninstantiated until the first recursive goal, which may fork because of the two candidate clauses, succeeds. Therefore, we need not make variants of the partial solution upon the recursive call. Clause (7.4–3) is no more tail-recursive, so we must instead push the remaining task, the task of consing `A` with `X` to obtain `X2`, onto the stack representing a continuation. However, since the variable `A` has a ground value, the information to be stacked can be represented as a ground term and hence the continuation need not be copied even when the '`append`' goal forks.

Now we are prepared for the elimination of nondeterminism. Figure 7.1 shows a GHC program which returns the result equivalent (up to the permutation of solutions) to the following DEC-10 Prolog goal:

```
:- ..., bagof((X,Y), append(X,Y,Z), S), ... .
```

The search corresponding to the two clauses of the original '`append`' is performed by the conjunctive goals '`ap1`' and '`ap2`'. Their arguments are as follows:

(i) the input (third) argument of the original program,

(ii) the continuation,

(iii) the head of the difference list of solutions, and

(iv) its tail.

Since Clause (7.4–1) is a unit clause, the corresponding predicate '`ap1`' activates the 'remaining tasks' by calling the predicate '`cont`' for continuation processing. At that time, two output results, `[]` and the input argument itself, are passed to the continuation processing goal. The predicate '`ap2`' activates the first recursive goal with the information used by the second goal attached to the continuation in case the input argument has the form `[A|Z]`. If the input argument is not of the form `[A|Z]`, the unification of the input argument fails and the empty difference list is returned immediately.

The predicate '`cont`' does continuation processing. If the continuation has the form `'L1'(A,Cont)`, it pushes `A` in front of the output `X` and calls '`cont`' to process the rest of the continuation, `Cont`. If the continuation has the form `'L0'`,

Calling form: `:- ..., ap(Z,'L0',S,[]), ...`

```
ap(Z,Cont,S0,S2) :- true | ap1(Z,Cont,S0,S1), ap2(Z,Cont,S1,S2).

ap1(Z,Cont,S0,S1) :- true | cont(Cont,[],Z,S0,S1).

ap2([A|Z],Cont,S0,S1) :- true | ap(Z,'L1'(A,Cont),S0,S1).
ap2(Z,    _,   S0,S1) :- otherwise | S0=S1.

cont('L1'(A,Cont),X,Y,S0,S1) :- true | cont(Cont,[A|X],Y,S0,S1).
cont('L0',        X,Y,S0,S1) :- true | S0=[(X,Y)|S1].
```

Fig. 7.1. List Decomposition Program

it inserts the two outputs it received into the difference list. The function symbols which construct the continuation can be regarded as indicating the locations of the original program: `'L0'` indicates the end of the top-level goal and `'L1'` indicates the end of the recursive call of Clause (7.4–3). Interestingly, the predicate '`cont`' is very similar to an efficient (non-naive) list reversal program, and the continuation in this example is essentially a list which represents the first part of each solution (which is a pair of lists) in a reversed form. Different solutions to be collected are created by different calls of '`cont`' which reverse different substructures of the shared continuation.

The program in Figure 7.1 collects the solutions from '`ap1`' and '`ap2`' by the concatenation of difference lists, but this is not a fair way of collection. For example, if the first clause of some predicate produced infinite number of solutions, we could not see any solutions from the second clause. When we need a fair collection, we must collect solutions by using a '`merge`' predicate implemented fair.

We can interpret Figure 7.1 also as a Prolog program, provided that the '`|`' operators are replaced by the '`!`' operators, that the '`otherwise`' goal in the second clause of '`ap2`' is deleted, and that the second clause of '`ap2`' is guaranteed to be the last clause of '`ap2`'.

7.5. General Transformation Procedure

This section first presents the class of Horn-clause programs to which the technique as illustrated in Section 7.4 can be easily and mechanically applied, and then briefly shows the transformation procedure. We use the permutation program (Figure 7.2) as an example.

First of all, we show the class of Horn-clause programs to which our transformation technique is applicable. A program in this class must enjoy the following property when the body goals in each clause are executed from left to right, following head unification:

- The arguments of every goal appearing in a program can be classified into input arguments and output arguments. When some goal is called, its input arguments must have been instantiated to ground terms, and then the goal must instantiate its output arguments to ground terms when it succeeds.

Although the above property may look restrictive at a glance, most programs which do not use the notion of 'multiple writers' (see Section 7.6) or the notion of a difference list (which is an incomplete data structure) will enjoy this property. Programs which do use multiple writers require pre-transformation as described in Section 7.6. On the other hand, programs which make use of difference lists could be handled by extending the above notion of input and output, as long as they allow static dataflow analysis. This conjecture is based on the observation that when we write

```
perm([],    []).
perm([H|T], [A|P]) :- del([H|T], A, L), perm(L, P).

del([H|T], H, T).
del([H|T], L, [H|T2]) :- del(T, L, T2).
```

Fig. 7.2. Permutation Program

Given Declaration: `perm(+, -).` ('+': input, '-': output)

```
       +      -
perm( [],    []).
       +      -             +    -  -        +  -
perm([H|T], [A|P]) :- del([H|T], A, L), perm(L, P).

      +    -    -
del([H|T], H,   T).
      +    -    -            +  -  -
del([H|T], L, [H|T2]) :- del(T, L, T2).
```

Fig. 7.3. Mode Analysis of the Permutation Program

a Prolog program which handles difference lists, we usually fully recognize how uninstantiated variables appear in the data structures.

One way to give input/output modes to a program would be to make the programmer declare them for every goal arguments appearing in the program. However, a more realistic way will be to make the programmer declare the mode of (the arguments of) the top-level goal only and to 'infer' the modes of other goals according to the following rules:

• *Moding Policy for a Single Goal*

(a) Arguments which have been instantiated to ground terms upon call are regarded as input arguments (though they could be classified otherwise).

(b) All the other arguments are regarded as output arguments.

The mode inference and the check whether the program belongs to the above transformable class can be done in a simple static analysis. We must perform the following analysis for each clause and for each mode in which the predicate containing that clause may be called:

• *Mode Analysis of a Single Program Clause*

(1) Mark all the variables appearing in the input head arguments as *ground*.

(2) While there is a body goal yet to be analyzed, do the following repeatedly:

  (i) Determine the mode of the next body goal according to the above moding policy for a single goal. Here, those terms which are composed only of variables marked as *ground* and function symbols, and only those, are regarded as ground terms.

  (ii) Then mark all the variables appearing in the output arguments of that goal as *ground*.

(3) Check if the variables appearing in the output head arguments are all marked as *ground*. If the check succeeds, terminate the analysis of this clause with success; otherwise report failure.

Initially, only the modes of top-level goals are known; possible modes of other goals are incrementally obtained during the above analysis. Therefore, the whole algorithm of the mode analysis should be as follows. In the following, S denotes a set of 'moded' predicates. A moded predicate is a predicate with a mode in which it is called; different modes of a predicate correspond to different moded predicates.

• *Mode Analysis of an Entire Program*

(A) Let S be a set of the moded predicates whose calls appear in the (declared) top-level goal. Mark those predicates as *unanalyzed*.

(B) Repeatedly do the following until no *unanalyzed* predicate remains in S. That is, take an *unanalyzed* predicate from S, unmark it, and analyze all its clauses using the above algorithm, adding to S with the mark *unanalyzed* all moded predicates whose calls are newly found during the execution of Step (2) .

Figure 7.3 shows the analyzed permutation program. It is easy to prove, by induction on the number of steps of resolution, that a successfully analyzed program instantiates the output arguments of each goal to ground terms upon successful termination, provided ground terms are given to the input arguments.

A successfully analyzed program is then transformed according to the following steps:

(1) If there is any predicate to be called in two or more different modes, give a unique predicate name for each mode.

(2) Rewrite each clause into the normal form.

(3) Transform each predicate in the program.

Step (1) removes multi-mode predicates. This transformation attaches the concept of a mode to each *predicate* as well as to each predicate *call*.

Step (2) is made up of the following steps:

(2a) For each clause other than unit clauses, replace output head arguments $T_1, \ldots,$ T_n by distinct fresh variables $V_1, \ldots, V_n$, and place the goals V_1=$T_1, \ldots, V_n$=T_n at the end of the clause.

(2b) For each goal G in the body of each clause, replace its output arguments $T_1, \ldots, T_n$ by distinct fresh variables $V_1, \ldots, V_n$ and place the goals V_1=$T_1, \ldots,$ V_n=T_n immediately after G unless $T_1, \ldots, T_n$ are already distinct variables not appearing in the previous goals or the clause head.

The purpose of Step (2b) is to simplify output arguments in a clause head. It is clear that a program which has passed the mode analysis and then has been rewritten according to Steps (2a) and (2b) is still in the transformable class. Figure 7.4 shows the normal form of the permutation program.

Now we will show the outline of Step (3), the main part of our transformation method. Figure 7.5 shows the result applied to the permutation program of Figure 7.4. In the following, we indicate in braces what in the example of the permutation program are mentioned by each term appearing in the explanation.

(a) The arguments of a transformed predicate are made up of

- the input arguments of the original predicate,
- the continuation, and

```
perm([],   []).
perm([H|T],X) :- del([H|T],A,L), /*L1*/ perm(L,P), /*L2*/ X=[A|P].

del([H|T],H,T).
del([H|T],X,Y) :- del(T,L,T2), /*L3*/ X=L, Y=[H|T2].
```

Fig. 7.4. Normal Form of the Permutation Program

```
<1>  p([],   Cont,S0,S1) :- true | contp(Cont,[],S0,S1).
<2>  p([H|T],Cont,S0,S1) :- true | d([H|T],'L1'(Cont),S0,S1).
<3>  p(L,    _,   S0,S1) :- otherwise | S0=S1.

<4>  d(L,Cont,S0,S2) :- true | d1(L,Cont,S0,S1), d2(L,Cont,S1,S2).

<5>  d1([H|T],Cont,S0,S1) :- true | contd(Cont,H,T,S0,S1).
<6>  d1(L,    _,   S0,S1) :- otherwise | S0=S1.

<7>  d2([H|T],Cont,S0,S1) :- true | d(T,'L3'(H,Cont),S0,S1).
<8>  d2(L,    _,   S0,S1) :- otherwise | S0=S1.

<9>  contp('L2'(A,Cont),P,S0,S1) :- true | contp(Cont,[A|P],S0,S1).
<10> contp('L0',        P,S0,S1) :- true | S0=[P|S1].

<11> contd('L3'(H,Cont),L,T2,S0,S1) :- true |
         contd(Cont,L,[H|T2],S0,S1).
<12> contd('L1'(Cont),  A, L,S0,S1) :- true |
         p(L,'L2'(A,Cont),S0,S1).
```

Fig. 7.5. Transformed Permutation Program

- the head and the tail of the difference list for returning solutions.

Each transformed predicate is responsible for doing the task of the original predicate, followed by the task represented by the continuation.

(b) For a predicate {'`perm`'} of which at most one clause can be used for reducing each goal, the transformed predicate consists of the transformed clauses {`<1>, <2>`} of the original ones (See (i)). For a predicate {'`del`'} of which more than one clause may be applicable for reduction, we give a separate subpredicate name {'`d1`', '`d2`'} to each transformed clause {`<5>, <7>`}, and let the transformed predicate {'`d`'} call all these subpredicates and collect solutions.

(c) The body of a clause {`<1>, <5>`} transformed from a unit clause calls a goal for continuation processing {'`contp`', '`contd`'}. This goal is given as arguments the output values {`[]`, `(H,T)`} returned by the original unit clause.

(d) The body of a clause {`<2>, <7>`} transformed from a non-unit clause calls the predicate {'`d`'} corresponding to the first body goal {'`del`'} of the original clause (See (e) and (j)).

(e) When calling a (transformed) predicate {e.g., '`d`' in `<7>`} corresponding to the i-th body goal G_i {the recursive call of '`del`'} of some clause, we push the label {`'L3'`} indicating the next goal G_{i+1} together with the input data {`H`} used by the subsequent goals $G_{i+1}, \ldots, G_n$ {`X=L, Y=[H|T2]`}. When calling a predicate {'`p`'} corresponding to the top-level goal {say '`perm(`L`,X)`' where L is some ground term}, we give as the initial value of the continuation the label {`'L0'`} indicating the termination of refutation together with the data {*none*} necessary for constructing a term to be collected {`X`}.

(f) Predicates for continuation processing are composed of clauses {`<9>, <10>, <11>, <12>`} each corresponding to the label pushed in Step (e). These clauses are classified according to the predicates immediately before those labels and are given separate predicate names {'`contp`', '`contd`'}.

(g) Each clause {e.g., `<12>`} of a predicate for continuation processing makes input data {`L`} for the next goal {`perm(L,P)`} indicated by the received label {`'L1'`}, by using the information {*none*} stacked with the label and the output {`A`, `L`} of the last goal. Then it calls a predicate {'`p`'} corresponding to the next goal (See (e) and (j)).

(h) The clause {`<10>`} for processing the label {`'L0'`} indicating termination generates a term to be collected {`P`} from the output {`P`} of the top-level goal and the information {*none*} stacked with the label, and returns a difference list having that term as a sole element.

(i) For those transformed predicates {'`p`', '`d1`', '`d2`'} which may fail in the unification of the input arguments, backup clauses {`<3>, <6>, <8>`} are generated which return empty difference lists when the unification fails.

(j) In spite of the above rules, no transformed predicates are generated for '=' and other system predicates; they are processed immediately 'on the spot', followed by the next task {<9>, <11>}.

It is worth noting that in spite of our restriction, a transformed program can handle some non-ground data structure correctly. That is, the portions of an input data structure which are only passed around and never examined by unification need not be ground terms. For example, when we execute the following goal,

```
:- p([A,B,C], 'L0', S, []).
```

S will be correctly instantiated to a list of six permutations:

```
[[A,B,C],[A,C,B],[B,A,C],[B,C,A],[C,A,B],[C,B,A]].
```

7.6. On the Class of Transformable Programs

For the technique described above to be useful from the practical point of view, the transformable class of Horn-clause programs defined in Section 7.5 must be large enough to express our problems naturally. The problem in this regard is that we often make use of the notion of 'multiple writers'. By 'multiple writers' we mean two or more goals sharing some data structure and trying to instantiate it cooperatively and/or competitively. In Prolog programming, such a data structure is usually represented directly by a Prolog term and it is operated by the direct use of Prolog unification; a typical example is the construction of the output data of a parser program.

However, this programming technique has problems from the viewpoint of the applicability of our transformation:

(1) It is generally impossible to analyze statically which part of the shared data structure is instantiated by which goal.

(2) The shared data structure may not be instantiated fully to a ground term.

Item (2) is considered a problem also from a semantical point of view. When extracting some information from the shared data structure generated by a search program, we have to use the extralogical predicate 'var' to see whether some portion of the data structure is left undetermined. One may argue that we need not use the predicate 'var' if we analyze the data structure after making it ground, that is, after instantiating its undetermined portions to some ground terms such as new constant symbols. He may further argue that making a term ground never calls for the predicate 'var' since we can accomplish this by trying to unify every subterm of it with a new constant. However, then, the search program which generates a non-ground result and the program to make it ground will be in the relationship of multiple writers, and the latter program should never start before the former program has finished because the latter program must have a lower priority with

respect to instantiation of the shared data structure. This means we have to use the concept of sequentiality or priority between conjunctive goals, both of which are concepts outside pure Horn-clause logic.

Anyway, we must make some pre-transformation to such a Horn-clause program in order to apply our transformation technique. That is, we must change the representation of the shared data structure to a ground-term representation—a list of binding information generated by each writer. Each writer must receive the current list of binding information and return a new one as a separate argument. When a writer is to add some binding information, it must check the consistency of the current and the new information to be added. This checking could be done by trying to construct the original representation from scratch each time, but it could be done more efficiently by adopting an appropriate data structure (possibly other than a list of bindings) for the binding information.

Comparing the original and the proposed implementation schemes of multiple writers from a practical point of view, the proposed scheme is apparently disadvantageous in the ease of programming. However, the difference does not lie in the specification of the abstract data but only in the ease of its implementation, which should not be so essential a problem since accumulation of programming techniques and program libraries should alleviate the difficulty.

Efficiency is another point on which comparison should be made. Although the original representation is suitable for the execution using backtracking, it requires a multiple environment mechanism for OR-parallel execution, which may cause additional complexity and overhead (Ciepielewski and Haridi [1983]; Ciepielewski [1984]). The proposed pre-transformation may make the consistency checking somewhat expensive, but will make parallel execution much easier because no multiple environment mechanism is necessary.

7.7. Performance Evaluation

Table 7.1 compares the performance of original and transformed programs. The programs measured are those described above, and an N-queens program with N being 5, 6, 7 and 8. The N-queens program we used was in the transformable class shown in Section 7.5.

All programs were measured using DEC-10 Prolog on DEC2065. For each original program, the execution time of exhaustive search (by forced backtracking) without any collection of solutions was measured as well as the execution time by the '`bagof`' primitive. The '`setof`' primitive was not considered because the sorting of solutions was inessential for us. Each program was measured after possible simplification which took advantage of the fact that Prolog checks candidate clauses sequentially.

As Table 7.1 shows, the proposed program transformation improved the efficiency of exhaustive search by 6 times for the permutation program and by more

Table 7.1. Performance of Exhaustive Search Programs (in msec.)

Program	Original (`bagof`)	Original (*search only*)	Transformed	Number Of Solutions
List Decomposition (50 elements)	836	4	27	51
Permutation Generation (5 elements)	354	34	57	120
5-Queens	45	20	28	10
6-Queens	90	75	106	4
7-Queens	441	325	446	40
8-Queens	1796	1484	1964	92

than 30 times for the list decomposition program '`append`'. This remarkable speedup was brought about by specializing the task of collecting solutions to fit within the framework of Horn-clause logic, while the '`bagof`' primitive uses a extralogical feature similar to '`assert`' which an optimizing compiler cannot help. A program such as N-queens, which has only a small number of solutions compared with its search space, cannot therefore expect remarkable speedup; the transformed N-queens program got slightly slower except for the case of 5-queens. After some manual optimization, however, the transformed 8-queens program surpassed the original '`bagof`' version.

Another important point to note is that in the case of 8-queens, the transformed program was only by 25% slower than the original program which does not collect solutions and which makes use of the dedicated mechanism for search problems: automatic backtracking. This suggests that the transformed program could not be improved very much without changing the search algorithm.

7.8. Summary and Future Works

We have described a method of compiling a Horn-clause program for exhaustive search into a GHC program or a deterministic Prolog program. Although not stated above, the method using the concept of a continuation can be applied also to the case where only one solution is required. Our method also provides the possibility of introducing control into search, since all activities are made to be performed in a single environment.

We restricted the class of Horn-clause programs to which our method is applicable. However, this class is never trivial and it is expected that we should not have so much difficulty in writing a program within this class or its natural extension. Rather, we believe that it is important from a practical point of view to show the class of Horn-clause programs which can be transformed without loss of efficiency and without resort to extralogical predicates.

The loss of performance by not using such dedicated mechanisms as automatic backtracking was small. Conversely, we found that our technique may greatly improve the efficiency of exhaustive search that has been done by using the '`bagof`' primitive.

The proposed transformation eases parallel search in that it eliminates the need of multiple environments, but it never eliminates other problems on resource management. Resource management is still an important problem for realizing parallel search. Therefore, our results need not and should not be interpreted as reducing the significance of OR-parallel Prolog machines: Specialized hardware can always perform better for a special class of programs. While our purpose was primarily to examine the possibility of efficient search on a general-purpose parallel machine, we do expect also that our technique will be utilized for improving the

efficiency of OR-parallel Prolog machines. Comparison of these two approaches should be an interesting research in the near future.

Chapter 8
CONCLUSIONS AND FUTURE WORKS

8.1. Contributions and Implications

The main contribution of this thesis is that it has presented a simple, basic framework of a programming language in which we can describe programs interacting with outside worlds. The framework is based on logic programming, but we made nontrivial extensions to make it a programming language. Any practical programming language must have some means to interact with outside worlds. The original logic programming framework regards an answer substitution or a set of answer substitutions from all possible refutations as the result of computation. However, a set of answer substitutions is a meta-level concept which is obtained by *observing* the proof. On the other hand, input and output of GHC are done by *participating* the proof. I/O processes, treated completely within the relational framework, can be viewed as modeling the outside worlds. Thus the framework of GHC is suitable for treating both computers and their surroundings uniformly; we can interchangeably use a human process and an automated process to perform the same task. There are many other good reasons to do without meta-level concepts. Absence of meta-level concepts keeps the language and the underlying theory simple. The simplicity may lead to efficient implementation also.

Including outside worlds in a uniform framework naturally called for parallelism, regardless of whether the computer is parallel or sequential. It also called for the notion of determinate bindings and causality among bindings. On the other hand, it did not call for sequentiality as in Prolog. The design of GHC reflects all these considerations.

Although we amended the original framework of logic programming, the framework greatly contributed to the simplicity of the resulting language. In particular, it enabled us to introduce an elegant notion of information flow and synchronization. It is hoped that the simplicity and the generality of GHC will contribute much to the future research on parallel programming and parallel programming languages.

However, one may feel still uneasy about the practical aspects of GHC, and it was actually the case. We had to steadily remove the anxiety, and Chapters 6 and 7 addressed three important issues: Communication in a large network of processes, efficient implementation on a sequential machine, and facilities for exhaustive search which were once discarded.

The results of Chapters 6 and 7 remind us of the general principle that a program need not be executed in a way it appears to specify. In *any* programming languages, what a program specifies is ultimately nothing more than an input-output relation. It never suggests how the result *must* be computed, though it

may suggest how it *can* be computed. Chapters 6 and 7 show how compilation is important for programming languages which are not machine-oriented.

A compilation technology serves to raise the level of a programming language with minimum loss of efficiency. Of course, this enlarges the semantic gap between the language and a target machine as long as the computer architecture remains the same. The correctness of a compiler may become harder to prove. However, a compiler must be made only once, and also its correctness must be proved only once. After that, the description and the verification of all user programs are simplified. This is the very benefit of a higher-level language.

8.2. Applications of GHC

GHC is a general-purpose language; it is intended to be used for any applications. Application programs written so far include a formula manipulation program, a window system, and a hardware simulator. In addition, many programs written in Concurrent Prolog and PARLOG can be easily translated into GHC; see Hellerstein and Shapiro [1984], Broda and Gregory [1984], Edelman and Shapiro [1984], Gregory, Neely and Ringwood [1985] and Matsumoto [1986]. Takeuchi [1986] wrote an algorithmic debugger of GHC in GHC (see Section 8.3.3). As shown in Chapter 7, Ueda [1985c] used GHC as a target language of exhaustive search programs written in ordinary Horn clauses. However, much more programs must be written before GHC is widely accepted as a practical programming language. This process is important also for identifying necessary syntactic constructs to be incorporated into a user language (Section 8.3.2), as well as for developing efficient implementations.

Real-time systems form an important application area. However, the computational model provided by GHC is too liberal to permit description of real-time systems; for example, it allows delay in communication using shared variables. To correctly handle real time in GHC, it will be necessary to introduce the notion of time and to impose appropriate restrictions on the operational semantics.

8.3. Future Works

8.3.1. Toward More Formal Semantics

Although the semantics of GHC in Section 4.4 is not given formally, it is described with much care. The simplicity of the language and the use of familiar and stable concepts such as unification contribute to the clarity of the informal description. Clear informal semantics is valuable as it is.

Nevertheless, we need a more formal semantics, of course. Formal semantics gives a foundation for correct implementation of the language, for mechanical and manual handling of programs such as partial evaluation, and for reasoning about programs. A declarative semantics of logic programs described in Section 2.1.4 is

of limited use because of the additional construct of GHC. Moreover, it seems hard to introduce causality in that declarative framework. A more promising approach should be to generalize the formal semantics of non-deterministic dataflow languages (Brock and Ackerman [1981])(Brock [1983])(Staples and Nguyen [1985]).

We must also develop a detailed operational semantics. An operational semantics shows guidelines for implementation algorithmically. It is especially important in parallel languages, since it constructs the semantics of a program by means of primitive operations. To this end, Ueda [1986] analyzes the operational aspects of anti-substitutability.

8.3.2. User Language

Although GHC as described in Chapter 4 provides no syntactic sugars, the author believes that a GHC program is fairly readable. A program with complex data flow is not easy to understand, of course, but it is mainly due to the complexity of the program itself.

However, we often use GHC as a process description language with process interpretation in mind. Since process interpretation is a kind of pragmatics, an appropriate syntactic support by a user language might be desirable. It should be an interesting work to design a process-oriented user language which can clearly express flexible data flow of GHC. It should also be interesting to design modularization facilities based on the notion of processes.

A much more conservative feature that could be incorporated into a user language is a notation for anonymous predicates. The current syntax of GHC forces a programmer to invent a new predicate name for each conditional branching, as in the following program appearing in Section 4.6.3.

```
filter(P,Xs,[Y|Ys1]) :- true | Xs=[X|Xs1], filter2(P,X,Xs1,Y,Ys1).
filter(P,Xs,[]     ) :- true | Xs=[].
filter2(P,X,Xs1,Y,Ys1) :- X mod P=:=0 |      filter(P,Xs1,[Y|Ys1]).
filter2(P,X,Xs1,Y,Ys1) :- X mod P=\=0 | Y=X, filter(P,Xs1,Ys1    ).
```

By eliminating the auxiliary predicate '`filter2`' and restricting the clause head to the most general form, we get the following representation:

```
filter(P, Xs, Ys) :-
  ( Ys=[Y|Ys1] |
      Xs=[X|Xs1],
      ( X mod P=:=0 |      filter(P, Xs1, Ys ) ;
        X mod P=\=0 | Y=X, filter(P, Xs1, Ys1)
      ) ;
    Ys=[]       | Xs=[]
  ).
```

Thus anonymous predicates look like guarded commands, and the program looks more procedural. Note, however, that we have used nothing more than a syntactic sugar, though the above representation seems inappropriate under Open World Assumption. Syntactic sugars of the same kind would include the notation similar to the `case` statement of Pascal for multiway branching according to the value of some variable. Syntactic conventions of this kind will be added when they are called for in writing programs.

8.3.3. Programming System and Metalevel Facilities

It is often said that a parallel program is hard to debug. Probably, a program with a simple process structure and simple dataflow is tractable enough, but it may be hard to analyze the behavior of a program with complex dataflow. Takeuchi [1986] developed a debugger of GHC based on the divide-and-query method of Shapiro [1982]. Techniques developed in the context of functional programming could also be incorporated. For example, Takahashi, Ono and Amamiya [1985] improved the efficiency of the divide-and-query debugging by taking the static structure of a program into account. Techniques for tracing or spying a parallel program must also be developed. Tracing and spying will be useful for analyzing an erroneous situation in a large, complex program rather than for identifying a bug. They could be used also for monitoring resource allocation and efficiency.

A programming system that automatically proves or helps us prove the correctness of a program, and a system that automatically derives or helps us derive a correct and efficient program are important also. For these purposes, a good specification language well suited to the basic concept of GHC should be valuable.

It seems best to write these programming systems in GHC itself, though we have not finished the design of the set of features for system programming. A system program must fulfill the following requirements:

(1) A system program must be able to communicate with a user program.

(2) A system program must be able to observe the execution of a user program; at least it must be able to detect termination of a user program.

(3) A system program must have necessary control over a user program.

(4) A system program must never be killed by a user program.

However, the previous proposals on system programming in parallel logic programming languages fail to satisfy the above requirements. A Unix-like shell by Shapiro [1984] does not satisfy (1); Requirement (1) claims that we cannot use a guard to guarantee (4). The two- (and three-) argument metacall by Clark and Gregory [1984b] does not satisfy (4) under anti-substitutability as we explained in Section 4.7.2. Further investigation is necessary to have a better solution. Introducing some notions that distinguish between object- and meta-levels will be inevitable, since

observation and control are inherently meta-level notions. The problem is how to introduce them.

8.3.4. Implementation

Besides the implementation described in Section 6.2, a compiler-based sequential implementation (Esaki, Miyazaki and Dasai [1986]) has been made on PSI (Taki, Yokota, Yamamoto, Nakajima and Mitsuishi [1984]). This implementation is intended to be modified into a distributed implementation on multi-PSI, a multiprocessor system composed of 4 to 6 PSI's. Another implementation effort is under way on VAX11-780, using the instruction set like Warren's [1983]. A sample object program for '`append`' marked more than 30 kRPS (kilo Reductions Per Second) in an idealized condition. These implementations do not allow user-defined goals in guards.

However, the speed of 30 kRPS for '`append`' on a machine of around 1 MIPS may still be unsatisfactory, since this figure means only 30,000 message receivings and 30,000 message sendings per second in process interpretation. We considered in Section 6.1 optimization techniques of stream merging, stream distribution and mutable arrays, but we did not go so far as to optimize the representation of streams. It is necessary to consider time- and space-efficient implementation of important data structures including streams, character strings, and large mutable objects such as databases.

8.3.5. Theoretical Issues of Parallel Computation

It is hoped that the simplicity and the generality of GHC stimulate theoretical research on the underlying parallel computational model. Also, it is important to find an appropriate cost criterion on which to discuss computational complexity, though it may be implementation dependent. Many theoretical results on sequential computation or those depending on sequential computation must be re-examined to adapt them to the framework of parallel computation. This re-examination may help in separating those points essential for any manner of computation from those specific to sequential computation, thus contributing also to sequential computation.

We have seen in the history of Lisp and Prolog that a 'pure' language becomes widely used only after impure extensions, and that even after such extensions only its pure subset attracts attention of theoretical persons. Extensions ignored by theorists tend to be intolerably dirty. We should try to minimize the discrepancy of the theoretical and the practical versions of a language, and a simple and powerful language should promote this direction. A good programming language should well mediate among theorists, application programmers, and implementors. GHC is the first step from Prolog along this direction; we have provided a framework in which we can handle input and output, and we have introduced a minimal control structure necessary to guide the computation in the right direction. However, it

still excludes some important concepts including those on control such as fairness of nondeterministic choice, though it would be very easy to introduce them in an irresponsible way. At the next step, we must consider the unsolved problems for the better approximation to the ultimate language.

REFERENCES

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. [1974] *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass.

Aida, H. [1984] A Proposal of a Logic Programming Language Suited for Parallel Processing. In *Proc. 10th WGSF Meeting, IPS Japan* (in Japanese).

Amamiya, M. [1984] On Dataflow Architecture. *Computer Software*, Vol. 1, No. 1, pp. 42–63 (in Japanese).

Amamiya, M., Hasegawa, R. and Mikami, H. [1983] List Processing with a Data Flow Machine. In *Proc. RIMS Symposia on Software Science and Engineering*, Goto, E., Furukawa, K., Nakajima, R., Nakata, I. and Yonezawa, A. (eds.), Lecture Notes in Computer Science 147, Springer-Verlag, Berlin Heidelberg, pp. 165–190.

Aoyagi, T., Fujita, M. and Moto-oka, T. [1985] Temporal Logic Programming Language Tokio—Programming in Tokio. *The Logic Programming Conf. '85*, Institute for New Generation Computer Technology, Tokyo, pp. 185–196. Also to appear in *Logic Programming '85*, Wada, E. (ed.), Lecture Notes in Computer Science 221, Springer-Verlag, Berlin Heidelberg (1986).

Apt, K. R. and Emden, M. H. van [1982] Contributions to the Theory of Logic Programming. *J. ACM*, Vol. 29, No. 3, pp. 841–862.

Backus, J. [1978] Can Programming Be Liberated from the von Neumann Style? A functional Style and Its Algebra of Programs. *Comm. ACM*, Vol. 21, No. 8, pp. 613–641.

Bowen, D. L. (ed.), Byrd, L., Pereira, F. C. N., Pereira, L. M. and Warren, D. H. D. [1983] *DECsystem-10 Prolog User's Manual*. Dept. of Artificial Intelligence, Univ. of Edinburgh.

Bowen, K. A. [1985] *Meta-Level Programming and Knowledge Representation*. Logic Programming Research Group, School of Computer and Information Science, Syracuse Univ.

Bowen, K. A. and Weinberg, T. [1985] A Meta-Level Extension of Prolog. In *Proc. 1985 Symp. on Logic Programming*, IEEE Computer Society, pp. 48–53.

Boyer, R. S. and Moore, J. S. [1972] The Sharing of Structure in Theorem-Proving Programs. In *Machine Intelligence 7*, Edinburgh University Press.

Brock, J. D. [1983] *A Formal Model of Non-determinate Dataflow Computation*. Tech. Report TR-309, Laboratory for Computer Science, Massachusetts Institute of Technology.

Brock, J. D. and Ackerman, W. B. [1981] Scenarios: A model of non-determinate

computation. In *Formalization of Programming Concepts*, Diaz, J. and Ramos, I. (eds.), Lecture Notes in Computer Science 107, Springer-Verlag, Berlin Heidelberg, pp. 252–259.

Broda, K. and Gregory, S. [1984] PARLOG for Discrete Event Simulation. In *Proc. Second Int. Logic Programming Conf.*, Uppsala Univ., Sweden, pp. 301–312.

Bruynooghe, M. [1982] Adding Redundancy to Obtain More Reliable and More Readable Prolog Programs. In *Proc. First Int. Logic Programming Conf.*, Faculté des Sciences de Luminy, Marseille, pp. 129–133.

Burstall R. M., MacQueen, D. B. and Sannella, D. T. [1980] HOPE: an Experimental Applicative Language. In *Proc. 1980 Lisp Conf.*, pp. 136–143.

Chang, C. -L. and Lee, R. C. -T. [1973] *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York.

Chikayama, T. [1984] Unique Features of ESP. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, Institute for New Generation Computer Technology, Tokyo, pp. 292–298.

Ciepielewski, A. [1984] *Towards A Computer Architecture for OR-Parallel Execution of Logic Programs.* TRITA-CS-8401, Dept. of Telecommunication Systems—Computer Systems, The Royal Institute of Technology, Stockholm.

Ciepielewski, A. and Haridi, S. [1983] A Formal Model for OR-Parallel Execution of Logic Programs. In *Proc. IFIP '83*, Mason, R. E. A. (ed.), Elsevier Science Publishers B. V., Amsterdam, pp. 299–305.

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

Clark, K. L. [1979] *Predicate Logic as a Computational Formalism.* Research Monograph 79/59 TOC, Dept. of Computing, Imperial College of Science and Technology, London.

Clark, K. L. and Gregory, S. [1981] A Relational Language for Parallel Programming. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, ACM, pp. 171–178.

Clark, K. L. and Gregory, S. [1983] *PARLOG: A Parallel Logic Programming Language.* Research Report DOC 83/5, Dept. of Computing, Imperial College of Science and Technology, London.

Clark, K. L. and Gregory, S. [1984a] *PARLOG: Parallel Programming in Logic.* Research Report DOC 84/4, Dept. of Computing, Imperial College of Science and Technology, London.

Clark, K. L. and Gregory, S. [1984b] Notes on Systems Programming in PARLOG, In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, Institute for

New Generation Computer Technology, Tokyo, pp. 299–306.

Clark, K. L. and Gregory, S. [1984c] *Notes on the Implementation of PARLOG.* Research Report DOC 84/16, Dept. of Computing, Imperial College of Science and Technology, London, 1984. Also in *J. of Logic Programming*, Vol. 2, No. 1 (1985), pp. 17–42.

Clark, K. L. and McCabe, F. G. [1980] IC-PROLOG—Language Features. In *Proc. Logic Programming Workshop*, Tärnlund, S. -Å. (ed.), Debrecen, Hungary, pp. 45–52. Also in *Logic Programming*, Clark, K. and Tärnlund, S. -Å. (eds.), Academic Press, London (1982), pp. 253–266.

Clark, K. L. and Tärnlund, S. -Å. [1977] A First Order Theory of Data and Programs. In *Proc. IFIP '77*, Gilchrist, B. (ed.), North-Holland, Amsterdam, pp. 393–944.

Clinger, W. D. [1981] *Foundations of Actor Semantics.* AI-TR-633, Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Colmerauer, A. [1982] *Prolog II Reference Manual and Theoretical Model.* Internal report, Groupe Intelligence Artificielle, Université Aix-Marseille II.

Cohen, S [1984] Multi-Version Structures in Prolog. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, Institute for New Generation Computer Technology, Tokyo, pp. 265–274.

Conery, J. S. [1983] *The AND/OR Process Model for Parallel Interpretation of Logic Programs.* Ph. D. Thesis, Tech. Report 204, Univ. California, Irvine.

Conery, J. S. and Kibler D. F. [1981] Parallel Interpretation of Logic Programs. In *Proc. 1981 Symp. on Functional Programming Languages and Computer Architecture*, ACM, pp. 163–170.

Conery, J. S. and Kibler D. F. [1985] AND Parallelism and Nondeterminism in Logic Programs. *New Generation Computing*, Vol. 3, No. 1, pp. 43–70.

DeGroot, D. [1984] Restricted AND-Parallelism. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, Institute for New Generation Computer Technology, Tokyo, pp. 471–478.

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

Dömölki, B. and Szeredi, P. [1983] Prolog in Practice. In *Proc. IFIP '83*, Mason, R. E. A. (ed.), Elsevier Science Publishers B. V., Amsterdam, pp. 627–636.

Dwork, D., Kanellakis, P. C. and Mitchell, J. C. [1984] On the Sequential Nature of Unification. *J. Logic Programming*, Vol. 1, No. 1, pp. 35–50.

Edelman, S. and Shapiro E. [1984] *Quadtrees in Concurrent Prolog.* Tech. Report CS84-09, Dept. of Applied Math., The Weizmann Institute for Science, Israel.

Emden, M. H. van and Kowalski, R. A. [1976] The Semantics of Predicate Logic as a Programming Language. *J. ACM*, Vol. 23, No. 4, pp. 733–742.

Emden, M. H. van and Lucena, G. J. de [1982] Predicate Logic as a Language for Parallel Programming. In *Logic Programming*, Clark, K. L. and Tärnlund, S. -Å. (eds.), Academic Press, London, pp. 189–198.

Eriksson, L. -H. and Rayner, M. [1984] Incorporating Mutable Arrays into Logic Programming. In *Proc. Second Int. Logic Programming Conf.*, Uppsala Univ., Sweden, pp. 101–114.

Esaki, R., Miyazaki, T. and Dasai, T. [1986] Sequential Implementation of GHC Subset. To appear in *Proc. 32nd Annual Convention IPS Japan*, 2G-4 (in Japanese).

Furukawa, K. and Ueda, K. [1985] GHC Process Fusion by Program Transformation. In *Proc. Second National Conf. of Japan Society of Software Science and Technology*, pp. 89–92.

Gallaire H. and Minker, J. (eds.) [1978] *Logic and Data Bases.* Plenum Press, New York.

Gelernter, D. [1984] A Note on Systems Programming in Concurrent Prolog. In *Proc. 1984 Int. Symp. on Logic Programming*, IEEE Computer Society, pp. 76–82.

Goldberg, A. and Robson, D. [1983] *Smalltalk-80—The Language and Its Implementation.* Addison-Wesley, Reading, Mass.

Goto, A., Tanaka, H. and Moto-oka T. [1984] Highly Parallel Inference Engine PIE—Goal Rewriting Model and Machine Architecture—. *New Generation Computing*, Vol. 2, No. 1, pp. 37–58.

Gregory, S. [1984] *How to Use PARLOG (C-Prolog version).* Dept. of Computing, Imperial College, London.

Gregory, S. [1985a] *private communication.*

Gregory, S. [1985b] *Design, Application and Implementation of a Parallel Logic Programming Language.* Ph. D. thesis, Dept. of Computing, Imperial College of Science and Technology, London.

Gregory, S., Neely, R. and Ringwood, G. [1985] *PARLOG for Specification, Verification and Simulation.* Research Report DOC 85/7, Dept. of Computing, Imperial College of Science and Technology, London. Also in *Proc. 7th Int. Symp. on Computer Hardware Description Languages and their Applications*, Tokyo.

Hagiya, M. [1983] On Lazy Unification and Infinite Trees. In *Proc. Logic Programming Conf. '83*, Institute for New Generation Computer Technology, Tokyo (in

Japanese).

Hagiya, M. [1984a] Foundation of Prolog. *J. IPS Japan*, Vol. 25, No. 12, pp. 1336–1344 (in Japanese).

Hagiya, M. [1984b] Theory of Modal Logic Programming. In *Proc. 9th WGSF Meeting*, IPS Japan (in Japanese).

Hellerstein, L. and Shapiro, E. [1984] Implementing Parallel Algorithms in Concurrent Prolog: The MAXFLOW Example. In *Proc. 1984 Int. Symp. on Logic Programming*, IEEE Computer Society (1984), pp. 99–115.

Hewitt, C. [1977] Viewing Control Structures as Patters of Passing Messages. *Artificial Intelligence*, Vol. 8, pp. 323–364.

Hewitt, C. and Baker, H. [1977] Laws for Communicating Parallel Processes. In *Proc. IFIP '77*, Gilchrist, B. (ed.), North-Holland, Amsterdam, pp. 987–992.

Hirakawa, H., Chikayama, T. and Furukawa, K. [1984] Eager and Lazy Enumerations in Concurrent Prolog. In *Proc. Second Int. Logic Programming Conf.*, Uppsala Univ., Sweden, pp. 89–100.

Hirata, M. [1984] Operational Semantics of Pure Concurrent Prolog. In *Proc. First Conf. of Japan Society of Software Science and Technology*, pp. 255–258.

Hirata, M. [1985] Self-Description of Oc and Its Applications. In *Proc. Second National Conf. of Japan Society of Software Science and Technology*, pp.153–156 (in Japanese).

Hoare, C. A. R. [1969] An Axiomatic Basis of Computer Programming. *Comm. ACM*, Vol. 12, No. 10, pp. 576–583.

Hoare, C. A. R. [1978] Communicating Sequential Processes. *Comm. ACM*, Vol. 21, No. 8, pp. 666–677.

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

Hogger, C. J. [1984] *Introduction to Logic Programming*. Academic Press, London.

Ito, N. and Masuda, K. [1984] Parallel Inference Machine Based on the Data Flow Model. In *Int. Workshop on High Level Computer Architecture 84*, pp. 4.31–4.40.

Ito, N., Shimizu, H., Kishi, M., Kuno, E. and Rokusawa, K. [1985] Data-flow Based Execution Mechanisms of Parallel and Concurrent Prolog. *New Generation Computing*, Vol. 3, No. 1, pp. 15–41.

Kahn, G. [1974] The Semantics of a Simple Language for Parallel Programming. In *Proc. IFIP '74*, North-Holland, Amsterdam London, pp. 471–475.

Kahn, G. and Macqueen, D. B. [1977] Coroutines and Networks of Parallel Processes. In *Proc. IFIP '77*, Gilchrist, B. (ed.), North-Holland, Amsterdam, pp. 993–998.

Kanada, Y. [1985] High-speed Execution of Prolog on Supercomputers. In *Proc. 26th Programming Symp.*, IPS Japan, pp. 47–55 (in Japanese).

Keller, R. M. [1982] FEL—An Experimental Applicative Language. In *Proc. 3rd WGSF Meeting*, IPS Japan.

Knuth, D. E. [1968] *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley, Reading, Mass.

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

Kowalski, R. [1985a] Directions for Logic Programming. In *Proc. 1985 Symp. on Logic Programming*, pp. 2–7.

Kowalski, R. [1985b] Logic Programming. *Byte*, Vol. 10, No. 8, pp. 161–177.

Kusalik, A. J. [1984] Bounded-Wait Merge in Shapiro's Concurrent Prolog. *New Generation Computing*, Vol. 2, No. 2, pp. 157–169.

Kusalik, A. J. [1984] *On Unification of Read-only Terms in Concurrent Prolog*. Discussion paper, Computer Science Dept., Univ. British Columbia, Canada.

Levy, J. [1985] *personal communication*.

Lloyd, J. W. [1984] *Foundations of Logic Programming*. Springer-Verlag, Berlin Heidelberg New York Tokyo.

Martelli, A. and Montanari, U. [1982] An Efficient Unification Algorithm. *ACM Trans. Prog. Lang. Syst.*, Vol. 4, No. 2, pp. 258–282.

Matsuda, H., Tamura, N., Kohata, M., Kaneda, Y. and Maekawa, S. [1985] Implementing Parallel Prolog System "K-Prolog". *Trans. IPS Japan*, Vol. 26, No. 2, pp. 296–303 (in Japanese).

Matsumoto, Y. [1986] A Parallel Parsing System for Natural Language Analysis. To be presented at *the Third Int. Conf. on Logic Programming*, London.

Mierowsky, C., Taylor, S., Shapiro. E., Levy, J. and Safra, M. [1985] *The Design and Implementation of Flat Concurrent Prolog*. Technical Report CS85-09, Dept. of Applied Mathematics, The Weizmann Institute of Science, Israel.

Miyachi, T., Kunifuji, S., Kitakami, H., Furukawa, K., Takeuchi, A. and Yokota, H. [1984] A Knowledge Assimilation Method For Logic Databases. In *Proc. 1984 Int. Symp. on Logic Programming*, IEEE Computer Society, pp. 118–125. Also in *New Generation Computing*, Vol. 2, No. 4 (1984), pp. 385–404.

Miyazaki, T. [1985a] *unpublished manuscript.* Institute for New Generation Computer Technology, Tokyo (in Japanese).

Miyazaki, T. [1985b] *GHC-to-Prolog compiler* (unpublished program).

Miyazaki, T., Takeuchi A. and Chikayama T. [1985] A Sequential Implementation of Concurrent Prolog Based on the Shallow Binding Scheme. In *Proc. 1985 Symp. on Logic Programming*, pp. 110–118.

Moto-oka, T., Tanaka, H., Aida, H., Hirata, K. and Maruyama, T. [1984] The Architecture of a Parallel Inference Engine —PIE—. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, Institute for New Generation Computer Technology, Tokyo, pp. 479–488.

Naish, L. [1985] All Solutions Predicates in Prolog. In *Proc. 1985 Symp. on Logic Programming*, IEEE Computer Society, pp. 73–77.

Nakagawa, H. [1984] AND-Parallel Prolog with Divided Assertion Set. In *Proc. 1984 Int. Symp. on Logic Programming*, IEEE Computer Society, pp. 22–28.

Nakashima, H. [1983a] *A Knowledge Representation System: Prolog/KR.* Tech. Report METR 83-5, Dept. of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, Univ. of Tokyo.

Nakashima, H. [1983b] *Prolog.* Sangyo-Tosho, Tokyo (in Japanese).

Nakashima, H., Ueda, K. and Tomura, S. [1983] Applicative Input-Output and String Manipulation facilities in Logic Programming Languages. *Trans. IPS Japan*, Vol. 24, No. 6, pp. 745–753 (in Japanese).

Nakashima, H., Ueda, K. and Tomura, S. [1984] What Is a Variable in Prolog? In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, Institute for New Generation Computer Technology, Tokyo, pp. 327–332.

Nitta, K. [1984] Parallel Prolog. *J. IPS Japan*, Vol. 25, No. 12, pp. 1353–1359 (in Japanese).

Nitta, K., Matsumoto, Y. and Furukawa, K. [1983] A Description of Prolog Interpreter and Its Parallel Extension. *Trans. Institute of Electronics and Communication Engineers of Japan*, Vol. J66-D, No. 11, pp. 1310–1317 (in Japanese).

Onai, R., Aso, M., Shimizu, H., Masuda, K. and Matsumoto, A. [1985] Architecture of a Reduction-Based Parallel Inference Machine: PIM-R. *New Generation Computing*, Vol. 3, No. 2, pp. 197–228.

Pereira, L. M. [1982] Logic Control with Logic. In *Proc. First Int. Logic Programming Conf.* Faculté des Sciences de Luminy, Marseille, pp. 9–18. Also in *Implementation of Prolog*, Campbell, J. A. (ed.), Wiley, New York (1984), pp. 177–193.

Pereira, L. M. and Nasr, R. [1984] Delta-Prolog: A Distributed Logic Programming Language. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, Institute for New Generation Computer Technology, Tokyo, pp. 283–291.

Plaisted, D. A. [1984] The Occur Check Problem in Prolog. In *Proc. 1984 Int. Symp. on Logic Programming*, IEEE Computer Society, pp. 272–280. Also in *New Generation Computing*, Vol. 2, No. 4 (1984), pp. 309–322.

Porto, A. [1982] Epilog: A Language for Extended Programming in Logic. In *Proc. First Int. Logic Programming Conf.*, Faculté des Sciences de Luminy, Marseille, pp. 9–18. Also in *Implementation of Prolog*, Campbell, J. A. (ed.), Wiley, New York (1984), pp. 268–278.

Robinson, J. A. [1965] A Machine-Oriented Logic Based on Resolution Principle. *J. ACM*, Vol. 12, No. 1, pp. 23–41.

Roussel, P. [1975] *Prolog: Manual de Reference et d'Utilisation.* Groupe d'Intelligence Artificielle, Marseille-Luminy.

Sakuragawa, Y. [1985] Temporal Prolog. To appear in *RIMS Kokyuroku*, Kyoto Univ.

Saraswat, V. A. [1985] *Problems with Concurrent Prolog (preliminary version).* Carnegie-Mellon Univ.

Sato, M. and Sakurai, T. [1984] Qute: A Functional Language Based on Unification. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, Institute for New Generation Computer Technology, Tokyo, pp. 157–165.

Sergot, M. J. [1983] A Query-the-User Facility for Logic Programs. In *Integrated Interactive Computer Systems*, Degano, P. and Sandewall, E. (eds.), North-Holland, Amsterdam New York Oxford, pp. 27–41.

Shapiro, E. Y. [1982] *Algorithmic Program Debugging.* MIT Press, Cambridge, Mass.

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

Shapiro, E. Y. [1983b] *Notes on Sequential Implementation of Concurrent Prolog: Summary of Discussions in ICOT*, Institute for New Generation Computer Technology, Tokyo (unpublished).

Shapiro, E. Y. [1984] Systems Programming in Concurrent Prolog. In *Conf. Record of the 11th Annual ACM Symp on Principles of Programming Languages*, ACM, pp. 93–105.

Shapiro, E. and Mierowsky, C. [1984] Fair, Biased, and Self-Balancing Merge Operators: Their Specification and Implementation in Concurrent Prolog. In *Proc. 1984 Int. Symp. on Logic Programming*, IEEE Computer Society, pp. 83–90.

Shapiro, E. and Safra, M. [1985] *Fast Multiway Merge Using Destructive Operations*. Tech. Report CS85-01, Dept. of Applied Mathematics, The Weizmann Institute of Science, Israel.

Shapiro, E. and Takeuchi, A. [1983] Object Oriented Programming in Concurrent Prolog. *New Generation Computing*, Vol. 1, No. 1, pp. 25–48.

Staples, J. and Nguyen, V. L. [1985] A Fixpoint Semantics for Nondeterministic Data Flow. *J. ACM*, Vol. 32, No. 2, pp. 411–444.

Takahashi, N., Ono, S. and Amamiya, M. [1985] A Bug-Location Method for Functional Programs Using Function Dependency Graph Transformation. In *Proc. 12th WGSF Meeting*, IPS Japan (in Japanese).

Takeuchi, A. [1986] Algorithmic Debugging of GHC Programs. In *Proc. 32nd Annual Convention IPS Japan*, 2G-5 (in Japanese). Also to appear as ICOT Technical Report, Institute for New Generation Computer Technology, Tokyo.

Takeuchi, A. and Furukawa, K. [1983] Interprocess Communication in Concurrent Prolog. In *Proc. Logic Programming Workshop '83*, Universidade Nova de Lisboa, Portugal.

Takeuchi, A. and Furukawa, K. [1985] Bounded Buffer Communication in Concurrent Prolog. *New Generation Computing*, Vol. 3, No. 2, pp. 145–155.

Taki, K., Yokota, M., Yamamoto, A., Nakashima, H. and Mitsuishi, A. [1984] Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI). In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, Institute for New Generation Computer Technology, Tokyo, pp. 398–409.

Tanaka, J., Miyazaki, T. and Takeuchi, A. [1984] Sequential Implementation of Concurrent Prolog—Copying Approach to Multiple Environments. *First National Conf. of Japan Society of Software Science and Technology*, pp. 303–306 (in Japanese).

Taylor, S., Lowry, A., Maguire, G. Q. Jr. and Stolfo, S. J. [1984] Logic Programming Using Parallel Associative Operations. In *1984 Int. Symp. on Logic Programming*, IEEE Computer Society, pp. 58–68.

Tick, E. and Warren, D. H. D. [1984] Towards a Pipelined Prolog Processor. In *Proc. 1984 Int. Symp. on Logic Programming*, IEEE Computer Society, pp. 29–40. Also in *New Generation Computing*, Vol. 2, No. 4 (1984), pp. 323–345.

Turner, D. A. [1981] The Semantic Elegance of Applicative Languages. In *Proc. 1981 Conf. on Functional Programming Languages and Computer Architecture*, ACM, pp. 85–92.

Ueda, K. [1982] *Comments on Names and Expressions in Ada*. Tech. Report METR 82-6, Dept. of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, Univ. of Tokyo.

Ueda, K. [1984] *Comments on the Concurrent Prolog Language Rules, Parts 1-3* (unpublished manuscript).

Ueda, K. [1985a] *Concurrent Prolog Re-Examined.* ICOT Tech. Report TR-102, Institute for New Generation Computer Technology.

Ueda, K. [1985b] *Guarded Horn Clauses.* ICOT Tech. Report TR-103, Institute for New Generation Computer Technology. Also to appear in *Logic Programming '85*, Wada, E. (ed.), Lecture Notes in Computer Science, Springer-Verlag, Berlin Heidelberg (1986), pp. 168-179.

Ueda, K. [1985c] Making Exhaustive Search Programs Deterministic. In *Proc. Second National Conf. of Japan Society of Software Science and Technology*, pp. 145–148 (in Japanese). Also to appear as ICOT Tech. Report TR-145, Institute for New Generation Computer Technology, Tokyo. Also to be presented at the *Third Int. Logic Programming Conf.*, London (1986).

Ueda, K. [1986] *On the Operational Semantics of Guarded Horn Clauses.* To appear as ICOT Tech. Memorandum, Institute for New Generation Computer Technology, Tokyo.

Ueda, K. and Chikayama, T. [1984a] Efficient Stream/Array Processing in Logic Programming Languages. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, Institute for New Generation Computer Technology, Tokyo, pp. 317–326.

Ueda, K. and Chikayama, T. [1984b] Practical Implementation of a Parallel Logic Programming Languages. In *Proc. First National Conf. of Japan Society of Software Science and Technology*, pp. 307–310 (in Japanese).

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

Wada, E., Tomura, S., Nakashima, H. and Kimura, M. [1982] Programming in Prolog. In *Conf. Record of the 23rd Programming Symp.*, IPS Japan, pp. 122–132 (in Japanese).

Warren, D. H. [1977] *Implementing PROLOG—Compiling Predicate Logic Programs, Vol. 1–2.* D. A. I. Research Report No. 39, Dept. of Artificial Intelligence, Univ. of Edinburgh.

Warren, D. H. [1980] An Improved Prolog Implementation Which Optimises Tail Recursion. In *Proc. Logic Programming Workshop*, Tärnlund, S. -Å. (ed.), Debrecen, Hungary, pp. 1–11.

Warren, D. H. D. [1983] *An Abstract Prolog Instruction Set.* Tech. Note 309, Artificial Intelligence Center, SRI International, CA.

Warren D. H. D, Pereira L. M. and Pereira F. [1977] PROLOG—The Language and Its Implementation Compared with Lisp. *Sigplan Notices*, Vol. 12, No. 8, pp. 109–115.

Warren, D. S. [1984] Efficient Prolog Memory Management for Flexible Control Strategies. In *Proc. 1984 Int. Symp. on Logic Programming*, IEEE Computer Society, pp. 198–202. Also in *New Generation Computing*, Vol. 2, No. 4 (1984), pp. 361–369.

Warren, D. S., Ahamad, M., Debray, S. K. and Kalé, L. V. [1984] Executing Distributed Prolog Programs on a Broadcast Network. In *Proc. 1984 Int. Conf. on Logic Programming*, IEEE Computer Society, pp. 12–21.

Weinreb, D. and Moon, D. [1981] *Flavors: Message Passing in the LISP Machine*. AI Memo No. 602, MIT Artificial Intelligence Lab.

Yasuura, H. [1984] On Parallel Computation Complexity of Unification. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1984*, Institute for New Generation Computer Technology, pp. 235–243.

Yonezaki, N., Atarashi, A. and Hourai, N. [1985] Temporal Logic Programming Based on Time Interval. In *Proc. 39th WGAI Meeting*, IPS Japan (in Japanese).

Yonezawa, A. [1979] A Tutorial on ACTOR Theory. *J. IPS Japan*, Vol. 20, No. 7, pp. 580–589 (in Japanese).

Yonezawa, A. [1984] On Object-Oriented Programming. *Computer Software*, Vol. 1, No. 1, pp. 29–41 (in Japanese).