2

WORKING MATERIAL

FOR THE LECTURES OF

EHUD Y. SHAPIRO

THE FAMILY OF CONCURRENT LOGIC PROGRAMMING LANGUAGES

INTERNATIONAL SUMMER SCHOOL

ON

LOGIC, ALGEBRA AND COMPUTATION

MARKTOBERDORF, GERMANY, JULY 25 - AUGUST 6, 1989

THIS SUMMER SCHOOL IS ORGANIZED UNDER THE AUSPICES OF THE TECHNISCHE UNIVERSITÄT MÜNCHEN AND IS SPONSORED BY THE NATO SCIENCE COMMITEE AS PART OF THE 1989 ADVANCED STUDY INSTITITES PROGRAMME. PARTIAL SUPPORT FOR THE CONFERENCE WAS PROVIDED BY THE EUROPEAN RESEARCH OFFICE AND THE NATIONAL SCIENCE FOUNDATION AND BY VARIOUS INDUSTRIAL COMPANIES.

The Family of Concurrent Logic Programming Languages

E. Shapiro

CS89–08

May 1989

Department of Applied Mathematics & Computer Science
The Weizmann Institute of Science
Rehovot 76100, Israel

The Family of Concurrent Logic Programming Languages

Ehud Shapiro
Department of Applied Mathematics and Computer Science
The Weizmann Institute of Science
Rehovot 76100, Israel

April, 1989

Abstract

Concurrent logic languages are high-level programming languages for parallel and distributed systems that offer a wide range of both known and novel concurrent programming techniques. Being logic programming languages, they preserve many advantages of the abstract logic programming model, including the logical reading of programs and computations, the convenience of representing data-structures with logical terms and manipulating them using unification, and the amenability to meta-programming. Operationally, their model of computation consists of a dynamic set of concurrent processes, communicating by instantiating shared logical variables, synchronising by waiting for variables to be instantiated, and making nondeterministic choices, possibly based on the availability of values of variables.

This paper surveys the family of concurrent logic programming languages within a uniform operational framework. It demonstrates the expressive power of even the simplest language in the family, and investigates how varying the basic synchronisation and control constructs affect the expressiveness and efficiency of the resulting languages.

In addition, the paper reports on techniques for sequential and parallel implementation of languages in this family, mentions their applications to date, and relates these languages to the abstract logic programming model, to the programming language Prolog, and to other concurrent computational models and programming languages. (KR) ←

Contents

PART I. INTRODUCTION

1. Introduction

In surveying concurrent logic programming languages, this paper:

- Introduces the computational models of logic programs, Prolog, and concurrent logic languages.
- Discusses the different role of nondeterminism in these three computational models.
- Explains the use of the logical variable as a communication channel, and the use of unification in the specification and implementation of sophisticated communication protocols.
- Demonstrates the powerful programming techniques available in concurrent logic languages, including: stream processing, the formation and manipulation of dynamic process networks, incomplete-message protocols for dialogues and network configuration, concurrent construction of shared data-structures, and short-circuit protocols for distributed termination and quiescence detection.
- Demonstrates the utility of enhanced meta-interpreters in concurrent logic programming, including their application to computation control, to the formation of live and frozen snapshots, and to computation replay and debugging.
- Exposes the spectrum of concurrent logic programming languages, ranging from the simpler and weaker ones, to the more complex and more expressive ones.
- Reports on implementation techniques for sequential and parallel computers developed for concurrent logic language, as well as on specialized architectures designed for them.

The paper does not aim at providing a historical account of the development of concurrent logic languages. Rather, it attempts to expose the core concepts of these languages, as well as the internal structure of the family and the qualities of each of its members, within a consistent operational framework. As a result, usually an idealized or a simplified version of each language is described. When applicable, the differences from the actual language, as well as relevant historical facts, are noted[1].

The paper consists of five parts. In the remainder of Part I, Section 2 surveys briefly the abstract computational model of logic programming and of (pure) Prolog, explaining the role of the logical variable, unification, and nondeterminism in this model.

Part II conveys the core concepts and techniques of concurrent logic programming. Section 3 introduces the basic concepts of concurrent logic programming, and the use of shared logical variables for communication and synchronization. Section 4 defines a simple concurrent logic language. This language is used in Section 5 to illustrate basic concurrent logic programming examples and techniques. Section 6 discusses fairness conditions for concurrent logic programs. Following that, Section 7 describes advanced concurrent logic programming techniques. Although this part uses a particular concurrent logic language, both the basic and advanced techniques shown are common to most programming languages in the family; exceptions are noted when each language is introduced.

Part III surveys the various members of the family of concurrent logic languages. Section 21 describes our method of comparing languages in the family. We compare languages for their expressiveness, simplicity, readability and efficiency. In comparing expressiveness, we explore embeddings among languages and the programming techniques provided by each language.

Section 9 discusses the semantics of concurrent logic programs. Sections 10 to 13 introduce and compare flat concurrent logic languages. A flat language is defined with respect to a given fixed set of primitive predicates (in the languages discussed these include mainly equality, inequality and arithmetic tests). In a flat language a process can perform only a simple computation, specified

[1] For additional historical notes see [145,164].

by a conjunction of atoms with primitive predicates, before making a committed nondeterministic choice. In non-flat languages such pre-commit computations may involve program-defined predicates, thus can be arbitrarily complex. During a computation of a non-flat language the processes form an And/Or-tree, whereas in a flat language the processes are a "flat" collection; hence their name. Non-flat concurrent logic languages are surveyed in Section 18.

Part IV describes implementations developed for concurrent logic languages, and references their applications. Implementation techniques for both sequential and parallel computers are reviewed, as well as specialized architectures designed for their efficient execution.

Part V concludes the paper by comparing the concurrent logic programming model with other approaches to programming and modeling concurrency, including Prolog, dataflow languages, functional languages, message-passing models of concurrency, object-oriented languages, and nondeterministic transition systems.

How to read the paper

The reader who wishes only to understand a single concurrent logic language can skim Part I and read Part II. There are sufficient intuitive explanations and examples so that the formal treatment of the semantics of logic programs can be skipped without loss of continuity. The reader interested in implementation techniques can read Section 20 of Part IV without reading Part III.

2. Logic Programming, Prolog, and the Power of the Logical Variable

This section introduced the logic programming computational model. It defines pure Prolog and relates it to the logic programming model. It discusses properties of the logical variable and unification and their relation to conventional data-manipulation operations.

2.1 Syntax and informal semantics of logic programs

We use the Edinburgh syntax [11] for logical variables, terms, and predicates.

Definitions: Term, atom, clause, logic program, vocabulary.

- A *term* is a variable (e.g. X) or a function symbol of arity $n \geq 0$, applied to n terms (e.g. c and $f(a,X,g(b,Y))$).
- An *atom* is a formula of the form $p(T_1,\ldots,T_n)$, where p is a predicate of arity n and $T_1,\ldots,T_n$ are terms.
- A *definite clause* (*clause* for short) is a formula of the form:

  $$A \leftarrow B_1,\ldots,B_n. \qquad n \geq 0.$$

  where A is an atom $B_1,\ldots,B_n$ is a sequence of atoms. A is called the clause's *head*, and $B_1,\ldots,B_n$ its *body*. We denote the empty sequence of atoms by *true*.
- A *logic program* is a finite set of definite clauses.
- A *goal* is a sequence of atoms $A_1,A_2,\ldots,A_n$. A goal is *empty* if $n=0$, *atomic* if $n=1$, and *conjunctive* if $n>1$. Each atom in a goal is called a *goal atom*. A goal atom is often called also a *goal* for short.
- The *vocabulary* of a logic program P is the set of predicates and function symbols that occur in the clauses of P. ∎

We use the Edinburgh notation for lists (and also for streams, as discussed below). The term $[X|Xs]$ (read "X *cons* Xs") is a list whose head is X and tail is Xs, and the constant $[\,]$ (read "*nil*") is used by convention to denote the empty list.

Informal semantics of logic programs

Logic programs can be read both declaratively and operationally. We describe these two views here informally, and make them precise in Section 2.4 below.

Declaratively, each clause in a logic program is read as a universally quantified implication. If $X_1,X_2,\ldots,X_n$ are the variables in the clause $A \leftarrow B_1,B_2,\ldots,B_k$, then the clause is read "for all $X_1,X_2,\ldots,X_n$, A is true if B_1 and B_2 and ... and B_k are true". A logic program is read as the conjunction of the universal implications corresponding to its clauses.

Operationally, logic programs can be viewed as an abstract computational model, like the Turing Machine, the Lambda Calculus, and the Random Access Machine. A computation in this model is a goal-driven deduction from the clauses of the program. Like the nondeterministic Turing machine, computations in this model are nondeterministic: from each state of the computation there may be several possible transitions. Specifically, the clauses of a logic program can be read as transition rules of a nondeterministic transition system.

The state of a computation consists of a goal (sequence of atoms) G and a substitution (assignment of values to variables) θ, and is denoted by a pair $\langle G;\theta\rangle$. A computation begins with an initial state consisting of the initial goal to be proven and the empty substitution ϵ, and progresses nondeterministically from state to state according to the following transition rules, Reduce and Fail. A computation can be viewed as an attempt to prove the initial goal from the program. At each state the goal represents a statement whose proof will establish the initial goal; the substitution represents the values computed so far for variables used in the computation, including the initial goal variables. A computation ends in a state whose goal is either *true* or *fail*. In the former case the computation is ***successful***, and it corresponds to a successful proof of the initial goal. In the latter it is ***failed***. The substitution in the terminal state, restricted to the variables in the initial goal, is called the ***answer substitution*** of the computation.

A successful computation has the property that its initial goal, instantiated by the answer substitution, is a logical consequence of the program.

A key step in the transitions is the unification of a goal atom with the head of a clause. Intuitively, a unifier of two terms T_1 and T_2 is a substitution θ, whose application to T_1 and T_2 yields the same term, i.e. $T_1\theta = T_2\theta$. The unification of two terms T_1 and T_2 returns their most general ("simplest") unifier θ if there is one, or *fail* if there is none. The two cases are denoted by $mgu(T_1,T_2)=\theta$ and $mgu(T_1,T_2)=fail$, respectively. For example, the most general unifier of $f(X,b)$ and $f(g(Y),Z)$ is the substitution $\{X \mapsto g(Y), Z \mapsto b\}$. Examples of other (less general) unifiers are $\{X \mapsto g(a), Z \mapsto b\}$, $\{X \mapsto g(b), Z \mapsto b\}$, and $\{X \mapsto g(g(W)), Z \mapsto b\}$.

We denote the ability to move from a state S to a state S' using a transition rule t by $S \xrightarrow{t} S'$. Substitutions can be viewed as functions from variables to values (see Section 2.4), hence we use $\theta \circ \theta'$ to denote the substitution whose application has the effect of applying θ then applying θ'. The Reduce and Fail transition rules require that the variables in the clause be consistently replaced by new variables that have not been used before in the computation. A clause to which this replacement has been applied is called *renamed apart*. The requirement to rename a clause is inherited from the resolution rule, and ensures that clauses are "re-entrant".

There are two transition rules:

1. Reduce
   $\langle A_1,\ldots,A_i,\ldots,A_n;\theta\rangle \xrightarrow{\text{Reduce}} \langle (A_1,\ldots,B_1,\ldots,B_k,\ldots,A_n)\theta';\theta \circ \theta'\rangle$
   If $mgu(A_i,A) = \theta'$ for some renamed apart clause $A \leftarrow B_1,\ldots,B_k$ of P.
2. Fail
   $\langle A_1,\ldots,A_i,\ldots,A_n;\theta\rangle \xrightarrow{\text{Fail}} \langle fail;\theta\rangle$
   If for some i and for every renamed apart clause $A \leftarrow B_1,\ldots,B_k$ of P, $mgu(A_i,A) = fail$.

Reduce has the following property: If $\langle G,\theta\rangle \xrightarrow{\text{Reduce}} \langle G',\theta \circ \theta'\rangle$, with mgu θ', then $G\theta'$ is a logical consequence of the program and G'. This implies, by induction, that the initial goal, to which the answer substitution of a successful computation is applied, is a logical consequence of the program.

Note that there are two types of nondeterministic choices in the Reduce transition: which goal atom to reduce, and which clause to reduce it with. The first is called ***And-nondeterminism***, the second ***Or-nondeterminism***. Fail has only an And-nondeterministic choice.

A computation progresses until it reaches a ***terminal state***, which is a state to which no transition applies. By the definition of Reduce and Fail, the goal in a terminal state is either ***true*** or ***fail***.

2.2 Examples of logic programs and their computations

We show some simple logic programs and illustrate their operational behavior. The following logic program defines the predicate *sum(Xs,S)*, which holds if *S* is the sum of the elements of the list *Xs*.

```
sum(Xs,S) ←                 %1
    sum'(Xs,0,S).
sum'([ ],S,S).              %2
sum'([X|Xs],P,S) ←          %3
    plus(X,P,P')
    sum'(Xs,P',S).
```

The program uses an auxiliary predicate *sum'(Xs,P,S)*, which holds if the sum of the *Xs* plus *P* is *S*, and the predicate *plus(X,Y,Z)*, which holds if *X* plus *Y* is *Z*. For the purpose of this example we assume that *plus* is defined by a large set of facts, including:

```
plus(0,0,0).                %4
plus(0,1,1).                %5
plus(1,0,1).                %6
plus(2,0,2).                %7
plus(2,1,3).                %8
plus(2,2,4).                %9
  ⋮
```

To increase readability of the following examples of computations we annotate Reduce transitior with a label (i,j) identifying the indices of goal atom and program clause that were used for reduction, Fail transitions with the index of the failing goal atom, and restrict the substitution in a state to the initial goal variables.

An example of a successful computation of the above program is:

$$\langle \text{sum}([1,2],S);\, \varepsilon\rangle \xrightarrow{\text{Reduce}(1,1)}$$

$$\langle \text{sum}'([1,2],0,S);\, \varepsilon\rangle \xrightarrow{\text{Reduce}(1,3)}$$

$$\langle \text{plus}(1,0,P),\ \text{sum}'([2],P,S);\, \varepsilon\rangle \xrightarrow{\text{Reduce}(1,6)}$$

$$\langle \text{sum}'([2],1,S);\, \varepsilon\rangle \xrightarrow{\text{Reduce}(1,3)}$$

$$\langle \text{plus}(2,1,P'),\ \text{sum}'([\ ],P',S);\, \varepsilon\rangle \xrightarrow{\text{Reduce}(1,8)}$$

$$\langle \text{sum}'([\ ],3,S);\, \varepsilon\rangle \xrightarrow{\text{Reduce}(1,2)}$$

$$\langle \mathit{true};\, \{S \mapsto 3\}\rangle$$

An example of a failing computation is:

$$\langle \text{sum}([1,2],S);\, \varepsilon\rangle \xrightarrow{\text{Reduce}(1,1)}$$

$\langle \mathrm{sum}'([1,2],0,S);\ \varepsilon\rangle \xrightarrow{\text{Reduce}(1,3)}$

$\langle \mathrm{plus}(1,0,P),\ \mathrm{sum}'([2],P,S);\ \varepsilon\rangle \xrightarrow{\text{Reduce}(2,3)}$

$\langle \mathrm{plus}(1,0,P),\ \mathrm{plus}(2,P,P'),\ \mathrm{sum}'([\,],P',S);\ \varepsilon\rangle \xrightarrow{\text{Reduce}(2,9)}$

$\langle \mathrm{plus}(1,0,2),\ \mathrm{sum}'([\,],4,S);\ \varepsilon\rangle \xrightarrow{\text{Fail}(1)}$

$\langle \mathit{fail},\ \varepsilon\rangle$

The failure in the last computation could have been avoided by deferring the reduction of the goal atom *plus*(*2,P,P'*) until more information was available. The Reduce transition of Prolog, introduced in Section 2.5 below, always chooses the leftmost atom in the goal for reduction. Thus a Prolog computation on an initial goal *sum*(*Xs,S*) whose first argument is a complete list[2] of integers and the second argument is a variable is bound to succeed. Furthermore, such a computation is deterministic, in the sense that at each step only one clause head unifies with the selected goal atom. Concurrent logic languages use other mechanisms to delay the reduction of a goal atom, which do not impose such strict sequentiality.

The following logic program defines the relation *in_both*(*X,L1,L2*), which holds if *X* is a member of both lists *L1* and *L2*. It uses the auxiliary predicate *member*(*X,Xs*), which holds if *X* is a member of the list *Xs*.

% *in_both*(X,L_1,L_2) ← *X* is a member of both lists L_1 and L_2.

```
    in_both(X,L1,L2) ←                    %1
        member(X,L1), member(X,L2).
```

% *member*(*X,Xs*) ← *X* is a member of the list *Xs*.

```
    member(X,[X|Xs]).                     %2
    member(X,[X1|Xs]) ←                   %3
        member(X,Xs).
```

Here are two possible computations from the goal *in_both*(*X*,[*a,b*],[*b,c*]). A failing computation, in which *X* is chosen to be *a*, and the computation of the remaining goal *member*(*a*,[*b,c*]) fails:

$\langle \mathit{in\_both}(X,[a,b],[b,c]) \cdot \varepsilon\rangle \xrightarrow{\text{Reduce}(1,1)}$

$\langle \mathit{member}(X,[a,b]),\ \mathit{member}(X,[b,c])\ ;\ \varepsilon\rangle \xrightarrow{\text{Reduce}(1,2)}$

$\langle \mathit{member}(a,[b,c])\ ;\ \{X \mapsto a\}\rangle \xrightarrow{\text{Reduce}(1,3)}$

$\langle \mathit{member}(a,[c])\ ;\ \{X \mapsto a\}\rangle \xrightarrow{\text{Reduce}(1,3)}$

$\langle \mathit{member}(a,[\,])\ ;\ \{X \mapsto a\}\rangle \xrightarrow{\text{Fail}(1)}$

$\langle \mathit{fail}\ ;\ \{X \mapsto a\}\rangle$

A successful computation from the same goal, in which *X* is chosen to be *b*:

$\langle \mathit{in\_both}(X,[a,b],[b,c])\ ;\ \varepsilon\rangle \xrightarrow{\text{Reduce}(1,1)}$

$\langle \mathit{member}(X,[a,b]),\ \mathit{member}(X,[b,c])\ ;\ \varepsilon\rangle \xrightarrow{\text{Reduce}(1,3)}$

[2] A list is *complete* if every instance of it is a list [171]; [*a,b*], [*X,b*], and [*X, Y*] are complete lists, and [*a,b*|*Xs*], [*a*|*Xs*], [*X*|*Xs*] and *Xs* are examples of incomplete lists [171].

$\langle member(X,[b]),\ member(X,[b,c])\ ;\ \varepsilon\rangle \xrightarrow{\text{Reduce}(1,2)}$

$\langle member(b,[b,c])\ ;\ \{X\mapsto b\}\rangle \xrightarrow{\text{Reduce}(1,2)}$

$\langle true\ ;\ \{X\mapsto b\}\rangle$

For this program, no ordering of goal atoms can make the computation deterministic on an initial goal atom whose first argument is a variable.

The following logic program uses the difference-list technique for efficient list concatenation, so we digress to explain it. A *difference-list* is a term representing a list as the difference between two (possibly incomplete) lists. By convention, the term $H\backslash T$ is used, where '\' is a binary function symbol written in infix notation; H is called the *head* and T the *tail* of the difference-list. Examples of difference-lists representing the list $[a,b,c]$ are $[a,b,c]\backslash[\ ]$, $[a,b,c,d,e],\backslash[d,e]$, and $[a,b,c|Xs]\backslash Xs$. Given two difference-lists $H_1\backslash T_1$ and $H_2\backslash T_2$, if $T_1=H_2$ then $H_1\backslash T_2$ is their concatenation. It is easy to see that the list represented by $H_1\backslash T_2$ is the concatenation of the lists represented by $H_1\backslash T_1$ and $H_2\backslash T_2$. For example, the concatenation of $[a,b,c,d,e]\backslash[d,e]$ and $[d,e]\backslash[e]$ is $[a,b,c,d,e]\backslash[e]$. Operationally, the precondition for difference-list concatenation, i.e. $T_1=H_2$, is usually met by keeping T_1 a variable. For example, $[a,b,c|Xs]\backslash Xs$ can be concatenated to any difference-list. Its concatenation with $[d,e|Ys]\backslash Ys$ gives $[a,b,c,d,e|Ys]\backslash Ys$, which can be further concatenated to any list.

Difference-lists are the preferred representation of lists when concatenation is required. Programs that use difference-lists do not require explicit list concatenation using a predicate like *append*, and are thus more efficient both in time and in space. Operationally, they achieve an effect similar to that of *rplcd* in Lisp, but without destructive data-manipulation operations. However, the precondition for difference-list concatenation, i.e. $T_1=H_2$, cannot always be met, for example when the same list needs to be concatenated to several lists.

The third example is a recursive program for flattening a tree. It operates on trees constructed recursively from $tree(L,R)$ and $leaf(X)$, where L and R are recursively trees, and X is the value at a leaf. The predicate $flatten(T,Xs)$ holds if Xs is the list of values at the leaves of the tree T, ordered from left to right. The predicate $flatten'(T,Xs\backslash Ys)$ holds if the difference-list $Xs\backslash Ys$ represents the list thus defined.

```
flatten(T,Xs) ←                      %1
    flatten'(T,Xs\[ ]).

flatten'(leaf(X),[X|Xs]\Xs).         %2
flatten'(tree(L,R),Xs\Zs) ←          %3
    flatten'(L,Xs\Ys),
    flatten'(R,Ys\Zs).
```

The program employs several standard difference-list cliché's. The call from *flatten* to *flatten'* in Clause 1 employs the standard translation between lists and difference-lists: if $H\backslash T$ represents the list L and $T=[\ ]$ then $H=L$. *flatten'* returns a singleton difference-list in Clause 2, and implicitly concatenates the difference-lists representing the leaves of the subtrees by calling the tail of the first and the head of the second with the same name, Ys, in Clause 3.

The program has only deterministic and successful computations on initial goals $flatten(T,Xs)$, where T is a complete tree and Xs is a variable. For example:

$\langle \text{flatten(tree(leaf(a),tree(leaf(b),leaf(c))),Xs)};\ \varepsilon\rangle \xrightarrow{\text{Reduce}(1,1)}$

$\langle \text{flatten}'\text{(tree(leaf(a),tree(leaf(b),leaf(c))),Xs}\backslash[\ ]);\ \varepsilon\rangle \xrightarrow{\text{Reduce}(1,3)}$

$\langle \text{flatten}'\text{(leaf(a),Xs}\backslash\text{Ys), flatten}'\text{(tree(leaf(b),leaf(c))), Ys}\backslash[\ ]);\ \varepsilon\rangle \xrightarrow{\text{Reduce}(1,2)}$

$$\langle \text{flatten}'(\text{tree}(\text{leaf}(b),\text{leaf}(c))),\text{Ys}\backslash[\ ]);\ \{\text{Xs}\mapsto[a|\text{Ys}]\}\rangle \xrightarrow{\text{Reduce}(1,3)}$$

$$\langle \text{flatten}'(\text{leaf}(b),\text{Ys}\backslash\text{Ys}'),\ \text{flatten}'(\text{leaf}(c),\text{Ys}'\backslash[\ ]);\ \{\text{Xs}\mapsto[a|\text{Ys}]\}\rangle \xrightarrow{\text{Reduce}(1,2)}$$

$$\langle \text{flatten}'(\text{leaf}(c),\text{Ys}'\backslash[\ ]);\ \{\text{Xs}\mapsto[a,b|\text{Ys}']\}\rangle \xrightarrow{\text{Reduce}(1,2)}$$

$$\langle \textit{true};\ \{\text{Xs}\mapsto[a,b,c]\}\rangle$$

The other three possible computations on the same initial goal would also be deterministic and yield the same answer substitution.

2.3 The operational view of the logical variable and unification

The main difference between logic programming and other computational models is the logical variable and its manipulation via unification.

The basic data-manipulation operation in logic programs — unification — results in a substitution. Operationally, a substitution can be thought of as a simultaneous assignment of values to variables, except that here:

- a variable can be assigned a value only once, and
- the value assigned can be itself another variable or a term containing variables.

The single-assignment property, the ability to assign one variable to another, and the ability to assign a term containing variables to a variable are all fundamental to logic programming, and are the source of many powerful logic programming techniques.

Since the basic computational step of a logic program requires the unification of a goal atom with the head of a clause, much of the effort in logic programming has been devoted to understanding both the implications of this operation and its efficient implementation. This study has led to the realization that many of the subcases of goal/clause unification correspond quite closely to basic data manipulation operations of conventional languages. For the logic programmer, this implies that these special cases can be used to achieve the effect of conventional data manipulation. For the logic programming language implementor this implies that unification can be implemented efficiently by compiling the special cases, when identifiable, into machine instructions that execute the more basic data manipulation operations.

The correspondence is illustrated in Figure 1. The left column enumerates basic data-manipulation operations of conventional languages such as Pascal or Lisp, with sample code fragments. The right column shows the corresponding special cases of unification, with the corresponding examples of goal and clause terms. In this figure $T = T'$ denotes the unification of the goal atom term T with the clause head term T'.

Note that the cases in the figure are not necessarily mutually exclusive. For example, the unification of a goal variable with a clause term both constructs the term and assigns it to the goal variable; the unification of a goal term with an incomplete clause term both tests for equality and performs data access.

2.4 Semantics of logic programs

We provide here definitions for some of the concepts used intuitively above.

Unification

A *substitution* is a function from variables to terms which is different from the identity function on a finite number of variables. A substitution θ is presented as the finite set of pairs $\{X_1\mapsto T_1,\ldots,X_n\mapsto T_n\}$, where $X_1,\ldots,X_n$ are the variables where θ is different from the identity function, and $T_i=\theta(X_i)$, $i=1,\ldots,n$.

For any term T and substitution θ, $T\theta$ denotes the term obtained by replacing every variable X in T by $\theta(X)$. A term T is an *instance* of a term T' if $T=T'\theta$ for some substitution θ. For

Conventional data manipulation operation	The corresponding special case of goal/clause unification
(Single-) Assignment $X := a$	Variable = Non-variable $X = a$
Equality testing $a = a?$	Term = Term $a = a$
Data access (e.g. *car* and *cdr* in Lisp, '.' in Pascal) $X := car([a,b,c])$, $Xs := cdr([a,b,c])$	Compound term = Incomplete compound term $[a,b,c] = [X\|Xs]$
Data construction (e.g. *cons* in Lisp, *new* in Pascal) $Ys := cons(a,Xs)$	Variable = Compound term $Ys = [a\|Xs]$
Parameter passing by value	Goal Term = Variable $f(a) = X$
Parameter passing by reference	Variable = Variable $X = Y$
No corresponding operation; similar to aliasing	Two variables = Same variable $(Y,Z) = (X,X)$

Figure 1: Basic data manipulation operations and the corresponding special cases in goal/clause unification

example, $f(X,a)$, $f(X,X)$, $f(a,a)$, $f(a,b)$, $f(g(Z),h(b))$ are all instances of $f(X,Y)$.

A substitution θ *is more general* than θ' if there is a substitution σ such that $\theta = \theta' \circ \sigma$, where $\circ$ denotes function composition. An equivalent condition is that $T\theta'$ is an instance of $T\theta$ for any term T. For example, $\{X \mapsto Y\}$ is more general than $\{X \mapsto a, Y \mapsto a\}$ and $\{X \mapsto f(Z)\}$ is more general than $\{X \mapsto f(a)\}$.

A substitution θ is a *unifier* of two terms T_1 and T_2 if $T_1\theta = T_2\theta$. For example, the substitution $\{X \mapsto a, Y \mapsto f(a), Z \mapsto b\}$ is a unifier of $p(X,b)$ and $p(f(Y),Z)$, and so is the substitution $\{X \mapsto f(Y), Z \mapsto b\}$.

A substitution θ is a *most general unifier* (*mgu*) of T_1 and T_2 if it is a unifier of T_1 and T_2 and is more general than any other unifier of T_1 and T_2.

In the previous example the second unifier is the most general one. The most general unifier of $m(X,[X|Xs])$and $m(X',[a,b,c])$ is $\{X \mapsto a, X' \mapsto a, Xs \mapsto [b,c]\}$, and the most general unifier of $a([X|Xs], Ys,[X|Zs])$ and $a([a,b,c],[d,e],Zs')$ is $\{X \mapsto a, Xs \mapsto [b,c], Ys \mapsto [d,e], Zs' \mapsto [a|Zs]\}$.

In the previous examples there was one most general unifier. The two terms $f(X)$ and $f(Y)$ have two most general unifiers, $\{X \mapsto Y\}$ and $\{Y \mapsto X\}$.

A *renaming* is a substitution that permutes its domain. An example is $\{X \mapsto Y, Y \mapsto X\}$. It can be shown that all most general unifiers are equivalent up to renaming, i.e. if θ and θ' are two most general unifiers of some terms than there is a renaming ρ such that $\theta = \theta' \circ \rho$. In addition, it can be shown that if two terms have a most general unifier, then they have an idempotent most general unifier, i.e. an *mgu* θ for which $\theta = \theta \circ \theta$.

We define a function *mgu*, which takes two terms and returns their set of idempotent most general unifiers, if there are any, and *fail* if there are none. Usually we do not care which mgu is employed; in such cases we write $mgu(T_1, T_2) = \theta$ instead of $\theta \in mgu(T_1, T_2)$.

For a detailed analysis of unification see [108]. The operational intuitions behind unification were elaborated in Section 2.3 above.

A transition system for logic programs

Transitions systems will be employed throughout this paper. We specify a transition system for logic programs, as well as general notions that will be used in subsequent transition systems for concurrent logic programs. The general style of the transition system is that of Pnueli [142]; the details are adapted from Gerth *et al.* [65].

Definition: Transition system for a logic program P.
We associate with every logic program P a ***transition system*** which consists of:

- A set of states.
  A ***state*** is a pair $\langle G;\theta\rangle$, where G (the ***goal***) is either a sequence of atoms or ***fail***, and θ is a substitution.
- A set of transitions.
  A ***transition*** is a function from states to sets of states. For states S, S' and transition t, we denote that $S' \in t(S)$ by $S \xrightarrow{t} S'$. The set includes the Reduce and Fail transitions defined in Section 2.1 above. ▮

Definition: Enabled transition, terminal state, success state, failure state.

- A transition t is ***enabled*** on a state S if $t(S)$ is non-empty.
- A state on which no transition is enabled is called a ***terminal state***. A terminal state of the form $\langle$*true*$;\theta\rangle$ is called a ***success state***, and $\langle$*fail*$;\theta\rangle$ a ***failure state***. ▮

Definition: Computation
A ***computation of a program P on a goal G*** is a (finite or infinite) sequence of states

$$c = S_1, S_2, \ldots$$

satisfying:

- Initiation: $S_1 = \langle G; \varepsilon\rangle$, where ε is the empty substitution.
- Consecution: For each k, $S_{k+1} \in t(S_k)$ for some transition t.
- Termination: c is finite and of length k only if S_k is terminal. ▮

Definition: Partial computation, partial answer substitution.
Any prefix of a computation is called a ***partial computation***. The ***partial answer substitution*** of the partial computation $\langle G,\varepsilon\rangle,\ldots,\langle G',\theta\rangle$ is θ restricted to variables of G. ▮

Soundness and completeness of the transition system

A rule that governs the And-nondeterministic choices, i.e. the choice which goal atom to reduce next, is called a ***computation rule*** [121]. Formally, it is a function from a goal to one of its constituent atoms. A computation obeys a computation rule if the goal atom selected at each transition is the one specified by the rule.

Theorem: Independence of the computation rule [19,121].
Let P be a program and R a computation rule. If P has a successful computation on a goal G with answer substitution θ, then it has a successful computation on G with answer substitution θ that obeys R.

The transition system for logic programs realizes, in effect, a proof procedure for logic programs. Each Reduce transition is actually an application of an inference rule, called SLD-resolution [80,121], which is a special case of Robinson's resolution inference rule [147]. SLD-resolution, and hence the transition system, have soundness and completeness properties that link their operational view to the logical view of of logic programs [121].

Notation: If A is an atom or a clause with variables $X_1,X_2,\ldots,X_n$, $(\forall)A$ denotes $(\forall X_1,X_2,\ldots,X_n)A$. If P is a program with clauses $C_1,C_2,\ldots,C_n$ then $(\forall)P$ is the conjunction $(\forall)C_1\wedge(\forall)C_2\wedge\cdots\wedge(\forall)C_n$.

Theorem: Soundness and completeness of SLD-resolution [80,19,121].

Let P be a program and A an atom.
1. (Soundness): If P has a computation on the initial goal A with answer substitution θ, then $(\forall)A\theta$ is a logical consequence of $(\forall)P$.
2. (Completeness): If $(\forall)A'$ is a logical consequence of $(\forall)P$, where A' is an instance of the atom A, then there is a computation of P on the initial goal A with answer substitution θ, such that A' is an instance of $A\theta$. ∎

Note that, in particular, if $(\forall)A$ is a logical consequence of $(\forall)P$, then there is a computation of P from A with answer substitution θ such that $A\theta$ is equal to A up to renaming.

The soundness theorem relates a successful computation with a proof of a goal. Given a program P, let $S_1 \xrightarrow{*} S_2$ denote that there is partial computation of P leading from S_1 to S_2. A partial computation from a unit goal can be viewed as a proof of a clause, whose head is the initial goal, instantiated by the partial answer substitution, and its body is the remaining goal, as shown by the following lemma:

Lemma: If $\langle G;\varepsilon\rangle \xrightarrow{*} \langle R;\theta\rangle$, then $(\forall)(G\theta \leftarrow R)$ is a logical consequence of $(\forall)P$. ∎

Hence every partial answer substitution can be thought of as a conditional answer to the query, whose condition is the yet-to-be-proved goal [146,205].

Program equivalence and observables
For simplicity, we assume the existence of some global vocabulary V, in which all programs and goals are written in.

A fundamental question in programming language semantics is when should two programs be considered equivalent. For example, correctness of program transformation can be studied only with respect such a notion of equivalence.

Usually, program equivalence is defined by assigning to each program a mathematical object, called its *meaning*, and defining two programs to be equivalent if they have the same meaning.

The meaning of a program is usually some abstraction of its possible computations. What is abstracted away and what is kept is, to some degree, arbitrary, and depends on what we wish to identify as the observable result of a computation. Hence the meaning of a program is sometimes referred to as its *observable behavior*, or, in case it is a set, as its *observables* for short.

In the case of logic programs there are several possible notions of equivalence. One considers successful computations. Define the *success set* of a program P to be the set of ground atoms from which P has a successful computation. Two programs are *success set equivalent* if they have the same success set.

Success set equivalence does not capture differences in the answer substitutions computed by two programs. Define the *answer substitution set* of a program P to be the set of pairs $\langle G,\theta\rangle$ such that P has a successful computation from the goal G with answer substitution θ [45]. Two program are *answer-substitution equivalent* iff they have the same answer substitution set.

2.5 Prolog

Prolog is a concrete programming language based on the abstract logic programming model. Prolog employs a procedural reading of logic programs, in which each goal atom is viewed as a procedure call, and each clause $A \leftarrow B_1,B_2,\ldots,B_n$ is viewed as a definition of a procedure, similar to:

```
procedure $A$
begin
        call $B_1$,
        call $B_2$,
          ⋮
        call $B_n$
```

end

Such a clause is interpreted: "To execute procedure A, call B_1 and call B_2 and ... and call B_n". Prolog uses unification to realize various aspects of procedural languages such as parameter passing by reference or by value, assignment, and data selection and construction, as was shown in Figure 1 above.

Formally, this operational behavior is achieved by employing a computation rule that selects the leftmost atom in a goal, thus eliminating And-nondeterminism. Instead of the Reduce transition of logic programs, Prolog employs the following transition rule:

- $\langle A_1, A_2, \ldots, A_n; \theta \rangle \xrightarrow{\text{Reduce}_{\text{Prolog}}} \langle (B_1, \ldots, B_k, A_2, \ldots, A_n)\theta'; \theta \circ \theta' \rangle$
  If $mgu(A_i, A) = \theta'$ for some renamed apart clause $A \leftarrow B_1, \ldots, B_k$ of P.

The resulting transition still incorporates Or-nondeterminism, which is interpreted in Prolog as implicit search for all solutions. That is, Prolog attempts to explore all computations from the initial goal, returning the answer substitutions of successful computations.

Most sequential Prolog systems compute the solutions to a goal by searching depth-first the computation tree induced by different choices of clauses. Typically, one solution is produced at a time, and additional solutions are searched for only by request. Under this behavior it is possible for a program to produce several solutions, and then diverge. The point of divergence is determined by the order of clause selection. Usually a Prolog program is defined as a sequence (rather than set) of clauses, and the order of clause selection is textual order.

The possibility of divergence in the face of both successful and infinite computations makes Prolog incomplete as a proof procedure for logic programs (see Section 2.4). However, this incompleteness is not a major problem in practice. Knowing the Prolog computation rule, Prolog programmers order bodies of clauses so that infinite computations are avoided on expected goals. In the example logic programs in Section 2.1 above, Prolog computations terminate on $sum(Xs,S)$ goals whose first argument is a complete list of numbers; on $in\_both(X,L1,L2)$ if both L_1 and L_2 are complete lists, and on $flatten(T,Xs)$ if T is a complete tree.

Prolog is a convenient language for a large class of applications. However, to be practical it augmented the pure logic programming model with extra-logical extensions [171]. The main purpose of these extensions is to specify input/output and to realise a shared modifiable store. As we shall see later, this deficiency is peculiar to Prolog, and is not inherent to the logic programming model. Indeed, concurrent logic programs can specify both input/output and shared modifiable store in a pure way, relying solely on their different computation rule and different interpretation of nondeterminism.

PART II. CORE CONCEPTS AND TECHNIQUES

3. Concurrent Logic Programming

Transformational vs. reactive languages

Prolog is a sequential programming language, designed to run efficiently on a von Neumann machine by exploiting its ability to perform efficient stack management. Sequential Prolog can be parallelized, and much research is devoted to effective ways of doing so [122,10,207]. Nevertheless, Prolog, whether executed sequentially or in parallel, should not be termed a concurrent programming language.

To understand why Prolog and other parallelisable sequential languages cannot be termed concurrent languages, it is useful to distinguish between two types of systems, or programs: transformational and reactive [71]. The distinction is closely related to the distinction between closed and open systems [79]. A transformational (closed) system receives an input at the beginning of its operation and yields an output at its end. On the other hand the purpose of a reactive (open) system is not necessarily to obtain a final result, but to maintain some interaction with its environment. Some reactive systems, such as operating systems, database management systems, etc., ideally never terminate, and in this sense do not yield a final result at all.

All classical sequential languages in general, and Prolog in particular, were designed with the transformational view in mind. These languages contain some basic interactive input/output capabilities, but usually these capabilities are not an integrated component of the language and sometimes, as in Prolog, are completely divorced from its basic model of computation.

It may seem that the distinction between transformational and reactive systems is not directly related to concurrent systems, and perhaps there could be concurrent transformational systems as well as concurrent reactive ones. Indeed, there are concurrent systems that exploit parallelism to achieve high performance in applications that are transformational in nature, such as the solution of large numerical problems. Following Harel [70], we call concurrent systems that are transformational as a whole *parallel systems*. However, if we investigate the components of any concurrent system — whether transformational or reactive as a whole — we find these components to be reactive; they maintain continuous interaction at least with each other and possibly also with the environment.

Hence, there seems to be a common aspect to all concurrent systems or algorithms, independently of what is their target architecture, and whether they exploit concurrency to achieve higher performance, physical distribution, or better interaction with their environment. The common aspect is that a language that describes and implements them needs to specify reactive processes — their creation, interconnection, internal behavior, communication and synchronization.

Don't-know and don't-care nondeterminism

Many abstract computational models are nondeterministic, including nondeterministic Turing machines, nondeterminitic finite automata, and logic programs. Reactive systems are also nondeterministic. However, the nature of nondeterminism in the former is very different from the one employed in the latter. Kowalski [102] adequately termed nondeterminism of the first type *don't-know* nondeterminism, and of the second type *don't-care* nondeterminism[3]. Don't-care nondeterminism is often called also *indeterminism*, and we will use these two notions interchangeably.

The don't-know interpretation of nondeterminism implies that the programmer need not know which of the choices specified in the program is the correct one; it is the responsibility of the execution of the program to choose right when several transitions are enabled. Formally, this is achieved by specifying results of only successful computations as observable. Examples of such observables are the set of strings accepted by a nondeterministic finite automaton, or goal–answer substitutions pairs of successful computations of a logic program.

Don't-know nondeterminism is a very convenient tool for specifying transformational closed systems, as witnessed by the Prolog language. However, it seems to be incompatible with reactive open systems. The essence of don't-know nondeterminism is that failing computations "don't count", and only successful computations may produce observable results. However, it is not possible, in general, to know in advance whether a computation will succeed or fail; hence a don't-know nondeterministic computation cannot produce partial output before it completes; and hence it cannot be reactive[4].

The don't-care interpretation of nondeterminism, on the other hand, requires that results of failing computations be observable. Hence a don't-care nondeterministic computation may produce partial output (partial answer substitutions, in the case of concurrent logic programs) even if it is

[3] Manna and Pnueli [125] call the first *existential nondeterminism* and the second *universal nondeterminism*.

[4] A related argument with a similar conclusion is given by Ueda [202].

not known whether the computation will eventually succeed or fail.

Don't-care nondeterminism seems to be unnecessary, sometimes even a nuisance, in the specification of transformational systems, but as we shall see it is essential in the specification of concurrent reactive systems.

Although the nondeterminism of abstract computational models is commonly interpreted as don't-know nondeterminism, such models are also open to the don't-care interpretation. For example, nondeterministic finite automata can be used to specify either formal languages [88] (don't-know nondeterminism), or finite-state reactive systems (don't-care nondeterminism) [125]. The logic programming model is also open to these two interpretations. Prolog takes the don't-know interpretation, whereas concurrent logic language, being geared for specifying reactive open systems, take the don't-care interpretation.

Formally, the two interpretations of nondeterminism induce different notions of equivalence on the set of programs. Assume some notion of equivalence of two (either failing and successful) computations. For example, in logic programs two computations on the same initial goal are equivalent if they have the same answer substitution and same mode of termination. Under the don't-know interpretation, two programs are equivalent if they have equivalent successful computations. Under the don't-care interpretation, two programs are equivalent if they have equivalent computations, whether successful or not.

We emphasize that concurrent logic languages are not unique in adopting the don't-care interpretation of nondeterminism. Rather, almost all models of concurrency and concurrent programming languages, including CSP [86,87], CCS [129], UNITY [16], Occam [91], Ada, and others, take this approach as well. The difference is that concurrent logic languages have as an ancestor an abstract nondeterministic computational model — namely logic programs — whose nondeterminism can be interpreted both as don't-know and as don't-care. The other concurrent models and languages do not have related models or languages which incorporate don't-know nondeterminism, hence for them the questions addressed here are usually not raised.

One active research direction in logic programming explores parallel (non reactive) languages that incorporate both don't-know and don't-care nondeterminsm [209,210,150,153,154,157,72,8, 179]. The goal of these languages it to execute logic programs more efficiently by exploiting determinism, more sophisticated control, and parallelism. This research direction is outside the scope of the survey. It is discussed further in Chapter 21.

What are concurrent logic languages?

Concurrent logic languages are logic programming languages that can specify reactive open systems, and thus can be used to implement concurrent systems and parallel algorithms. A concurrent logic program is a don't-care nondeterministic logic program augmented with synchronization. A logic program thus augmented can realize the basic notions of concurrency — processes, communication, synchronization, and indeterminism.

The process reading of logic programs [42], employed by concurrent logic programs, is different from the procedural reading employed by Prolog and mentioned in Section 2.5. In the process reading of logic programs each goal atom $p(T_1,\ldots,T_n)$ is viewed as a process, whose program state ("program counter") is the predicate p/n and data state ("process registers") is the sequence of terms $T_1,\ldots,T_n$. The goal as a whole is viewed as a network of concurrent processes, whose process interconnection pattern is specified by the logical variables shared between goal atoms. Processes communicate by instantiating shared logical variables and synchronize by waiting for logical variables to be instantiated. This view is summarized in Figure 2.

The possible behaviors of a process are specified by guarded Horn clauses, which have the form:

$$Head \leftarrow Guard \mid Body.$$

The head and guard specify the conditions under which the Reduce transition can use the clause, as well as the effect of the transition on the resulting state. This is explained further below. The

Process model	Concurrent logic programming model
Process	Goal atom
Process network	Goal (collection of atoms)
Instruction for process action	Clause (See Figure 3)
Communication channel; Shared location	Shared logical variable
Communication	Instantiation of a shared variable
Synchronization	Wait until a shared variable is sufficiently instantiated

Figure 2: The process reading of logic programs

body specifies the state of the process after taking the transition: a process can halt (empty body) change state (unit body), or become several concurrent processes (a conjunctive body). This is summarized in Figure 3.

Halt:	$A \leftarrow G \mid \mathit{true}.$
Change (data and/or program) state (i.e., become a different process):	$A \leftarrow G \mid B.$
Become k concurrent processes:	$A \leftarrow G \mid B_1,\ldots,B_k.$

Figure 3: Clauses as instructions for process behavior

Concurrent logic languages employ the don't-care interpretation of nondeterminism. Intuitively, this means that once a transition has been taken the computation is committed to it, and cannot backtrack or explore in parallel other alternatives. Formally, this is realized by making observable partial results of the computation, as well as the final results of both successful, failing, and deadlcoked computations [65], as explained in Section 9 below.

The *head* and *guard* of a guarded clauses specify conditions on using the clause for reduction. A guarded clause can be used to reduce a goal atom only if the conditions specified by the head and the guard are satisfied by the atom. Concurrent logic languages differ in what can be specified by the head and the guard. A flat concurrent logic language incorporates a set of primitive predicates; in the languages surveyed these include mainly equality, inequality and arithmetic predicates. A guard in a flat language consists of a (possibly empty) sequence of atoms of these predicates. In a non-flat language, on the other hand, the guard may contain both primitive and defined predicates, and thus guard computations may be arbitrarily complex. Since guards of a non-flat language are recursively defined by guarded clauses, a computation of it forms an And/Or-tree of processes. In a flat language the processes are a "flat" collection; hence their name. Flat languages have received most of the recent attention of researchers, because it was found that their simplicity and amenability to efficient implementation come at a relatively low cost in expressiveness and convenience, when compared to non-flat languages (discussed in Section 18).

Concurrent processes communicate by instantiating shared logical variables, and synchronize by waiting for variables to be instantiated. Variable instantiation is realized in most concurrent logic language by unification. Three approaches were proposed to the specification of synchronisation in concurrent logic programming: input matching (also called input unification, one-way unification, or just matching) [20,24,198], read-only unification [160], and determinacy conditions

[210]. All share the same general principle: the reduction of a goal atom with a clause may be suspended until the atom's arguments are further instantiated. Once the atom is sufficiently instantiated, the reduction may become enabled or terminally disabled, depending on the conditions specified by the head and guard. Since input matching is the simplest and most useful synchronization mechanism, we present it here and defer the discussion of the others till the languages that employ them are introduced.

The matching of a goal atom A with a head of a clause $A' \leftarrow G|\ B$ succeeds if A is an instance of A'; in such a case it returns a most general substitution θ such that $A = A'\theta$. It fails if the goal atom and the head are not unifiable. Otherwise it suspends. More precisely,

$$match(A,A') = \begin{cases} \theta & \theta \text{ is the most general substitution such that } A = A'\theta \\ fail & \text{if } mgu(A,A') = fail \\ suspend & \text{otherwise.} \end{cases}$$

Unlike unification, there is only one most general matching substitution. Using matching for reducing a goal with a clause delays the reduction until the goal is sufficiently instantiated, so that its unification with the clause head can be completed without instantiating goal variables. Examples are given in Figure 4.

Goal	Clause head	Result
$p(a)$	$p(X)$	$\{X \mapsto a\}$
$p(X)$	$p(a)$	*suspend*
$p(a)$	$p(b)$	*fail*
$sum([1\|In],Out)$	$sum([X\|Xs],S)$	$\{X \mapsto 1, Xs \mapsto In, S \mapsto Out\}$
$sum(In,Out)$	$sum([X\|Xs],S)$	*suspend*
$sum([\,],Out)$	$sum([X\|Xs],S)$	*fail*

Figure 4: Examples of input matching of goals with clause heads

The dataflow nature of matching is evident: an "instruction" (clause) is enabled as soon as sufficient "data" (variable instantiations) arrive. Although simple, matching is found in practice sufficiently powerful for all but the most complex synchronization tasks, as demonstrated by the programming techniques in Section 7.

Languages in the concurrent logic programming family differ mainly in the capabilities of their output mechanism. On one end of the spectrum there are languages that allow only matching prior to clause selection and perform unification past clause selection. On the other end there are languages which allow both matching and unification as tests prior to such a commitment. Test unification in its most general form subsumes powerful synchronization mechanisms used in more conventional models such as multiple simultaneous test-and-set and CSP-like output guards.

These differences and others are further elaborated upon when discussing the various languages in Part III of the paper. Until then we concentrate on the common aspects of the family.

4. FCP(|) — A Simple Concurrent Logic Programming Language

We illustrate the various aspects of concurrent logic programming discussed in the previous section using a simple concurrent logic language, FCP(|) (read "FCP-commit")[5]. FCP(|) is closely related

[5] The nomenclature we use to describe concurrent logic languages is influenced by the one used by Saraswat [150].

to Flat GHC [198] and to Oc (read "Oh see!") [81,83]. We use FCP(|) as the introductory language instead of the more familiar language Flat GHC since its definition is simpler, and since it can more easily express some of the programming techniques related to distributed termination detection, discussed in Section 7. However all programs shown in Sections 5 and 7 are legal Flat GHC programs as well, and, except for the termination detection programs, the difference between the behavior of these programs under the operational semantics of Flat GHC and of FCP(|) is immaterial. See the discussion of Flat GHC in Section 10.

4.1 Syntax

Definition: Guard test predicates, guarded clause, FCP(|) program.

- We assume a fixed finite set of guard test predicates, including $integer(X)$, $X < Y$, $X = Y$, $X \neq Y$, and others. The predicates assumed in this paper are given in Section 4.2 below.
- A *guarded clause* is a formula of the form:

  $$A \leftarrow G_1,\ldots,G_m \mid B_1,\ldots,B_n. \qquad m,n \geq 0,$$

  where A, $G_1,\ldots,G_m$, $B_1,\ldots,B_n$ are atoms, the predicate of each G_i, $i = 1,\ldots,m$ is a guard test predicate and the variables of G_i occur in A. If the quard is empty ($m = 0$) then the commit operator '|' is omitted. An empty body ($n = 0$) is denoted by *true*.
- An FCP(|) *program* is a finite sequence of guarded clauses, which contains the unit clause $X = X$ as the only clause with head predicate '='. ∎

Note: '=' is a primitive predicates in FCP(|) that cannot be redefined by a program. The reason for a program being a sequence of clauses, rather than a set, will become apparent when we discuss the *otherwise* predicate in Section 7.

4.2 Operational semantics

Modelling concurrency by interleaving atomic actions

We specify the behavior of concurrent logic programs in general, and FCP(|) programs in particular, using a transition system very similar to that of logic programs. In this standard approach [142,16], concurrency is modelled by the nondeterministic interleaving of the atomic actions of the processes participating in the computation. The approach requires, therefore, a precise specification of what is an atomic step of execution, as differences in the grain of atomic actions may lead to radically different computational models. As we shall see, one of the major differences between the various concurrent languages is indeed the grain of their atomic actions.

Our transition system is not reactive: it does not model input from an outside environment. This is not a major drawback, since if we wish to investigate a reactive computation of a program P from a goal G, we can model the environment as another process G', whose behavior is specified by a program, say E, with predicates disjoint from P, and investigate computations of the program $P \cup E$ from the conjunctive goal (G,G') [59]. An alternative is to add an explicit input transition [119].

Modeling concurrency by interleaving is a common approach, which has the advantage of being simple and well understood. Its disadvantage is that concurrency is not explicit, and hence an interleaving model sometimes gives rise to artificial fairness problems, which are not present if the concurrency is explicit in the model. We defer the discussion of fairness to Section 6.

Guard test predicates and guard checking

The meaning of the guard test predicates is given via a fixed set of ground atoms T over these predicates. The predicates used in this paper and their meanings are:

$X=X$ for every ground term X.

but is different from it.

$X \neq Y$	for every two ground terms X and Y which are not equal.
integer(X)	for every integer X.
$X<Y$	for every ground arithmetic expressions X and Y such that the value of X is less than the value of Y.
$X \leq Y$	for every ground arithmetic expressions X and Y such that the value of X is less than or equal the value of Y.
$X=:=Y$	for every ground arithmetic expressions X and Y whose values are the same.
$X \backslash=:= Y$	for every ground arithmetic expressions X and Y whose values are different.

There are three guard primitives *var*(X), *unknown*(X), and *otherwise*, whose semantics cannot be given simply by a set of ground atoms. It is discussed when the primitives are introduced. The set of guard test predicates of "real" concurrent logic languages is not much larger than the list above. See for example [100,170].

An atom is true in T if every instance of it is in T, and false otherwise. A conjunction of atoms is true in T if all its members are true in T, and false otherwise. The check of a guard G *succeeds* if G is true in T; it *fails* if no instance of G is true in T; and it *suspends* otherwise. In other words, the check suspends if some future instantiation of the guard may result in it being true. For example:

checking *integer*(*2*) succeeds
checking *integer*(a) fails
checking *integer*(X) suspends

checking $3 < 5$ succeeds
checking $5 < 3$ fails
checking $3 < X$ suspends
checking $a < X$ fails

The clause try function
The only difference between the transition system of logic programs and of FCP(|) is that the Reduce and Fail transitions employ matching and guard checking instead of unification. The operation of matching and guard checking is captured by the clause try function, *try*.

The *mgu* function unifies the goal atom with the clause head, and returns a substitution or *fail*. The *try* function does the same if the goal atom is an equality and the clause is the equality clause $X=X$. Otherwise it matches the goal atom and clause head, and, if successful, checks the guard, instantiated by the matching substitution. It may return *suspend* if the matching or the guard check suspends. *try* is defined for equality goals as follows:

$$try(T_1=T_2, X=X) = mgu(T_1,T_2).$$

And for clauses whose head predicate is different from the equality predicate:

$$try(A,(A' \leftarrow G|B)) = \begin{cases} \theta & \text{if } match(A,A') = \theta \wedge \text{checking } G\theta \text{ succeeds} \\ fail & \text{if } mgu(A,A') = \theta \wedge \text{checking } G\theta \text{ fails} \\ & \vee\ mgu(A,A') = fail \\ suspend & \text{otherwise} \end{cases}$$

The transition system
The state of a computation of an FCP(|) program is, as in a logic program, a pair $(G;\theta)$, where G is a goal and θ a substitution. A computation begins from an initial goal $(G;\epsilon)$ and progresses using Reduce and Fail transitions, similar to computations of logic programs. The difference is that instead of unifying the goal atom with the clause head, the Reduce and Fail transitions use the clause try function, *try*, instead of the *mgu* function:

1. Reduce

   $\langle A_1,\ldots,A_i,\ldots,A_n;\theta\rangle \xrightarrow{\text{Reduce}} \langle (A_1,\ldots,B_1,\ldots,B_k,\ldots,A_n)\theta';\theta\circ\theta'\rangle$

   If $try(A_i,C) = \theta'$ for some renamed apart clause $C = A \leftarrow G \mid B_1,\ldots,B_k$ of P.
2. Fail

   $\langle A_1,\ldots,A_i,\ldots,A_n;\theta\rangle \xrightarrow{\text{Fail}} \langle fail;\theta\rangle$

   If for some i and for every renamed apart clause C of P, $try(A_i,C) = fail$.

Note that the *suspend* result of the try function is not used in the Reduce and Fail transitions. Its effect, therefore, is to prevent a goal atom from reducing with a clause and from failing. Specifically, if A is a goal atom for which $try(A,C)$=*suspend* for some clause C in P, and $try(A,C')$=*suspend* or *fail* for every other clause C' in P, then A can participate neither in a Reduce transition nor in a Fail transition. Such a goal atom is called *suspended*. A state consisting of a goal in which all atoms are suspended is terminal, as no transition applies to it. Such a state it is called a *deadlock state*, and a computation ending in a deadlock state is called a *deadlocked computation*[6].

The following lemma relates successful computations of an FCP(|) program to computations of the corresponding logic program.

Lemma (Soundness of FCP(|)):
Let c be a non-deadlocked computation of an FCP(|) program P. Then there is a finite subset T' of T such that c is also a computation of the logic program $P \cup T'$.
Proof: Immediate. ∎

The opposite direction of this lemma is of course not true. In particular, the logic program can proceed with a Reduce transition on states which the corresponding FCP(|) program deadlocks.

Observables of concurrent logic programs
As explained in Section 3, the observables of a concurrent logic program reflect both successful and failing computations.

Definition: Observables of a concurrent logic program
The *observable behavior* of a finite computation $c = \langle G_1,\varepsilon\rangle,\ldots,\langle G_n,\theta_n\rangle$ of a concurrent logic program P is the triple $\langle G_1,\theta,x\rangle$ where θ is the answer substitution of the computation (i.e. θ_n restricted to variables of G_1), and x=G_n if G_n=*true* or G_n=*fail*, and x=*deadlock* otherwise. The *observables* of a concurrent logic program P, $[\![P]\!]$, are the set of observable behaviors of every computation c of P. ∎

4.3 Examples of concurrent logic programs

We show several simple examples of concurrent logic programs, written in FCP(|), that correspond to the logic programs shown in Section 2.2, and describe their behavior.

The following concurrent logic program defines the process $sum(Xs,S)$, which unifies S with the sum of the elements of the input stream Xs.

```
sum(Xs,S) ←                %1
    sum'(Xs,0,S).

sum'([ ],P,S) ← P=S.       %2
sum'([X|Xs],P,S) ←         %3
    plus(X,P,P')
    sum'(Xs,P',S).
```

[6] Note, however, that this terminology is appropriate only for a closed computation. In an open computation variables may be instantiated by the environment, and hence a more refined definition of deadlock is required, which takes into account which variables are accessible to the environment.

There are two differences between this program and the logic program for *sum* shown in Section 2.2. The first is that the base clause of *sum′* unifies the partial sum *P* with the answer *S* explicitly in the body. This is necessary since in FCP(|) the goal atom is matched with the clause head, not unified with it. The second is that the definition of *plus* has to be modified to reflect the direction of the computation. *plus* behaves as if defined by clauses of the form:

```
plus(0,0,X) ← X=0.
plus(0,1,X) ← X=1.
plus(2,2,X) ← X=4.
    ⋮
```

Note also that each clause has an implicit commit operator "|", which is omitted by our syntactic conventions since the guard is empty. In contrast to the logic program for *sum*, the concurrent logic program has only successful computations on an initial goal *sum*(*Xs*,*S*), where *Xs* is a list of integers and *S* is a variable.

Consider the following partial computation:

$$\langle \mathrm{sum}([1,2],S);\, \varepsilon\rangle \xrightarrow{\text{Reduce}(1,1)}$$
$$\langle \mathrm{sum}'([1,2],0,S);\, \varepsilon\rangle \xrightarrow{\text{Reduce}(1,3)}$$
$$\langle \mathrm{plus}(1,0,P),\, \mathrm{sum}'([2],P,S);\, \varepsilon\rangle \xrightarrow{\text{Reduce}(2,3)}$$
$$\langle \mathrm{plus}(1,0,P),\, \mathrm{plus}(2,P,P'),\, \mathrm{sum}'([\,],P',S);\, \varepsilon\rangle$$

At this state the logic program could reduce any of the three atoms. However, the concurrent logic program cannot reduce the second *plus* process: it is suspended until its second argument *P* is instantiated. A possible continuation of this partial computation is:

$$\langle \mathrm{plus}(1,0,P),\, \mathrm{plus}(2,P,P'),\, \mathrm{sum}([\,],P',S);\, \varepsilon\rangle \xrightarrow{\text{Reduce}(1,\text{plus})}$$
$$\langle P=1,\, \mathrm{plus}(2,P,P'),\, \mathrm{sum}([\,],P',S);\, \varepsilon\rangle \xrightarrow{\text{Reduce}(3,2)}$$
$$\langle P=1,\, \mathrm{plus}(2,P,P'),\, P'=S;\, \varepsilon\rangle \xrightarrow{\text{Reduce}(3,=)}$$
$$\langle P=1,\mathrm{plus}(2,P,S);\, \varepsilon\rangle \xrightarrow{\text{Reduce}(1,=)}$$
$$\langle \mathrm{plus}(2,1,S);\, \varepsilon\rangle \xrightarrow{\text{Reduce}(1,\text{plus})}$$
$$\langle S=3;\, \varepsilon\rangle \xrightarrow{\text{Reduce}(1,=)}$$
$$\langle \mathit{true};\, \{S \mapsto 3\}\rangle$$

The logic program for flattening a tree can also be easily turned into a concurrent logic program. The only syntactic change required is specifying the construction of the output difference-list explicitly, by an equality goal:

```
flatten(T,Xs) ←                  %4
    flatten'(T,Xs\[ ]).
flatten'(leaf(X),Xs\Ys) ←        %5
    Xs=[X|Ys].
flatten'(tree(L,R),Xs\Zs) ←      %6
    flatten'(L,Xs\Ys),
    flatten'(R,Ys\Zs).
```

The *flatten* process can operate on a tree that is provided incrementally, by a concurrent process, as the two clauses of *flatten′* suspend until it is known whether their first argument is *leaf*(_) or *tree*(_,_). It can also be connected to the *sum* process, which incrementally sums the list produced by *flatten*, as shown by the following computation:

⟨flatten(tree(leaf(17),leaf(19)),Xs), sum(Xs,S); ε⟩ $\xrightarrow{\text{Reduce(1,4)}}$

⟨flatten′(tree(leaf(17),leaf(19)),Xs\[ ]), sum(Xs,S); ε⟩ $\xrightarrow{\text{Reduce(1,6)}}$

⟨flatten′(leaf(17),Xs\Ys), flatten′(leaf(19),Ys\[ ]),Xs\[ ]), sum(Xs,S); ε⟩ $\xrightarrow{\text{Reduce(1,5)}}$

⟨flatten′(leaf(19),Ys\[ ]), sum([17|Ys],S); {Xs↦[17|Ys]}⟩ $\xrightarrow{\text{Reduce(2,1)}}$

⟨flatten′(leaf(*19*),Ys\[ ]), *sum*([17|Ys],0,S); {Xs↦[17|Ys]}⟩ $\xrightarrow{\text{Reduce(2,3)}}$

⟨flatten′(leaf(19),Ys\[ ]), plus(17,0,P′), sum(Ys,P′,S); {Xs↦[17|Ys]}⟩ $\xrightarrow{\text{Reduce(2,plus)}}$

⟨flatten′(leaf(19),Ys\[ ]), P′=17, sum(Ys,P′,S); {Xs↦[17|Ys]}⟩ $\xrightarrow{\text{Reduce(2,=)}}$

⟨flatten′(leaf(19),Ys\[ ]), sum(Ys,17,S); {Xs↦[17|Ys]}⟩ $\xrightarrow{\text{Reduce(1,5)}}$

⟨sum([19],17,S); {Xs↦[17,19]}⟩ $\xrightarrow{\text{Reduce(1,3)}}$

⟨plus(19,17,P″), sum([ ],P″,S); {Xs↦[17,19]}⟩ $\xrightarrow{\text{Reduce(1,plus)}}$

⟨P″=36, sum([ ],P″,S); {Xs↦[17,19]}⟩ $\xrightarrow{\text{Reduce(2,2)}}$

⟨P″=36, P″=S; {Xs↦[17,19]}⟩ $\xrightarrow{\text{Reduce(1,=)}}$

⟨36=S; {Xs↦[17,19]}⟩ $\xrightarrow{\text{Reduce(1,=)}}$

⟨*true*; {Xs↦[17,19],S↦36}

Summing the elements of a tree can be done more efficiently by combining the two procedures, *flatten* and *sum*, into a single procedure *tree_sum*:

```
% tree_sum(T,S) ←
  S is the sum of values of the leaves of the tree T.

      tree_sum(T,S) ←
          tree_sum′(T,0,S).

      tree_sum′(tree(L,R),P,S) ←
          tree_sum′(L,P,P′),
          tree_sum′(R,P′,S).
      tree_sum′(leaf(X),P,S) ←
          plus(X,P,S).
```

This program spawns a network of linearly connected *plus* processes, which sum the leaf elements sequentially from left to right. A possible computation from the initial goal *tree_sum*(*tree*(*tree*(*leaf*(*17*),*leaf*(*19*)),*leaf*(*23*)),*S*) may content the intermediate goal *plus*(*0*,*17*,*P*), *plus*(*17*,*P′*,*P″*), *plus*(*23*,*P″*,*S*), which is then reduced from left to right. (Note however that the leftmost *plus* process may be reduced even if the spawning of the *plus* processes to its right has not been completed yet.) The program demonstrates that in a recursively constructed process tree, leaf processes (*plus* processes in our case) can communicate directly even if they are not directly related.

Turning the *in_both* logic program into a concurrent logic program is more difficult, since it employs don't-know nondeterminism in an essential way: it guesses a member of one list, and verifies that it is also a member of the second. The following logic program does not need to guess which clause to use, if the two lists are given in the initial goal, as in the goal *in_both(Xs,[1,2,3,4],[3,4,5,6])*. In such a case all computations of the following program are deterministic: each goal atom created during the computation of the program unifies with exactly one clause head. The program employs the difference-list technique.

% *in_both*(Xs,L_1,L_2) ←
  The list Xs contains the members of both lists L_1 and L_2.

```
        in_both(Xs,[X|L1],L2) ←
            member(X,L2,Xs\Xs'),
            in_both(Xs',L1,L2).
        in_both([ ],[ ],_).
```

% *member*($X,L,Xs\backslash Xs'$) ←
  X is a member of L and $Xs = [X|Xs']$ or X is not a member of L and $Xs = Xs'$.

```
        member(X,[X|L],[X|Xs]\Xs).
        member(X,[Y|L],Xs\Xs') ←
            X≠Y, member(X,L,Xs\Xs').
        member(X,[ ],Xs\Xs).
```

In contrast to the previous program, this program returns the (possibly empty) list of elements common to the two input lists, rather than nondeterministically selecting a common single element if one exists, or failing if there is none. Here *in_both*(Xs,L_1,L_2) holds if Xs is the list of all elements common to L_1 and L_2. The multiplicity of elements in Xs is the same as in L_2. This program employs a difference-list to construct the output. *member*($X,L_1,Xs\backslash Xs'$) holds if both X is in L_1 and X is the difference between Xs and Xs' (i.e. $Xs = [X|Xs']$) or if X is not in L_1 and the difference between Xs and Xs' is empty (i.e. $Xs = Xs'$).

Since the program is deterministic on the desired set of goals, it can be turned into a concurrent logic program quite easily. The following is such an FCP(|) program:

```
        in_both(Xs,[X|L1],L2) ←
            member(X,L2,Xs\Xs'),
            in_both(Xs',L1,L2).
        in_both(Xs,[ ],_) ← Xs=[ ].

        member(X,[X|L],Xs\Xs') ← Xs=[X|Xs'].
        member(X,[Y|L],Xs\Xs') ←
            X≠Y | member(X,L,Xs\Xs').
        member(X,[ ],Xs\Xs') ← Xs=Xs'.
```

Intuitively, the program operates as follows. On a call *in_both(Xs,L1,L2)* it spawns parallel *member(X,L2,Xs\Xs')* processes, one for each element of *L1*, using the recursive clause of *in_both*. Each of these processes searches down the list *L2* for an element equal to its X. If it finds one, it returns X in the difference-list $Xs\backslash Xs'$, by unifying Xs with $[X|Xs']$. Otherwise it returns the empty difference-list by unifying Xs with Xs'. The difference-lists are implicitly concatenated into the output list by the recursive clause of *in_both*. The output list is closed by the base clause of *in_both*.

The program operates correctly even if the two input lists *L1* and *L2* are given incrementally, by some concurrent process. This is achieved since the program inspects them using matching (specified by clause heads), which suspends if the input list is still unavailable. The output, in contrast, is constructed using unification, specified explicitly in the body of the clauses.

The guard of the second clause of *member* ensures that the clause is selected only after it is determined that X is different from Y. In the other clauses the guard is empty and the commit operator is implicit.

4.4 The power of the logical variable in concurrent programming

The standard uses of the logical variable and unification were mentioned in Section 2.3. Concurrent logic programming extends its use also to process communication. By following specific conventions and protocols, a wide range of concurrent programming techniques can be a realized using shared logical variables, unification, and matching.

In this section we provide a glossary of the major uses of the logical variable in concurrent programming. These will be demonstrated by concrete examples throughout the paper. When applicable, references to the relevant sections are provided.

It is useful to view variables shared between processes and their instantiation in two complementary ways: as communication channels, which *transmit message streams*, and as shared locations, which are instantiated, possibly incrementally and cooperatively, to compound data structures. Both are explained below.

Shared logical variables as communication channels, and communication stream protocols

Given the single-assignment nature of logical variables, it may seem that a communication channel implemented by a shared logical variable might carry at most one message. In some sense this is true. A better way to understand the situation, however, is to view a shared logical variable as a Genie, who will grant you a single wish[7]. A good strategy to follow when encountering such a Genie is, of course, to request to have two wishes. This is realized by instantiating the logical variable to a list (*cons*) cell, whose head and tail are logical variables. The head may be used to send the current message. Its tail is a new variable, shared by the processes sharing the original variable, and can be used for subsequent communications *ad infinitum*, as in

$$Xs = [m_1|Xs1],\ Xs1 = [m_2|Xs2],\ Xs2 = [m_3|Xs3],\ \ldots$$

In this way multiple "wishes" are achieved by the multiplicity of elements in a single list.

Hence, when serving as a communication channel, a shared logical variable is typically instantiated to a stream of messages. Several protocols can be followed in constructing such a message stream. They differ in the number of processes sharing the variable, and whether they share the writing and/or reading of the stream. Useful stream communication protocols include:

- point-to-point communication (single-writer single-reader), e.g. *sum_tree* in Section 4.3 above.
- broadcast communication, (single-writer multiple-reader) (Section 7.7)
- duplex communication (two writers/readers, who use the stream both for bidirectional communication and for tight synchronization) (Section 14)
- many-to-one communication (multiple-writer, single-reader), (Section 15) and
- blackboard communication (multiple writer/reader, cooperatively reading and writing the stream) (Section 15).

These stream protocols require progressively stronger synchronization mechanisms. Only the first two can be implemented by all concurrent logic languages. The language properties required to realise the duplex protocol and multiple-producer protocols are described when the protocols are introduced.

A single stream is not always the preferred data structure for communication. For high-volume many-to-one communication, nondeterministic merging of multiple single-writer streams is usually preferred over having multiple writers cooperatively produce a single stream, since it eliminates contention on the stream's tail. Stream merging is discussed in Sections 6 and 7. In addition, when the set of writer and the set of readers in a multiple-writer multiple-reader stream are disjoint, it is

[7] This analogy is due to Bill Silverman.

often the case that the total ordering of a stream imposes on its elements unnecessarily serialized sending and receiving of messages. In such a case, a more general data-structure, called *Channel*, may be appropriate [194]. A Channel is a partially ordered set of messages, which can be produced in parallel without contention, and can be read with the same degree of parallelism with which it was produced.

Incomplete-message protocols

Messages sent on a stream need not be ground (i.e. variable-free) terms. A message containing variables is called an *incomplete message*. The ability to send incomplete messages is perhaps the single most important reason for the added flexibility and expressiveness of concurrent logic programming languages over other concurrent languages. D.H.D. Warren once characterized Prolog as "pointers made easy". Adapted to concurrent logic programming, the slogan may read "communication channels made easy". Indeed, incomplete messages may be viewed as a structured and high-level way of dynamically allocating and distributing communication channels.
There are several useful protocols employing incomplete messages:

- *Back-communication protocol* (the "remote procedure call" effect) (Section 7.3).
  A sender sends a message with a newly allocated reply variable R, and waits for R to be instantiated. The receiver responds to the message by instantiating R to the reply. This protocol achieves essentially the effect of a remote-procedure-call mechanism, without adding any special constructs to the language.
  It is easy to program servers using many-to-one message streams and the back-communication protocol. There is no need for the server to know the identity of the client who sent the request, nor for specially programmed mechanisms to route replies back to the clients.
  If a server can serve multiple requests in parallel, then a many-to-many channel may be preferred to a stream.
- *Dialogue* (Section 7.4).
  The back-communication protocol constitutes only one round in a possibly longer dialogue. The reply to which the variable R is instantiated may contain another reply variable R', with which the original sender can continue the dialogue, repeating the back-communication protocol as long as desired by both parties.
- *Network formation protocol* (Section 7.2).
  Assume that two processes p and c sharing a variable X want each to become n processes $p_1,\ldots,p_n$ and $c_1,\ldots,c_n$, and establish a communication stream X_i between p_i and c_i, $i = 1,\ldots,n$. This can be achieved as follows. p sends to c the message $[X_1,\ldots,X_n]$, which contains n variables, and then creates n processes, providing the i^{th} process with X_i. Upon receipt of the message c creates n processes and similarly provides its i^{th} process with X_i.
  In a variant of this protocol, the stream $[X_1,\ldots,X_n]$ is constructed incrementally by p. The process c need not know the stream's length in advance, and can create the next process c_i when X_i is available. This technique is heavily used in the formation of recursive process networks, as described in Section 7.2.
- *Network reconfiguration protocol.*
  Assume a process p that shares a variable Q with a process q and a variable R with a process r. If p wants to establish direct communication between q and r it sends to q on the stream Q an incomplete message containing R. Once q receives that message, it can use R to communicate with r directly.
  This technique can be employed to form an arbitrary communication graph in a network, independently of how the network was created to begin with. In particular, in a recursively constructed network the communication graph need not follow the path of the recursion, and two "leaf" processes may communicate directly no matter how high up their common ancestor is (Section 4.3).
- *Bounded-buffer communication protocol* (Section 7.3).

The basic stream communication protocol is asynchronous. However, using incomplete messages one can implement synchronised communication. For example, a k-bounded-buffer protocol, for $k \geq 1$, can be implemented using a single-writer single-reader stream of incomplete messages as follows [177]: Each message contains an acknowledgement variable. The reader acknowledges the message by instantiating the variable to some constant upon receipt. The writer does not send the $n+k^{th}$ message before receiving an acknowledgement for the n^{th} message.

Shared logical variables as shared locations

A shared logical variable can also be viewed as a shared location that can be assigned a data structure, i.e. a logical term. The term may be compound, and its construction may proceed incrementally and in cooperation between the processes initially sharing the variable.

In the special case that the term is a stream we obtain the stream communication protocols described above. However, the fact that stream communication results in a data structure, and is not just a sequence of events that occur in time, has several ramifications. Two of them are:

- *Communication history can be kept and later examined.*
  A shared variable used as a communication channel is incrementally instantiated to a stream data structure, which contains the messages and replies sent on it. Typically, a stream writer or reader iterates with the tail of the stream once its head was written or read, and eventually the head becomes inaccessible. Memory occupied by inaccessible data-structures is eventually reclaimed by garbage-collection. Alternatively the initial stream variable can be kept, either by the process communicating via the stream itself, or by a concurrent observer who shares the initial stream variable. The data structure kept by the observer can be used later for various purposes, such as debugging, logging, and recovery.
  The stream data structure reflects only the order in which messages were sent and their content, but does not record the order in which message subterms, including replies, were constructed. In addition, if a process communicates via several independent streams, their content cannot be used to determine the temporal relations between messages on different streams.
  For some applications this abstract form of communication history, represented by a stream term, is sufficient. If a more precise history of the computation is required, e.g., to diagnose transient timing bugs, a different technique for recording information about a computation, which is sufficient for its accurate reconstruction, can be used [118]. This is further discussed in Section 14.
- *A message stream can be inspected and transformed.*
  A process may examine or transform its incoming stream before processing its messages, if it so desires. A simple illustration of this is the ability of a process to "send to self", a useful object-oriented programming paradigm. To do so a process prepends a message to its input stream and proceeds with the resulting stream.

The constructed term need not be a stream, however. For example, in the distributed database system of Reches *et al.* [144], a transaction is a tree-structured term that is constructed cooperatively by the user program and the database system. The user program constructs the term, possibly concurrently, out of terms corresponding to sub-transactions, leaving in it variables for the database system replies. The database system consumes the term, executing, possibly concurrently, the subtransactions, and instantiates the reply variables to the answers.

Another example is the type-checker of Yardeni [211]. In it, multiple processes cooperate in constructing a term representing a finite automaton that defines the type of the program being checked. The programming technique used is similar to the multiple-writer stream described in Section 15.

5. Basic Programming Examples and Techniques

This section examines the operational behavior of FCP(|) programs via examples, and illustrates basic concurrent logic programming techniques.

Writers and readers
A writer process $p(X)$, that unifies X with a and halts, can be defined using the single clause program:

$$p(X) \leftarrow X = a. \qquad \%1$$

A reader $c(X)$ that waits till X is a and then halts is defined by:

$$c(a). \qquad \%2$$

A computation starting from a writer and a reader connected by the variable X, progresses as follows:

$$\begin{array}{l} \langle p(X),\ c(X);\ \varepsilon\rangle \xrightarrow{\text{Reduce(1,1)}} \\ \langle X{=}a, c(X);\ \varepsilon\rangle \xrightarrow{\text{Reduce(1,=)}} \\ \langle c(a);\ \{X{\mapsto}a\}\rangle \xrightarrow{\text{Reduce(2,2)}} \\ \langle true;\ \{X{\mapsto}a\}\rangle \end{array}$$

The final state is a success state, and this is the only possible computation from that initial state.

Many reader may read the same value, giving the effect of a broadcast. The computation starting from the initial state:

$$\langle p(X), c(X), c(X), \ldots, c(X);\ \varepsilon\rangle$$

reduces $p(X)$, unifies $X = a$, and then reduces all the $c(a)$ goal atoms one by one in some arbitrary order.

Nondeterminism in writers and readers
A process $p_2(X)$ that nondeterministically chooses to unify X with a or with b is defined by:

$$\begin{array}{ll} p_2(X) \leftarrow X{=}a. & \%1 \\ p_2(X) \leftarrow X{=}b. & \%2 \end{array}$$

There are two possible computations if we use the nondeterministic writer p_2 instead of p in the example above. A successful one, essentially identical to the one above, and a failing one:

$$\begin{array}{l} \langle p_2(X),\ c(X);\ \varepsilon\rangle \xrightarrow{\text{Reduce(1,2)}} \\ \langle X{=}b,\ c(X);\ \varepsilon\rangle \xrightarrow{\text{Reduce(1,=)}} \\ \langle c(b);\ \{X{\mapsto}b\}\rangle \xrightarrow{\text{Fail(1)}} \\ \langle fail;\{X{\mapsto}b\}\rangle \end{array}$$

If instead of $c(X)$ we use a nondeterministic reader $c_2(X)$, which accepts either a or b as values for X:

$$\begin{array}{l} c_2(a). \\ c_2(b). \end{array}$$

then instead of failing this latter computation would proceed and terminate successfully.

The process $c_2(X)$ has two alternatives. Which one is taken is completely determined by its environment. $p_2(X)$ has also two alternatives. However, the environment has no effect on its choice. An intermediate example is the process $cp(X,Y,Z)$, which behaves as follows. If $X = a$ it

Z with a. If $Y = b$ it unifies Z with b. If both $X = a$ and $Y = b$ it nondeterministically chooses one of the two.

$$cp(a,Y,Z) \leftarrow Z{=}a.$$
$$cp(X,b,Z) \leftarrow Z{=}b.$$

Starting from the initial state:

$$\langle p_2(X), p_2(Y), cp(X,Y,Z); \varepsilon \rangle$$

there are several possible computations, depending on the choice of the goal atom and the clause. To focus on clause choices, assume that goal atoms are reduced from left to right. Then there are five possible computations. Three of the four choices of the two p_2 processes uniquely determine the behavior of $cp(X,Y,Z)$. For example, if $p_2(X)$ unifies $Y = a$ and $p_2(Y)$ unifies $X = a$ then cp must unify $Z = a$. If $X = b$ and $Y = a$, then cp fails and the computation fails. However, if $p_2(X)$ chooses $X = a$ and $p_2(Y)$ chooses $Y = b$, then cp has a choice: it can either reduce with the first clause, and unify $Z = a$, or with the second, and unify $Z = b$. Both computations are possible.

Streams: producers, consumers, transducers, distributors, and mergers
As mentioned in Section 3, a stream is a list constructed incrementally.

Stream producers
Assume a process $X := E$, which evaluates the arithmetic expression E when it becomes ground, and unifies X which its value (it can be defined in FCP(|) using more primitive arithmetic processes, such as *plus* shown in Section 4.3). A process *integers(From,To,Xs)*, which, given integers *From* and *To*, produces the stream *[From,From +1,...,To]*, can be defined by:

% *integers(From,To,Ns)* ← *Ns* is the list of integers from *From* to *To*[8].

```
integers(From,To,Ns) ← From > To | Ns=[ ].
integers(From,To,Ns) ← From ≤ To | Ns=[From |Ns'],
    From' := From +1,
    integers(From',To,Ns').
```

A more interesting stream producer is $fib(N,Ns)$, which produces the elements of the Fibbonacci series less than or equal to N.

% $fib(N,Ns)$ ← Ns is the Fibbonacci series less than or equal to N.

```
fib(N,Ns) ←
    fib'(N,0,1,Ns).

fib'(N,N1,N2,Ns) ← N < N1 | Ns=[ ].
fib'(N,N1,N2,Ns) ← N ≥ N1 | Ns=[N1|Ns'],
    N3 := N1+N2,
    fib'(N,N2,N3,Ns').
```

Stream consumers
A process which sums the elements of its input stream, was defined in Section 4.3.

The following process reads two vectors of equal length, represented by streams of numbers, and computes their inner product. It will form the building block of a matrix-multiplication program, shown in Section 7.2.

% $ip(Xs,Ys,S)$ ← S is the inner product of Xs and Ys.

[8] The comments associated with concurrent logic programs explain only their logical reading; the reactive aspects are usually explained in the text.

```
ip(Xs,Ys,S) ←
    ip1(Xs,Ys,0,S).

ip1([ ],[ ],P,S) ← P=S.
ip1([X|Xs],[Y|Ys],P,S) ←
    P' := P + X*Y,
    ip1(Xs,Ys,P',S).
```

Stream transducers

The following process multiplies its input stream by some integer, to produce an output stream:

% *multiply(In,N,Out)* ←
  Out is the stream resulting from multiplying each element of the stream *In* by *N*.

```
multiply([ ],N,Out) ← Out=[ ].
multiply([X|In],N,Out) ← Out = [Y|Out'],
    Y := X * N,
    multiply(In,N,Out').
```

The following transducer filters all multiple of an integer from a stream. It is a building block of the parallel Sieve of Eratosthenes, shown in Section 7.2.

% *filter(In,P,Out)* ←
  Out is the stream resulting from deleting all multiples of *P* from the stream *In*.

```
filter([X|In],P,Out) ← 0 =/= X mod P | Out=[X|Out'],
    filter(In,P,Out').
filter([X|In],P,Out) ← 0 =:= X mod P |
    filter(In,P,Out).
filter([ ],P,Out) ← Out=[ ].
```

A stream distributor

The following stream distributor has one input stream and two output streams. If an input message *send(1,X)* is received *X* is sent on the first output stream, and if *send(2,X)* is received *X* it sent on the second output stream. A variant of this program is used in the *msg* message sending system shown in Section 7.2.

% *distribute(In,Out$_1$,Out$_2$)* ←
  In is a stream of elements of the form *send(1,_)* and *send(2,_)*. *Out$_1$* is the stream of *X*'s such that *send(1,X)* is in *In*, and *Out$_2$* is the stream of *X*'s such that *send(2,X)* is in *In*.

```
distribute([send(1,X)|In],Out1,Out2) ← Out1=[X|Out1'],
    distribute(In,Out1',Out2).
distribute([send(2,X)|In],Out1,Out2) ← Out2=[X|Out2'],
    distribute(In,Out1,Out2').
distribute([ ],Out1,Out2) ← Out1=[ ], Out2=[ ].
```

A deterministic stream merger

The following process receives two ordered lists of integers, and produces an ordered merge of them. A variant of it is used in the solution to Hamming's problem and in mergesort in Section 7.

% *omerge(In$_1$,In$_2$,Out)* ←
  If *In$_1$* and *In$_2$* are ordered streams of numbers, then *Out* is an ordered merge of *In$_1$* and *In$_2$*.

```
omerge([X|In1],[Y|In2],Out) ← X≤Y | Out=[X|Out'],
    omerge(In1,[Y|In2],Out').
omerge([X|In1],[Y|In2],Out) ← X>Y | Out=[Y|Out'],
    omerge([X|In1],In2,Out').
omerge([ ],In2,Out) ← In2=Out.
omerge(In1,[ ],Out) ← In1=Out.
```

6. Fairness

A nondeterministic stream merger
The following is a nondeterministic stream merger. Its output stream is some order-preserving interleaving of its two input streams.

% *merge*(In_1,In_2,*Out*) ← The stream *Out* is an interleaving of the streams In_1 and In_2.

```
merge([X|In1],In2,Out) ← Out=[X|Out'],
    merge(In1,In2,Out').
merge(In1,[X|In2],Out) ← Out=[X|Out'],
    merge(In1,In2,Out').
merge([ ],In2,Out) ← In2=Out.
merge(In1,[ ],Out) ← In1=Out.
```

The nondeterministic *merge* process thus defined guarantees nothing about the rate in which it serves its two input streams. In particular, if one of the streams is unbounded then it is possible, according to the semantics of FCP(|) defined in Section 4 above, that only elements of that stream will be copied to the output stream. Furthermore, if the merger (or any other process) is executed in parallel with a nonterminating process, e.g. $p \leftarrow p$, then there is no guarantee that it will reduce at all.

A fairness requirement states conditions under which an event that may happen must eventually happen. The purpose of incorporating fairness requirements into the definition of a language is to provide the programmer with confidence that even in the presence of nondeterminism and unbounded computations certain program steps will eventually occur.

In concurrent logic programming it is useful to distinguish two types of fairness: And-fairness and Or-fairness. An *And-fairness* requirement states conditions under which a certain process would eventually be reduced; thus it constrains And-nondeterminism. An *Or-fairness* requirement states conditions under which a certain clause would eventually (not) be taken, thus constraining Or-nondeterminism.

An And-fairness requirement should guarantee, for example, that even in the presence of diverging processes, deterministic stream consumers will eventually read all their stream elements; similarly for producers. However, And-fairness cannot provide such a guarantee for nondeterministic consumers such as stream mergers or interrupt handlers. This is the purpose of Or-fairness requirements. Or-fairness requirements should allow, for example, specifying a fair stream merger and an interrupt handler in the language. Together, And-fairness and Or-fairness requirements should allow one to compose a controlling process and an interruptible process (e.g. in the style of the computation controller and the interrupt-handling meta-interpreter shown in Section 7) and guarantee that the controller process can interrupt the controlled process even if the latter is nonterminating. The following fairness requirements achieve this. Their definition can be skipped without loss of continuity.

And-fairness
For simplicity we restrict the discussion to computations whose initial state has a unit goal. Let P be a program and $c = \langle G;\epsilon \rangle, \ldots$ be a computation from the unit goal G. Let b be the maximal number of atoms in the body of any clause in P. We label goal atoms in the computation with strings in $\{1,\ldots,b\}^*$. The initial goal is labeled with the empty string. Let A be a process labelled s, which is reduced using the clause $A' \leftarrow B_1,\ldots,B_k$. Then each of the new atoms B_i in the new goal is labeled with the string s^i.

We extend the Reduce transition label to contain *p*, the label of the reduced process, and θ, the try substitution restricted to variables of the process *P*, as in Reduce(p,θ). We extend the Fail transition label to contain the label of the failing process *p*, as in Fail(p).

Definition: And-fairness.
A computation *c* is ***And-fair*** if there is no Reduce(p,θ) transition or Fail(p) transition which is almost always enabled on the states of *c*.[9] ∎

Notes:

1) Since the state of a process changes in a monotonic way (it can only be instantiated further), if Reduce(p,θ) or Fail(p) are infinitely often enabled, they are also almost always enabled, hence there is no distinction in this case between weak fairness (also called justice) and strong fairness [50,142].
2) We have defined the fairness condition by ruling out certain computations allowed by the transition system. An alternative approach is to define a transition system which generates only fair computations to begin with. The approach of Costa and Stirling [32] to weak fairness in CCS can be applied here as well.

Or-fairness
In programs implementing stream merging and interrupt handling, complete freedom in clause selection would result in undesirable behaviors: a merger can ignore one of its input streams indefinitely; a process may ignore a message on its interrupt stream for arbitrarily long. Several approaches to constraining clause selection in such programs were suggested; none of them seems completely satisfactory.

A global fairness requirement, which states that all clauses in a program satisfying certain conditions should be used eventually seems unreasonable, because of the dynamic nature of processes, and the fact that multiple processes may share the same set of clauses. Therefore approaches which specify conditions on the selection of a clause by a process were pursued.

One approach is to impose preference on clauses, and require the Reduce transition to select the most preferred clause among the applicable ones [160]. Assuming such a preference, specified by textual order, the following program implements a fair merger. It achieves fairness by switching the two input streams when an element from the preferred stream is read:

```
merge([X|In1],In2,Out) ← Out=[X|Out'],
    merge(In2,In1,Out').
merge(In1,[X|In2],Out) ← Out=[X|Out'],
    merge(In1,In2,Out').
merge([ ],In,Out) ← In=Out.
merge(In,[ ],Out) ← In=Out.
```

A process $p(\ldots,Is)$ that responds to interrupts on the stream *Is* can be d [illegible] by placing the clause testing for an interrupt first:

```
p(...,[I|Is]) ← interrupt_handler(...,[I|Is]).
... other clauses for p...
```

Although simple to define operationally, strict preferences are problematic. From a methodological point of view, they destroy the clause-wise modularity of programs. This may suggest awkward programming techniques (*à la* red cuts in Prolog [171]), and make the life of program analysers, transformers and compilers more difficult. From an implementation point of view, preferences require strict synchronization, since a second clause can be selected only if nothing happens that may enable the first one. In fact, a correct distributed implementation of strict preferences may require locking all the variables involved in the reduction of the first clause while attempting to

[9] i.e. enabled on all but a finite number of the states of *c*.

reduce with the second clause. A weaker notion of preference seems more desirable, but it is not clear how it should be defined.

Another approach is to use explicit conditions on clauses. Assume a guard primitive *var*(X), which succeeds if X is a variable, and fails otherwise. Using *var*, a fair merger can be written as follows, with base cases as above:

```
merge([X|In1],In2,Out) ← Out=[X|Out'],
    merge(In2,In1,Out').
merge(In1,[X|In2],Out) ← var(In1) | Out=[X|Out'],
    merge(In1,In2,Out').
```

A process $p(\ldots,Is)$ which is sensitive to interrupts on Is can be defined by adding to all its clauses which do not serve the interrupt the test *var*(Is). The use of *var* to achieve Or-fairness has been first proposed by Kusalik [104]. The *var* test approach is better than preferences since it does not destroy clause-wise modularity, and has no effect when not used. Its drawback is that *var* seems too strong a tool for this purpose. From a methodological point of view, it offers opportunities for abuse. From an implementation point of view, *var*, like preferences, implies tight synchronization. To implement correctly an interrupt-sensitive process thus defined, the Is variable has to be locked whenever a reduction of the processes using a clause with a *var*(Is) test is attempted. If there are hundreds or thousands of such processes, all sharing the same interrupt stream Is, this would be prohibitively expensive.

For that reason a weaker primitive, called *unknown*(X), was defined [191]. Intuitively, *unknown* is similar to *var*, except that its definition allows *unknown*(X) to 'ignore' for some finite, but bounded, amount of time the fact that X was instantiated. For example, if X is instantiated to a in processor P_1, *unknown*(X) can succeed after that time in another processor P_2. However, the fact that X has a value should eventually reach P_2, preventing *unknown*(X) tests from succeeding thereafter.

In our interleaving based transition system, this intuitive definition is formalized as follows. *unknown*(X) behaves like *var*(X), except that it may succeed only a finite number of times after X becomes a non-variable. In other words, if in a computation a variable X is instantiated to a non-variable term, then the computation *does* not have *infinitely* many transitions in which the check of the guard predicate *unknown*(X) succeeds. Using *unknown* instead of *var* in the above programs would achieve the desired effect: the merger would be fair, and the process p would eventually respond to an interrupt. This is achieved without heavy synchronization costs, and without giving the programmer too powerful a tool, since *unknown* (X) succeeding *does not* imply that X is presently not instantiated. In a language without atomic variables (see the discussion of Flat GHC in Section 10), the difference between *var* and *unknown* is immaterial.

Note that unlike the other guard test predicates introduced, the success of the *var* and *unknown* primitives is defined operationally, without reference to the notion of truth.

7. Advanced Concurrent Logic Programming Techniques

The power of concurrent logic programming languages comes from the wide range of concurrent programming techniques they support. To convey this, we have assembled a range of FCP(|) programs which demonstrate these techniques.

7.1 Static process networks

Processes operating on streams can be composed into networks. This section shows two examples of a static process network.

A static network of stream transducers: a solution to Hamming's problem

The following program [42] solves the so-called Hamming's problem [39]: generate an ordered stream of all numbers of the form $2^i3^j5^k$ without repetition.

% *hamming*(*Xs*) ← *Xs* is the ordered stream of all numbers of the form $2^i3^j5^k$.

```
hamming(Xs) ←
     multiply([1|Xs],2,X2),
     multiply([1|Xs],3,X3),
     multiply([1|Xs],5,X5),
     omerge'(X2,X3,X23),
     omerge'(X5,X23,Xs).
```

where *omerge'* is a variant of *omerge* shown in Section 5, which removes duplicates from its input stream.

% *omerge'*(In_1,In_2,*Out*) ←
If In_1 and In_2 are ordered streams of numbers, then *Out* is an ordered merge of In_1 and In_2 with duplicates removed.

```
omerge'([X|In1],[X|In2],Out) ← Out = [X|Out'],
     omerge'(In1,In2,Out').
omerge'([X|In1],[Y|In2],Out) ← X<Y | Out=[X|Out'],
     omerge'(In1,[Y|In2],Out').
omerge'([X|In1],[Y|In2],Out) ← X>Y | Out=[Y|Out'],
     omerge'([X|In1],In2,Out').
omerge'([ ],In2,Out) ← In2=Out.
omerge'(In1,[ ],Out) ← In1=Out.
```

multiply was defined in Section 5.

A static network: the MSG message sending system

The *msg* process network is a simple message sending system for two computer terminals. Input from each of the keyboards K_1 and K_2 is a stream of messages, including messages of the form *message*(X). Every message M on K_1 is echoed on S_1 as *1*:M. In addition, a message of the form *message*(X) on K_1 is also echoed on S_2 as *1*:*message*(X). Similarly for messages on K_2 [93]. The program uses the *merge* process and a variant of the *distribute* process defined in Section 5.

% *msg*(*K1*,*S1*,*K2*,*S2*) ←
S_1 is an interleaving of *1*:X such that X is in K_1 and *2*:*message*(X) such that *message*(X) is in K_2. Similarly S_2 is an interleaving of *2*:X such that X is in K_2 and *1*:*message*(X) such that *message*(X) is in K_1.

```
msg(K1,S1,K2,S2) ←
     distribute(1,K1,K11,K12),
     distribute(2,K2,K22,K21),
     merge(K11,K21,S1),
     merge(K22,K12,S2).

distribute(Id,[message(X)|In],Out1,Out2) ←
     Out1=[Id:message(X)|Out1'],
     Out2=[Id:message(X)|Out2'],
     distribute(In,Out1',Out2').
distribute(Id,[X|In],Out1,Out2) ←
     X≠message(_) |
     Out1=[Id:X|Out1'], .
```

```
        distribute(In,Out1',Out2).
    distribute([ ],Out1,Out2) ← Out1=[ ], Out2=[ ].
```

7.2 Dynamic process networks

We show examples of dynamic process networks of various topologies. They are dynamic since their size depends on their input. Dynamic process networks which solve algorithmic problems typically exhibit two phases of operation. A spawning phase, in which the process network is spawned, and a "systolic" phase [103], in which the processes in the network perform both local computations and communication. It is interesting to note that many of the concurrent logic programs shown below, which implement "systolic"-like parallel algorithms, are almost identical, as logic programs, to Prolog programs which implement the corresponding sequential algorithms. More on the relation between systolic algorithms and concurrent logic programming can be found in [162].

Process pipes: linear process networks

The following program is a parallel implementation of the Sieve of Eratosthenes [163]. It consists of a process generating all integers in the desired range, and a set of filter processes, one per prime number found, which erase multiples of their prime from the remaining stream. This program overlaps the dynamic construction of the process network with the computation of the result. Its network consists of a dynamic linear pipeline of transducers. It uses the *integers* and *filter* processes defined in Section 5.

% *primes(N,Ps)* ← *Ps* are all the primes up to *N*.

```
    primes(N,Ps) ←
        integers(2,N,Ns), sift(Ns,Ps).
```

% *sift(Ns,Ps)* ← *Ps* are the numbers in *Ns* which are prime relative to their predecessors.

```
    sift([P|Ns],Ps) ← Ps=[P|Ps'],
        filter(Ns,P,Ns'), sift(Ns1,Ps').
    sift([ ],Ps) ← Ps=[ ].
```

Mergesort

The following program implements a parallel mergesort algorithm. Given a list of length N of (possibly singleton) sorted lists, it forms a pipeline of length $\log_2 N$ of *msort* processes. Each stage in the pipeline performs a pairwise order-preserving merge of the sublists, using the *omerge* procedure defined in Section 5. Each stage doubles the length of the sublists and divides their number by approximately 2. Using $\log_2 N$ processors, this program can sort an N elements list in time $O(N)$. See [189] for further discussion of the complexity of concurrent logic programs.

% *mergesort(In,Out)* ←
 If *In* is a list of ordered lists of numbers then *Out* is a sorted list of these numbers.

```
    mergesort([ ],Ys) ← Ys=[ ].
    mergesort([Xs],Ys) ← Xs=Ys.
    mergesort([X1,X2|Xs],Ys) ←
        msort([X1,X2|Xs],Zs),
        mergesort(Zs,Ys).

    msort([X1,X2|Xs],Ys) ← Ys=[Y|Ys'],
        omerge(X1,X2,Y),
        msort(Xs,Ys').
    msort([X],Ys) ← Ys=[X].
    msort([ ],Ys) ← Ys=[ ].
```

Vector-matrix multiplication

A linear array of *ip* processes can multiply a matrix, represented by a list of vectors, by a vector. It uses the *ip* process defined in Section 5.

% *vm(Xv,Ym,Zv)* ← multiplying the vector *Xv* by the matrix *Ym* gives the vector *Zv*.

```
vm(_,[ ],Zv) ← Zv=[ ].
vm(Xv,[Yv|Ym],Zv) ← Zv=[Z|Zv'],
     ip(Xv,Yv,Z), vm(Xv,Ym,Zv').
```

Process trees

The following program merges a list of streams into one stream, by creating a balanced tree of binary merge processes. It uses the merge process defined in Section 6.

% *merger(In,Out)* ← *Out* is the merge of the list of streams *In*.

```
merger([Xs1,Xs2|In],Out) ←
     merge_layer([Xs1,Xs2|In],Out'),
     merger(Out',Out).
merger([Xs],Out) ← Xs=Out.

merge_layer([Xs1,Xs2|In],Out) ← Out=[Ys|Out'],
     merge(Xs1,Xs2,Ys),
     merge_layer(In,Out').
merge_layer([Xs],Out) ← Out=[Xs].
merge_layer([ ],Out) ← Out=[ ].
```

Note that this program operates correctly only if the complete list of streams is given, since it emits elements from the root of the tree only after the construction of the tree is completed. If the list is given incrementally, i.e. it is actually a stream of streams, then a different approach is needed. One naive solution is to create an unbalanced tree incrementally, using the following program.

```
merger1([Xs|In],Out) ←
     merge(Xs,Out',Out),
     merger1(In,Out').
merger1([ ],Out) ← Out=[ ].
```

The program builds a linear tree top down. At each point in its construction, elements of the input streams already connected to *merge* processes can reach the root of the tree.

More sophisticated balanced merge trees can be constructed, which support the dynamic addition and deletion of merged streams, using the concept of two-three trees, as shown in this section below.

Process arrays

Two matrices can be multiplied by an array of *ip* processes, each computing the inner product of the appropriate row and a column of the matrices. We assume that the second matrix is already transposed.

% *mm(Xm,Ym,Zm)* ←
  Zm is the result of multiplying the matrix *Xm* with the transposed matrix *Ym*.

```
mm([ ],_,Zm) ← Zm=[ ]).
mm([Xv|Xm],Ym,Zm) ← Zm=[Zv|Zm']
     vm(Xv,Ym,Zv), mm(Xm,Ym,Zm').
```

The program uses the *vm* process defined above.

The behavior of the *mm* program is reminiscent of the well-known systolic algorithm for multiplying two matrices [103]. However, there are two differences. First, the process network is created dynamically, to fit the size of the input matrices. This suggested the name[10] "soft-systolic" to these kinds of software-oriented systolic algorithms, in contrast with the classical hardware-oriented "hard-systolic" algorithms. Another difference is that the program, as specified, does not pipeline the matrices along the connections between the *ip* processes, but rather "broadcasts" each row and column to all processes requiring it. The program, however, can be easily modified to do the pipelining. See [163,189] for further discussions of the subject.

To achieve pipelining, the *mm* program has to be modified to form direct connections between adjacent *ip* processes. A similar goal is achieved by the following *torus* program. Given a matrix *Array, represented as a list of list of values, it spawns a torus cell* processes, each with one value and with communication links to adjacent *cell* processes. The array is augmented with end-round connections, to form a torus process network. This program schema (cliché) has several applications, including array relaxation and the like.

```
torus(Array,...) ←
    torus'(Array,Bottoms,Tops,...),
    Bottoms=Tops.

torus'([Row|Array],Bottoms,Tops,...) ←
    row(Row,Left,Right,Bottoms,Middles,...),
    Left=Right,
    torus'(Array,Middles,Tops,...).
torus'([ ],Bottoms,Tops) ←
    Bottoms=Tops.

row([Element|Row],Left,Right,[Bottom|Bs],[Top|Ts],...) ←
    cell(Element,Left,Middle,Bottom,Top,...),
    row(Row,Middle,Right,Bs,Ts,...).
row([ ],Left,Right,[ ],[ ],...) ←
    Left=Right.

cell(Element,Left,Right,Bottom,Top,...) ←
    An application specific program.
```

The layered stream method

Search problems that are amenable to depth-first search have elegant and efficient solutions in Prolog. Assume that the solution to the search problem is in the form of a list of elements, which satisfy some consistency criterion. The incremental construction of a solution in Prolog often relies on its backtracking mechanism, where forward computation consists of extending some prefix of a solution, and backtracking occurs when it is discovered that the prefix cannot be extended further, either because of inconsistencies or because it is a complete solution and additional solutions are required.

The layered stream data structure, proposed by Okumura and Matsumoto [139], allows an incremental and parallel construction of solutions to search problems without relying on a Prolog-like backtracking mechanism. A layered stream represents a set of lists sharing a common head as the cross-product of the head and the list of tails. The list of tails in turn can be represented by a layered stream. For example the lists *[1,2,4]*, *[1,2,8]*, *[1,3,9]* are represented by the layered stream *1 * [[2,4], [2,8], [3,9]]*, and also by the layered stream *1 * [2*[[4],[8]], [3,9]]*. The function symbol '*' is used for mnemonic purposes, but of course any other function symbol would do. The product $X*[\,]$ represents the empty set of lists, and the product $X*true$ represents X.

The following programming methodology is associated with the layered stream. Suppose the

[10] Due to Vijay A. Saraswat.

problem is to find values for N variables ranging over some finite domain of values V. $N*|V|$ processes are initially created, one for each possible value of each variable. Denote the process associated with value v of variable n by $p_{n,v}$. Each $p_{n,v}$ process receives an input stream of partial solutions which consist of an assignment to variables $1,\ldots,n-1$, and produces an output stream of partial solutions obtained by extending each input assignment with the assignment of v to variable n, provided the resulting assignment is consistent. That is, given the input partial solution $v_1,\ldots,v_{n-1}$, if the extended partial solution $v_1,\ldots,v_{n-1},v$ is consistent, it is output. Otherwise it *is not*.

The layered stream data structure allows all processes $p_{n,v}$, $v \in V$, to share the same input partial solutions, thus saving space and hence also time. It allows the pipelining of partial solutions, hence increases the available parallelism. An example of a search program using a layered stream is the four queens program shown below [139]. The program easily generalizes to N queens by replacing the explicit construction of the filter processes by iterative procedures to do so.

% *four_queens(Qs)* ←
  Qs is a layered stream of all legal assignments of four queens on a 4×4 board.

```
four_queens(Qs) ←
     queen(true,Qs1), queen(Qs1,Qs2), queen(Qs2,Qs3), queen(Qs3,Qs).
```

% *queen(In,Out)* ←
  If *In* represents the set of legal assignments of N queens on a 4×4 board, *Out* represents the set of legal assignments of $N+1$ queens on a 4×4 board.

```
queen(In,Out) ←
     filter(In,1,1,Out1), filter(In,2,1,Out2), filter(In,3,1,Out3), filter(In,4,1,Out4),
     Out=[1*Out1,2*Out2,3*Out3,4*Out4].
```

% *filter(In,I,D,Out)* ←
  If *In* is a set of assignments of N queens to consecutive columns, *Out* is the set of assignments of $N+1$ queens obtained as follows: Extend each assignment in *In* with a queen on the next column and on row I. If the added queen does not interfere with the previous queens, incorporate the extended assignment in *Out*.

```
filter(true,_,_,Out) ← Out=true.
filter([ ],_,_,Out) ← Out=[ ].
filter([I*_|In],I,D,Out) ← filter(In,I,D,Out).                 % Same row
filter([J*_|In],I,D,Out) ← D=:=abs(I-J) | filter(In,I,D,Out).  % Same diagonal
filter([J*In'|In],I,D,Out) ← I≠J, D\=:=abs(I-J) |              % No interference
     D':=D+1,
     filter(In',I,D',Out'),
     filter(In,I,D,Out''),
     Out=[J*Out'|Out''].
```

The answer obtained from the goal *four_queens(Qs)* is the following layered stream:

```
Qs = [1 * [3 * [ ], 4 * [2 * [ ]]],
      2 * [4 * [1 * [3 * true]]],
      3 * [1 * [4 * [2 * true]]],
      4 * [1 * [3 * [ ]], 2 * [ ]]]
```

which represents the list of lists:

```
Qs = [ [2,4,1,3], [3,1,4,2] ].
```

A comparison of the sequential and parallel performance of this program with other concurrent logic programs and Prolog programs for the N-queens problem is given by Tick [193].

7.3 Incomplete message protocols

Incomplete message protocols were reviewed in Section 4.4. Here we show several examples of their application. The first application is monitors. A monitor is a process that maintains some local state, and serves requests to inspect and/or modify the state. It is called so since its function is similar to Hoare's original concept of a monitor [85]. Clients of a monitor are typically connected to it via a merge network, and communicate with it using incomplete messages.

Perhaps the simplest monitor is the *counter*, which maintains a local counter, and reponds to the messages *clear*, *add*, and *read*(X), the last of which is an incomplete message.

% *counter*(*In*) ←
In is a stream of *clear*, *add*, and *read*(X) such that X is the number of *add*'s since the most recent *clear*.

```
counter(In) ← counter'(In,0).

counter'([clear|In],C) ← counter'(In,0).
counter'([add|In],C) ← C' := C+1, counter'(In,C').
counter'([read(X)|In],C) ← X=C, counter'(In,C).
counter'([ ],C).
```

The client of a counter who wishes the know its value sends it the message *read*(X), and waits for X to be instantiated.

Shared queues

A more sophisticated monitor is the following queue process. It serves requests of the form *enqueue*(X) and *dequeue*(X), by unifying the arguments of corresponding enqueue and dequeue requests, and maintaining arguments of superfluous requests. While a list is a natural data structure for representing a stack, a difference-list is most convenient for representing a queue. The arguments of superfluous requests are maintained in a difference-list data-structure, which is positive if it received more enqueue than dequeue requests, empty if the number of requests received of each type were equal, and negative otherwise. A difference-list, explained in Section 2.2, is a common data-structure both in Prolog and in concurrent logic programming languages [27,160,171].

% *queue*(*In*) ←
In is a stream of *enqueue*(X) and *dequeue*(X), for which the list of X's such that *enqueue*(X) is in *In* is identical to the list of X's such that *dequeue*(X) is in *In*.

```
queue(In) ←
    queue'(In,Q\Q).

queue'([dequeue(X)|In],H\T) ← H=[X|H'],
    queue'(In,H'\T).
queue'([enqueue(X)|In],H\T) ← T=[X|T'],
    queue'(In,H\T').
queue'([ ],H\T).
```

Another useful monitor is a priority queue. A priority queue has two input streams. One of enqueue requests of the form *enqueue*(X,P), and one of dequeue requests of the form *dequeue*(X). It maintains an internal priority queue, which is a list of the elements enqueued but not dequeued, sorted by their priority. It serves a dequeue request only if the queue is non-empty.

% *pqueue*(*Es*,*Ds*) ←
Es is a list of *enqueue*(X,P) and *Ds* is a list of *dequeue*(Y) for the which the corresponding multisets of X's and Y's are the same, and if *dequeue*(X) precedes *dequeue*(Y) in *Ds* then either *enqueue*(X,PX) precedes *enqueue*(Y,PY) in *Es* or *enqueue*(Y,PY) precedes *enqueue*(X,PX) in *Es* and $PY<PX$.

```
pqueue(Es,Ds) ←
    pqueue'(Es,Ds,[ ]).
pqueue'([enqueue(X,P)|Es],Ds,Q) ←
    insert(X,P,Q,Q'),
    pqueue'(Es,Ds,Q').
pqueue'(Es,[dequeue(X)|Ds],[(Y,P)|Q]) ← X=Y,
    pqueue'(Ed,Ds,Q).
pqueue'([ ],[ ],Q).
insert(X,P,[ ],Q) ← Q=[(X,P)].
insert(X,P,[(X',P')|Q],Q') ←
    P < P' | Q'=[(X,P),(X',P')|Q].
insert(X,P,[(X',P')|Q],Q') ←
    P ≥ P' | Q'=[(X',P')|Q''],
    insert(X,P,Q,Q'').
```

Merge trees

Another application of incomplete message protocols is network reconfiguration. We show here a simple example of a dynamic two-three merge tree. A process *p(Xs,...)* with a stream *Xs* to the tree can create a new process *q(Ys,...)* with a stream *Ys*, and join *Ys* to the tree by sending down *Xs* the message *merge(Ys)*. For example:

```
p(Xs,...) ←
    Xs=[merge(Ys)|Xs'],
    p(Xs',...),
    q(Ys,...).
```

A balanced tree of merge processes capable of handling such messages can be composed of binary and ternary mergers, defined below. We show the clauses which handle messages on the first input stream only. Handling messages on the other streams is done by similar clauses of the same procedure. Note how a *merge2* process that receives a *merge(X)* message turns into a *merge3*, and a *merge3* process that receives such a message turns into two *merge2*'s, and sends up another *merge* message.

```
merge2([X|Xs],Ys,Zs) ←
    X≠merge(_) | Zs=[X|Zs'],
    merge2(Xs,Ys,Zs').
merge2([merge(Ws)|Xs],Ys,Zs) ←
    merge3(Ws,Xs,Ys,Zs).
merge2([ ],[ ],Zs) ← Zs=[ ].
merge3([W|Ws],Xs,Ys,Zs) ←
    W≠merge(_) | Zs=[W|Zs'],
    merge3(Ws,Xs,Ys,Zs').
merge3([merge(Ws1)|Ws],Xs,Ys,Zs) ←
    Zs=[merge(Zs1)|Zs'],
    merge2(Ws1,Ws,Zs1),
    merge2(Xs,Ys,Zs').
merge3([ ],[ ],[ ],Zs) ← Zs=[ ].
```

The merge tree described grows dynamically, but does not shrink in a balanced way. An extension based on a distributed variant of the two-three tree deletion algorithm is described in [167].

The bounded-buffer protocol

In several situations it is desirable to allow the reader of a stream some degree of control over its writer. Examples are when the reader is much slower than the writer, and when only some prefix of the produced stream is required, but its size can only be determined by the reader at runtime. The bounded-buffer protocol [177] employs difference lists and incomplete messages to realize this kind of control.

The idea of the bounded-buffer protocol is simple: the controlling reader process maintains a difference-list $H\backslash T$ of incomplete messages, say of the form *message*(X), where X is a variable. The difference-list represents the "buffer". After the buffer is initialized to a list of n incomplete messages, the reader operates as follows: when it is ready to process the next message, it waits until the first element in the buffer is known, i.e. $H=[message(X)|H']$, where X is known, dequeues it, and enqueues an incomplete message, *message*(X'), at the tail of the difference list. When it does not desire to receive any further messages it unifies the tail with nil. What to do in such a case with the messages pending in the buffer is application dependent.

The producer is given initially the head of the difference list as its input stream. It then operates as follows. It waits until its input stream has the message *message*(X), produces the next element, unifying it with X, and iterates with the tail of its input stream. It terminates when its input stream is nil.

Schematic programs for the producer and the reader are shown below.

```
bounded_buffer_network(...) ←
    buffer(n,H\T),
    read(H\T,...),
    produce(H,...).

% buffer(N,H\T) ←
    H\T is a difference list of message(_) of size N.

buffer(0,H\T) ← H=T.
buffer(N,H\T) ←
    N>0 |
    N':=N-1, T=[message(_)|T'],
    buffer(N',H\T').

read([message(X)|H]\T,...) ←
    known(X),
    ...I want more X's... |
    T=[message(_)|T'],
    ... process X...,
    read(H\T',...).
read(H\T,...) ←
    ...I don't want more X's... |
    T=[ ],
    ...process remaining messages in H....

produce([message(X)|In],...) ←
    ...produce X...,
    produce(In,...).
produce([ ],...).
```

Several variations on this protocol are possible. For example, it is not necessary for the reader to maintain a fixed size buffer: it can increase or decrease the size of the buffer if it so desires. It is not necessary to synchronize on every message: a more efficient protocol might be to produce k stream elements per incomplete message, or to provide a parameter in the incomplete message, specifying how many more elements to produce. Finally, it is possible for the incomplete message to be simply a variable, rather than a term containing a variable.

7.4 Mutual exclusion protocols

Mutual exclusion can be achieved in FCP(|) using the following mechanism. The set of processes participating in the mutual exclusion protocol are connected via a merge network into a *mutex* process. A single round mutual exclusion protocol is as follows: all processes competing for lock send a *lock(Reply)* incomplete message to *mutex*. *mutex* grants the first *lock* request received by unifying *Reply = granted*, and denies the other requests by unifying *Reply = denied*. It is defined as follows:

% *mutex(In)* ← *In* is a list containing one *lock(granted)* followed by zero or more *lock(denied)*.

```
mutex([lock(Reply)|In]) ← Reply=granted,
    mutex'(In).
mutex'([lock(Reply)|In]) ← Reply=denied,
    mutex'(In).
mutex'([ ]).
```

The single-round mutual exclusion protocol can be used to simulate CSP with input guards [87]. A simulation of CSP with both input and output guards is discussed in Section 14.

A multiple round mutual exclusion protocol is only slightly more complex. Instead of simple back-communication, it uses a three stage dialogue: the process requests the lock, then *mutex* grants it, then the process releases the lock, and *mutex* serves the next lock request. Processes competing for permission send *lock(Reply)* as before. *mutex* answers the first by *Reply = granted(Done)*, and waits for *Done = done*. When the process to which the lock was granted ends it critical operation, it releases the lock by unifying *Done = done*. *mutex* then grants the next lock, and so on. If the merge network is fair, and every process that is granted a lock eventually releases it, then every lock request will eventually be granted.

The definition of the multiple-round *mutex* process is as follows. Its trivial logical reading indicates that its interest lies in its reactive aspects only.

% *mutex(In)* ← *In* is a list of *lock(granted(done))*.

```
mutex(In) ←
    mutex'(In,done).
mutex'([lock(Reply)|In],done) ← Reply=granted(Done),
    mutex'(In,Done).
mutex'([ ],_).
```

A program schema for a perpetual process *p* participating in a multiple-round mutual exclusion protocol is shown below. We assume that initially its first argument is a stream merged to *mutex*; other arguments are application specific.

```
p(ToMutex,...) ← p_request(done,ToMutex,...).
p_request(done,ToMutex,...) ← ToMutex=[lock(Reply)|ToMutex'],
    p_wait(Reply,ToMutex',...).
p_wait(granted(Done),ToMutex',...) ←
    ... do critical operation; when done, unify Done=done...
    p_request(Done,ToMutex,...).
```

7.5 Short-circuit protocols for distributed termination, quiescence detection, and distributed event-driven simulation

The problems of distributed termination detection and quiescence detection have received considerable attention [15,16,40,51,105,126]. In concurrent logic programming, these problems have very elegant solutions, using the short-circuit protocol. The protocol is originally due to Takeuchi [175], and was later extended by Weinbaum and Shapiro [208] and Saraswat *et al.* [158]; we largely follow [158] in the following discussion. The underlying behavior of implementations of this protocol are closely related to that of distributed termination and quiescence detection algorithms based on distributed counters [105,126]. We do not know of algorithms for distributed event-driven simulation corresponding to the one based on the short-circuit.

Distributed termination detection

The idea of the short circuit for termination detection is as follows. Call the computation whose termination should be detected the underlying computation, and the program it executes the underlying program. Augment each process participating in the underlying computation with two additional arguments, called *Left* and *Right*. For readability, these arguments are typically packed in one term using the '–' infix function symbol, as in *Left–Right*. The pair is called a *switch*. It is *closed* if *Left=Right*, *open* otherwise.

Initially, connect all processes in a chain, by unifying *Left* of the i^{th} process with *Right* of the $i^{th}+1$ process. The *Right* of the first process and the *Left* of the last process are called the ends of the short-circuit. For *n* processes, the chain contains *n* open switches.

Each process in a computation operates as follows. If it halts it unifies its *Left* and *Right* variables. If it iterates it leaves them unchanged. If it creates *n* new processes, it extends the short-circuit by *n*–1 intermediate links. This behavior is achieved by transforming the clauses of the underlying program along the following schema, where '...' denotes underlying program arguments.

```
p(...).          ⇒ p(...,L-R) ← L=R.
p(...) ←         ⇒ p(...,L-R) ←
    p'(...).          p'(...,L-R).
p(...) ←         ⇒ p(...,L-R) ←
    p1(...),          p1(...,L-X1),
    p2(...),          p2(...,X1-X2),
    ⋮                 ⋮,
    pn(...).          pn(...,Xn-1-R).
```

In FCP(|), a correct use of the short-circuit requires threading it to the equality goal atoms in a special way. If the underlying program has a body atom *T1=T2*, the transformed program should have the atom (*Left,T1*)=(*Right,T2*) for the appropriate switch variables *Left* and *Right*, so that the switch would not close before the underlying unification completes.

The invariant of the short circuit under this behavior is that the number of open switches is identical to the number of processes in the computation. In particular, all switches are closed, which implies that the two ends of the initial chain are identical, if and only if all processes in the computation have terminated, which is a stable property.

Any process wishing to detect that the computation has terminated is given the initial ends of the short circuit. Assume the termination detecting process is called *halted*(*Left–Right*,...). It can be implemented in FCP(|) in two ways:

```
halted(X-X,...) ← ... report termination ...
```

or:

```
halted(Left-Right,...) ←
    Left=done, wait_for_done(Right,...)
wait_for_done(done,...) ← ... report termination ...
```

Distributed phased termination detection

Some computations consist of phases, where a process is allowed to begin computations of the next phase only if all processes have completed the previous phase [130,208]. The short circuit can be generalized to achieve phased termination detection as well. Instead of having one short circuit, a stream of short circuits is threaded through the underlying computation. Each process is augmented with a *Left-Right* switch as before, and with the original left and right ends of the circuit, *LeftEnd-RightEnd*. However, instead of unifying *Left* and *Right* upon termination, it treats *Left* and *Right* as streams. At the termination of a phase it unifies the head of *Left* with the head of *Right*. Following that, it waits for the heads of *LeftEnd* and and *RightEnd* to be identical before it proceeds with the next phase. This is achieved by the following iterative schema:

```
p(Left-Right,LeftEnd-RightEnd,...) ←
    ... do computation of this phase, when done, do the following ...,
    Left=[X|Left'],
    Right=[X|Right'],
    p_wait(Left'-Right',LeftEnd-RightEnd,...).
p_wait(Left-Right,[X|LeftEnd]-[X|RightEnd],...) ←
    p(Left-Right,LeftEnd-RightEnd,...).
```

Process creation and termination is handled as before.

Note that the solution is completely symmetric. There is no centralized process that detects the termination of a phase; rather, the ends of the circuit are distributed to all processes, and each of them detects the end of phase independently.

Quiescence detection

Consider a network of processes participating in some underlying computation by exchanging messages. The computation begins by a designated process, which sends one or more messages to other processes. Each process that receives a message sends out zero or more messages in response. No process spontaneously initiates new messages. The computation ends when all messages sent have been received, and no new response messages need to be generated. Normally, this results in a deadlock of the underlying computation. We would like to augment the underlying computation, so that instead of deadlocking it would report quiescence [15,16].

This can be achieved by another variant of the short circuit protocol. In this variant, switches are embedded in messages, rather than in processes. The initial set of messages are threaded with a short circuit, as was the initial set of processes above. A process wishing to detect quiescence holds the ends of the circuit and waits for them to become identical. Each message in the underlying computation is augmented with a switch, and each process in the underlying computation is augmented to obey the following protocol. When it absorbs a message, i.e. receives a message without generating any additional messages in response, it closes the switch in the message. When it sends one message in response to a message, it includes in the outgoing message the switch of the incoming message, intact. When it generates n response messages, $n>1$, it extends the switch into n switches, and embeds the new switches in the outgoing messages.

For simplicity, assume that each process has one input stream and one output stream of messages. Mergers and distributers can be attached to these streams if necessary. The schema of an augmented process is:

```
p([m(Left-Right,...)|In],Out,...) ←          % Absorb a message
    Left=Right,
    ...,
    p(In,Out,...).
p([m(Left-Right,...)|In],Out,...) ←          % Send one message
    ...,
    Out=[m'(Left-Right,...)|Out'],
        p(In,Out',...).
    p([m(Left-Right,...)|In],Out,...) ←          % Send many messages
        ...,
        Out=[m_1(Left-Middle_1,...),
             m_2(Middle_1-Middle_2,...),
             ...,
             m_n(Middle_{n-1}-Right,...)|Out'],
        p(In,Out,...).
```

The invariant of this protocol is that the number of open switches is the number of message sent (or to be sent) but not yet received. When this number reaches 0, the short circuit is closed, and quiescence can be reported. Note that this protocol requires that each message has at most one receiver. To achieve broadcasting the underlying program must be augmented with explicit distributors, which follow the same protocol.

<u>Distributed event-driven simulation</u>
One interesting application of the above techniques is distributed event-driven simulation. In event-driven simulation, in contrast to clock-driven simulation, only changes are communicated between the components participating in the simulation. This is especially important in hardware simulation, where very often only a small percentage of the simulated device is active at any given time.

An event-driven simulation is phased, since changes which occur in the next phase can be reliably communicated only when all changes related to the previous phase have been received. The method for phased termination detection, using the stream of short circuits described above could be used, except that it requires every process participating in the simulation to be activated in every cycle in order to close its segment of the short-circuit, contrary to our goal. Our solution is a combination of the quiescence detection and phased termination detection techniques.

Each message is augmented with a stream of switches and the ends of the short circuit; these are the same data structures each process is augmented with in phased computation detection. In addition, each process is augmented to behave as follows. In each phase, the process treats the first message it receives as follows. It closes the head of its switch, and keeps the tail of the switch and the circuit's ends. It then waits either for the head of the circuit's ends to close, or for additional messages. (Note that only one of them can occur, since the head of the circuit's ends close only after all messages sent in this phase have been received.) If an additional message is received, it closes the message's entire switch, after verifyng that the message-circuit's ends are identical to the ones it maintains (this is necessary to ensure that the message belongs to the current phase; otherwise it is possible that this message was sent by a process that has already detected the end of the current phase and sent a message belonging to the next phase). If the head of the circuit's ends close, it sends zero or more messages, as required by the underlying computation, each with a segment of the tail of the switch, and with the tail of the circuit's ends.

A schema of such a process follows. For simplicity a process which sends out one message per phase is shown.

```
p_dormant([m(Left-Right,LeftEnd-RightEnd,...)|In],Out,...) ←   % received first message
    Left=[X|Left'],                                             % acknowledge receipt
    Right=[X|Right'],
    ...,                                                        % process and store message
    p_passive(In,Out,Left'-Right',LeftEnd-RightEnd,...).
p_passive([m(Left1-Right1,LeftEnd-RightEnd,...)|In],           % received additional message
        Out,Left-Right,LeftEnd-RightEnd,...) ←                  % of current phase
    Left1=Right1,                                               % acknowledge receipt
    ...,                                                        % process and store message
    p_passive(In,Out,Left–Right,LeftEnd–RightEnd,...).
p_passive(In,Out,Left–Right,[X|LeftEnd]–[X|RightEnd],...) ←      % detect end of phase
    ...,                                                           % compute outgoing message
    Out=[m(Left–Right,LeftEnd–RightEnd,...)|Out'],
    p_dormant(In,Out',...).
```

The reason for embedding the circuit's ends in messages is efficiency. If the ends were distributed to all processes in the network initially, a process receiving a message after being dormant for some time would have to search for the tail of the end's streams. In the current scheme it receives the updated tails in the message.

More details on this subject can be found in [158,208].

7.6 Object-oriented programming, delegation, and *otherwise*

Concurrent logic programming languages naturally give rise to an object-oriented programming style, where the objects are processes communicating via message streams. Much research was devoted to understanding the relation between classical object-oriented concepts and techniques and the object-oriented style offered by concurrent logic programming [95,169]. For a further discussion of object-oriented programming see Section 21.

One common object-oriented technique is delegation. A process that does not understand a certain message delegates it to another process, who may be better equipped to handle it. Consider a process $p(In,\ldots,Out)$, which receives messages on In. Some messages it handles by itself; others are delegated to the Out stream. If the set of messages it recognizes is simple, say a and b, then p can be coded easily:

```
p([a|In],...,Out) ←
    ..., p(In,...,Out).
p([b|In],...,Out) ←
    ..., p(In,...,Out).
p([X|In],...,Out) ←
    X ≠ a, X ≠ b | Out=[X|Out'],
    p(In,...,Out').
```

However, if the messages are complex, and have arguments which should have specific combinations of values, then the explicit specification of conditions under which the message should be delegated becomes harder. To that effect a new guard primitive, called *otherwise*, is introduced. The operational semantics of *otherwise* is given assuming an ordering on clauses (say textual order). Given a goal atom G, an *otherwise* guard in a clause C succeeds if $try(G,C') = fail$ for every clause C' preceding C.

Using *otherwise*, defaults can be handled easily:

```
p([X|In],...,Out) ←
    otherwise | Out=[X|Out'],
    p(In,...,Out').
```

Otherwise destroys clause-wise modularity, and the explicit formulations of the conditions under which it succeeds is often cumbersome[11]. This is the source of its power, but also an indication that it should not be used excessively. *Otherwise* is best thought of as a primitive exception handling

[11] However, K. Kahn (personal communication) notes that there is a sense in which *otherwise* enables clause modularity. If a procedure needs to specify a default case, as in this example, which applies when all other clauses don't apply, then without *otherwise* it must encode explicitly the negation of the other guards, and should be updated if the other clauses change. However, by encoding the default with *otherwise* there is no textual dependency between the default clause and the other clauses of a procedure.

mechanism, which should be used only to handle exceptions, and not in normal programming practice.

7.7 Enhanced meta-interpreters

A meta-interpreter for a language L is an interpreter for L written in L. If a language has simple meta-interpreters, then one of the most convenient ways to enhance a language, or implement sublanguages, is by starting from a meta-interpreter and enhancing it [60,149,171,178,182]. There can be several meta-interpreters for a language, which differ in what aspects of the execution model they *reify*, i.e. execute themselves, and what aspects they *absorb*, i.e. default to the underlying languages. The most useful type of meta-interpreter in logic programming is the one that reifies goal reduction and absorbs unification.

Another distinction is how the meta-interpreter is composed with the program to be interpreted. One method is to pass a data-structure representing the program as a parameter to the interpreter. This approach is the most flexible, but usually imposes unacceptable runtime overhead. On the other extreme, the meta-interpreter and the program to be interpreted can be bound together at compile time. This may give the most efficient result, especially if source to source transformation techniques, such as partial evaluation, are applied to the combined program (see below). This approach, however, is very inflexible.

The most common approach in logic programming, which is also taken here, is an intermediate one in terms of efficiency and flexibility. The program to be interpreted is compiled in a special way, and an interface to the meta-interpreter is provided. The interface determines which aspects of the computation are absorbed, and hence compiled efficiently, and which are to be reified by the meta-interpreter.

A plain FCP(|) meta-interpreter

We demonstrate the approach for FCP(|). Each clause

$$A \leftarrow G \mid B$$

of the FCP(|) program to be interpreted is transformed into the unit clause

$$clause(A,X) \leftarrow G \mid X = B'.$$

where B' is the conjunction obtained by replacing every goal atom G in B whose predicate is neither *true* nor '=' by the term $goal(G)$.

For example, the *omerge* program is represented by clauses like the following:

```
clause(omerge([X|In1],[Y|In2],Out), B) ← X ≤ Y |
     B=(Out=[X|Out'],goal(omerge(In1,[Y|In2],Out'))).
clause(omerge([ ],In2,Out), B) ← B=(In2=Out).
```

Given such a representation, an FCP(|) meta-interpreter can be written as follows:

% *reduce*(*Goal*) ← *Goal* is reducible using the program defined by the *clause* predicate.

```
reduce(true).                              %1
reduce(X=Y) ← X=Y.                         %2
reduce((A,B)) ← reduce(A), reduce(B).      %3
reduce(goal(A)) ← clause(A,B), reduce(B).  %4
```

The meta-interpreter reifies process termination (clause 1) spawning (clause 3) and reduction (clause 4). Note that the meta-interpreter interpreters the parallel processes (A,B) in parallel, by forking into the two processes *reduce*(A) and *reduce*(B). It absorbs unification (clause 2) by calling FCP(|)'s primitive unification predicate when interpreting a unification goal. It also absorbs goal/clause matching and guard evaluation, since these are carried by the *clause/2* predicate.

A termination detecting meta-interpreter
The meta-interpreter described is not so interesting on its own right. However, it may be enhanced in several ways, to provide useful functionalities. One example is the following meta-interpreter, employing the short-circuit technique to detect the termination of the interpreted program. On the call *reduce(A,Done)*, *Done* is unified with *done* when the computation of *A* successfully terminates.

```
% reduce(Goal,Done) ← Goal is reducible and Done=true.

        reduce(A,Done) ←
            reduce'(A,done-Done).

        reduce'(true,L-R) ← L=R.
        reduce'(X=Y,L-R) ← (X,L)=(Y,R).
        reduce'((A,B),L-R) ← reduce'(A,L-M), reduce'(B,M-R).
        reduce'(goal(A),L-R) ← clause(A,B), reduce'(B,L-R).
```

One of the main weaknesses of FCP(|) is that, although it can reflect on termination, it cannot reflect on failure, without reifying unification. In other words, it is not possible in FCP(|) to enclose a computation within a meta-interpreter in the style shown above, which reports failure when the computation it interprets fails, without failing itself.

This problem is alleviated in more powerful languages such as FCP(:), as discussed in Section 14.

An alternative solution is to replace FCP(|)'s unification primitive with a three-argument predicate, which returns an indication whether unification succeeded or failed. This approach is taken by Fleng [133], and is discussed in Section 21.

Interrupt handling
Processes in FCP(|) are anonymous. Their number and rate of creation and termination renders any conventional operating system approach to process management infeasible. Therefore the implementation of standard operating system capabilities, such as the ability to suspend, resume, and abort processes requires novel solutions. The natural unit of control in concurrent logic programming is not a process, but a (reactive) computation.[12]

In the Logix system [170] several, possibly interacting, computations can proceed concurrently. We show below a meta-interpreter that can control an interpreted reactive computation by responding to control signals.

```
% reduce(Goal,Is) ←
  Is is a stream of suspend, resume and abort messages. Goal is reducible or Is contains abort.

        reduce(true,Is).
        reduce(X=Y,Is) ← X=Y.
        reduce((A,B),Is) ← reduce(A,Is), reduce(B,Is).
        reduce(goal(A),Is) ← clause(A,B,Is), reduce(B,Is).
        reduce(A,[I|Is]) ← serve_interrupt([I|Is],A).

        serve_interrupt([abort|Is],A).
        serve_interrupt([suspend|Is],A) ← serve_interrupt(Is,A).
        serve_interrupt([resume|Is],A) ← reduce(A,Is).
```

The plain meta-interpreter is enhanced with an interrupt stream *Is*. Whenever an interrupt is sensed, an interrupt-handling routine is called. The interrupt handler can serve the messages *suspend*, *resume*, and *abort*. To ensure that an interrupt will eventually be served, even if the interpreted computation is non-terminating, the *unknown(Is)* guard should be added to all but

[12] The notion of computation employed here is closely related to the one used in the semantic definitions, but is different from it in being reactive.

the last clause of *reduce*. To ensure that even a suspended process responds to an interrupt, an additional clause is added to the representation of the interpreted programs:

```
clause(A,B,[I|Is]) ← A=B.
```

Its purpose is to return the interpreted process intact when an interrupt is sensed. If an interrupt is sensed, the *clause* process terminates and returns in the body argument the goal atom it was called with. This ensures that suspended goal atoms of the interpreted computation are halted rather than being left suspended. Once the computation is resumed, the process is retried. This feature is used for another purpose by the following snapshot meta-interpreters.

Repeated live snapshots

The problem of obtaining a snapshot of the state of a distributed computation has been investigated is various models [14,15]. The meta-interpreter shown above can be enhanced to obtain repeated snapshots of the interpreted computation, by treating the short-circuit as a (possibly empty) stream of snapshot requests. To obtain a snapshot, a message *state*([ ]) is sent down the left end of the short-circuit. A process P that senses a message *state*(S) on its left-end of the switch sends the message *state*($[P|S]$) on the right end of the switch. This is achieved by augmenting the termination detection meta-interpreter shown above with the clause [149] :

```
reduce(A,[state(S)|L]-R) ← R=[state([A|S])|R'],
    reduce(A,L,R').
```

When the message *state*(S) arrives at the right end of the circuit, it contains a list of processes.

There are several delicate points to note. First, as specified, the message is guaranteed to arrive eventually only if the *interpreted computation* terminates or deadlocks. To improve upon this the guard *unknown*(L) can be added to the other clauses of the meta-interpreter. This ensures that if the number of processes created in the computation is bounded (i.e. the number of times a clause with more than one atom in the body is used is finite), then the message would eventually arrive, even if the computation is nonterminating. To obtain a snapshot in a computation with unbounded process creation, the frozen snapshot technique, discussed below, must be used.

Second, the distributed fashion in which the live snapshot was obtained implies that the list of processes obtained is not necessarily a possible state that actually occured in a computation [158]. For example, process A could have been added to the snapshot, then reduced, performed a unification that enabled some other reduction, which created a process B, which was then added to the snapshot. So the live snapshot may contain two processes which are causally related, and therefore could never exist simultaneously. Furthermore, processes in the snapshot could appear more instantiated than they were when added to it, due to other processes reducing before the snapshot was completed. Nevertheless, under certain circumstances[13], a live-snapshot is restartable, in the following sense. If G has a successful computation, and G' is a live snapshot of this computation, then G' also has a successful computation (but may also have failing and deadlocked ones). In spite of these limitations live snapshots are useful for various purposes, including the detection of stable properties of networks. This subject is further discussed in [158].

Combining the concepts: interrupt handling, termination detection, and the computation of live and frozen snapshots

We show a meta-interpreter which combines the various features discussed. It has both an interrupt stream and a short circuit, and it uses the clause form of the interrupt-handling meta-interpreter.

```
reduce(true,Is,L-R) ← L=R.
reduce(X=Y,Is,L-R) ← (X,L)=(Y,R).
reduce((A,B),Is,L-R) ← reduce(A,Is,L-M), reduce(B,Is,M-R).
reduce(goal(A),Is,L-R) ←
```

[13] Specifically, in the case of FCP(|), that neither *var* nor *unknown* are used in the interpreted program.

```
    clause(A,B,Is), reduce(B,Is,L-R).
reduce(A,[I|Is],L-R) ←
    serve_interrupt([I|Is],A,L-R).

serve_interrupt([halt|Is],A,L-R) ← L=R.
serve_interrupt([suspend|Is],A,L-R) ←
    L=[Done|L'], R=[Done|R'],
    serve_interrupt(Is,A,L'-R').
serve_interrupt([resume|Is],A,L-R) ←
    reduce(A,Is,L-R).
serve_interrupt([snapshot|Is],A,L-R) ←
    L=[state(S)|L'], R=[state([A|S])|R'],
    serve_interrupt(Is,A,L'-R').
```

The meta interpreter, called with the goal *reduce(G,Is,L-R)*, can be used to obtain a live snapshot *S*, even in the presence of unbounded *process creation*, by providing it with the following input:

```
Do in parallel:
Is=[snapshot,resume|Is'], L=[state([ ])|L'], R=[state(S)|R'].
```

which cause each process to suspend, add its state to the snapshot, and resume immediately.

A frozen snapshot is obtained by suspending the computation, and only then collecting the state of its processes. The following sequence of unifications can be used to get a frozen snapshot and then resume a computation.

```
Suspend the computation:
    Is=[suspend|Is'], L=[done|L'],
    wait till R=[done|R'],
Take a snapshot:
    Is'=[snapshot|Is''], L'=[state([ ])|L''],
    wait till R'=[state(S)|R'']
Resume:
    Is''=[resume|Is'''].
```

Specialization of meta-interpreters

We have shown that an enhanced meta-interpreter is a very convenient tool for specifying functions of computation control. However, a naive implementation of these functions via enhanced meta-interpreters could be quite costly. It is quite common that a program interpreted under an enhanced meta-interpreter runs an order of magnitude slower compared with its direct execution.

One approach to the problem employs the concept of partial evaluation [58,43], first explored in this context by Gallagher [60,61], and refined by others [111,150,149,178,101]. It is to specialise at compile-time the meta-interpreter for the execution of a given program.

For example, consider the following (inefficient) FCP(|) program for reversing a list:

```
rev([X|Xs],Ys) ← rev(Xs,Zs), append(Zs,[X],Ys).
rev([ ],Ys) ← Ys=[ ].

append([X|Xs],Ys,Zs) ← Zs=[X|Zs'], append(Xs,Ys,Zs').
append([ ],Ys,Zs) ← Ys=Zs.
```

The plain meta-interpreter, specialised to execute this program, is the program itself (although *append* can be specialized further, see [149]). In [149,150] a partial evaluator for Flat Concurrent Prolog, capable of partially evaluating meta-interpreters, was developed. As there is no partial evaluator for FCP(|), we show here examples of manual specializations of meta-interpreters. Using partial evaluation techniques similar to those of [149,150], the termination-detection meta-interpreter can be specialized to execute this list reversal program, resulting in the program [149]:

```
rev([X|Xs],Ys,L-R) ← rev(Xs,Zs,L-M), append(Zs,[X],Ys,M-R).
rev([ ],Ys,L-R) ← (Ys,L)=([ ],R).

append([X|Xs],Ys,Zs,L-R) ← (Zs,L)=([X|Zs'],M), append(Xs,Ys,Zs',M-R).
append([ ],Ys,Zs,L-R) ← (Ys,L)=(Zs,R).
```

And the interrupt-handling meta-interpreter can be specialised to execute this program, resulting in:

```
rev([X|Xs],Ys,Is) ← rev(Xs,Zs,Is), append(Zs,[X],Ys,Is).
rev([ ],Ys,Is) ← Ys=[ ].
rev(Xs,Ys,[I|Is]) ← serve_interrupt([I|Is],rev(Xs,Ys)).

append([X|Xs],Ys,Zs,Is) ← Zs=[X|Zs'], append(Xs,Ys,Zs',Is).
append([ ],Ys,Zs,Is) ← Ys=Zs.
append(Xs,Ys,Zs,[I|Is]) ← serve_interrupt([I|Is], append(Xs,Ys,Zs)).

serve_interrupt([abort|Is],A).
serve_interrupt([suspend|Is],A) ← serve_interrupt(Is,A).
serve_interrupt([resume|Is],rev(Xs,Ys)) ← rev(Xs,Ys,Is).
serve_interrupt([resume|Is],append(Xs,Ys,Zs)) ← append(Xs,Ys,Zs,Is).
```

Note how the state of the interrupted process is passed to the *serve_interrupt* routine, and that this routine has two clauses, one for resuming *rev* and one for resuming *append*.

Such specialisations eliminate the overhead of interpretation, while preserving the functionality of the enhanced meta-interpreter. The transformed programs are usually only 10% to 50% slower than the original programs, depending on the added functionality, compared to the order of magnitude slowdown of naive execution of the interpreter [84].

Techniques for proving the correctness of transformations of concurrent logic programs are not, as yet, well established. One question under debate is whether a transformation should preserve the meaning of a program, including all possible nondeterministic choices, an approach taken by [57,204], or whether a transformation could fix some choices at "compile time" thus change the meaning of a program; this approach views the source program as a specification, which may have several, nonequivalent but correct, implementations.

PART III. CONCURRENT LOGIC PROGRAMMING LANGUAGES

8. Language Comparison

In a trivial sense all reasonable programming languages are equivalent, since they are Turing-complete (i.e. can simulate a Turing machine, which is a universal computational model). However, if the differences between languages were not material, we would not have invented so many of them.

Concurrent logic languages are similar enough to allow a more precise comparison than is usual among programming languages. They all share the same abstract computational model, share the same principles, and employ very similar syntax. Therefore it it easier to focus on their differences. In comparing languages in this family, we consider mostly expressiveness, simplicity, readability, and efficiency.

In comparing languages for expressiveness, we use two methods: the first is to embed one language in another; the second is to show programming techniques available in one but not in another. We conclude that one language is more expressive, or "stronger" than another if the latter can be "naturally" embedded in the former, but not vice versa, and/or if all programming techniques of the latter are available in the former, but not vice versa.

We first define the notion of language embedding, which can be used to compare any two languages, and then discuss the finer notion of natural embedding, which is tailored for the comparison of logic programming languages. Related notions of language and their application to the comparison of concurrent logic lang ages were studied by Saraswat [155] and Levy [114].

Definition: Language embedding

Let L_1 and L_2 be two languages, c a function from L_1 programs to L_2 programs, and v a function from observables of L_2 to observables of L_1. We say that (c,v) is an *embedding of* L_1 *in* L_2 if $v(\llbracket c(P) \rrbracket_{L_2}) = \llbracket P \rrbracket_{L_1}$ for every L_1-program P. In such a case c is called the *compiler* of the embedding and v its *viewer*.

We say that L_1 can be embedded in L_2 if there are effective functions c and v such that (c,v) is an embedding of L_1 in L_2. ∎

In other words, a compiler c and a viewer v form an embedding of L_1 in L_2 if the observable behavior of every L_1 program P is the same as the observable behavior of the L_2 program obtained by compiling P using c and viewing its behavior through v.

The notion of embedding is rather weak. Because of the Turing-completeness of the languages under consideration, any language L_1 can be embedded in any other language L_2, by writing in L_2 an interpreter of L_1, and "compiling" an L_1 program P to the L_2 program consisting of the interpreter augmented with a representation of P.

The real issues with embeddings from L_1 to L_2 are what is the complexity of the compilation (e.g. how complex is the L_1 interpreter written in L_2), what is the runtime overhead of compiled programs, how much of the parallelism of L_1 is preserved in the compilation, etc. This is usually related to how much of the execution mechanism of L_1 needs to be *reified* in the translation, and how much of it can be *absorbed* in the execution mechanism of L_2.

The basic execution mechanism in logic programming is unification. Therefore we are interested in embeddings from L_1 to L_2 which absorb unification, i.e. use the unification mechanism of L_2 to implement the unification mechanism of L_1. In such an embedding, logical variables of L_2 represent logical variables of L_1 and hence nonground goals of L_2 can be used to represent similar goals of L_1. We call an embedding that maps logical variables in one language to logical variables in another *natural*. We formalise this notion by requiring that the viewer be the identity function on observables containing goals with predicates of the source program (although it may hide auxiliary predicates introduced in the target program, if any). This precludes embeddings in which the compiler encodes variables in the source language by constants of the target language and the viewer decodes the answer substitution given in terms of these constants[14].

Definition: An embedding (c,v) between two concurrent logic programming languages is *natural* if v is defined by:

$$v(\llbracket c(P) \rrbracket) = \{\langle G,\theta,x\rangle \in \llbracket c(P) \rrbracket \mid G \text{ is a predicate of } P\}. \quad ∎$$

The observables of a concurrent logic program $P, \llbracket P \rrbracket$, were defined in Section 4.2. In the following discussions of embeddings we assume that the viewer is defined as above and hence discuss only the compiler.

In the following we show natural embeddings among concurrent logic programming languages, and argue (although not prove) the lack of opposite natural embeddings. Our findings are summarised in Figure 5. An arrow in the figure indicates the existence of a natural embedding.

Most of the embeddings we show have additional pleasant properties. For example, being defined clause-wise, and preserving not only the observables but also the behavior in context (the so-called compositional semantics). We do not address these aspects further here.

[14] We note, however, that if the concurrent logic language has sufficiently powerful extra-logical constructs, which enable it to examine and compare logical variables, then it can construct its internal "variable-value" dictionaries. Using this dictionary encoding and decoding can be done internally by the target program. The languages discussed in this survey do not have this capability.

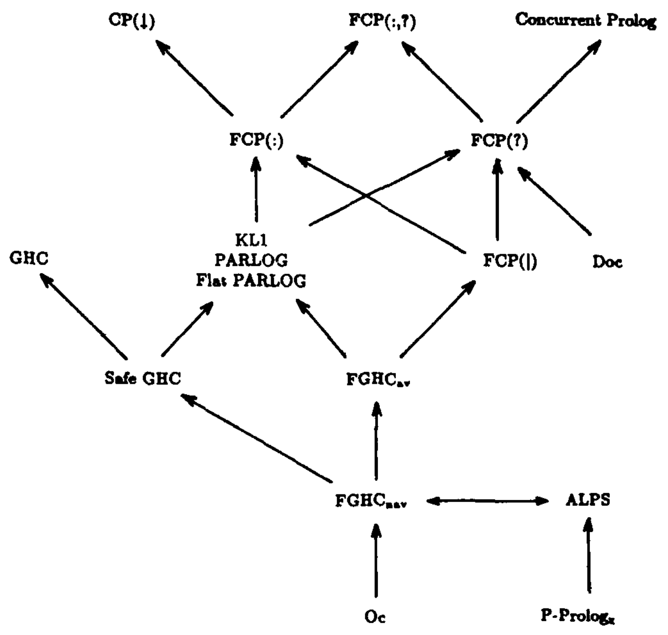


Figure 5: Natural embeddings among concurrent logic programming languages

A second dimension of comparison is simplicity of the syntax and semantics. A simpler language is preferred since it is easier both to grasp and to be used by humans, and is more amenable to automatic program analysis and transformation. Usually a weaker language is also simpler, but this is not always so, especially when the difference lies in the granularity of the atomic operations. Usually a language with coarser granularity, i.e. with larger atomic operations, is also stronger. Sometimes, in addition, its transition system is also simpler to define. For example, the languages FCP(|), FGHC$_{av}$, and FGHC$_{nav}$ discussed below, have, progressively, finer granularity and more complicated transition systems[15].

A third dimension of comparison is readability. All languages described in this survey use guarded Horn clauses, first employed in the Relational Language [20]. Most of them follow the syntactic conventions of GHC [197], that matching is used in the head, and unification is specified explicitly in the body. Exceptions are FCP(?), P-Prolog, ALPS, and Doc; the impact of their different syntactic conventions on readability is discussed when the languages are introduced.

A fourth dimension of comparison is ease of implementation. In general, the weaker the language the easier it is to implement. In particular, the finer the granularity of the language's atomic operations, the simpler the synchronization mechanisms required by its parallel implementation.

[15] The phenomenon is true for the interleaving semantics used in this paper, as well as for the approach to defining semantics for languages with non-atomic variables proposed by Maher [122] and extended by Saraswat [155]. It is conceivable that using another method for defining semantics may result in a different measure of simplicity.

We defer the comparative discussion of implementation to Section 20.

Each of the dimensions mentioned — expressiveness, simplicity, readability, and efficiency — is only one dimension in a multidimensional design space, which usually involves design tradeoffs. For example, a more expressive language may have a more complicated semantics, and be more difficult to implement. A weaker language may need extra-lingual facilities to compensate for its lack of expressiveness. Presently there is no consensus which language in this design space is optimal as a general purpose programming language for parallel and distributed computers, and several languages are being pursued actively as candidates for this role. Notable efforts which comprise of both language design, system design, and sequential and parallel implementations include KL1 (Flat GHC + control meta-call) [55] and PIMOS at ICOT [18], PARLOG and its flat variants [162,66,49], and a PARLOG system [48] at the Imperial College of Science and Technology, and Flat Concurrent Prolog [162] and its variants [99], and the Logix system [84,170] at the Weizmann Institute of Science.

For completeness, we provide a historical chart of concurrent logic languages in Figure 6. It is an extension of an earlier chart by Ringwood [145]. In the chart the vertical axis denotes the time in which the language design was published, and an arrow indicate some kind of intellectual influence.

9. Semantics of Concurrent Logic Programming Languages

In the following sections we investigate several concurrent logic languages. All the flat languages are defined similarly to FCP(|), and assume the same set of guard test predicate. Although small, this set turns out in practice to be sufficient for most practical purposes[16]. Their state of computation, as well as transitions, are identical to the ones of FCP(|) defined in Section 4.2. The differences between most of the flat languages are captured simply by varying the definition of the clause try functions. Although different, all try functions employed satisfy the following two properties:

a) *Suspension is not stable*:

If $try(Goal, Clause) = suspend$
then there is a substitution θ such that:
$try(Goal\theta, Clause) \neq suspend$.

b) *Failure is stable*:

If $try(Goal, Clause) = fail$
then for every substitution θ
$try(Goal\theta, Clause) = fail$.

Property *a* implies that a suspended clause try may succeed or fail in the future, if the goal atom is further instantiated (e.g. by reducing other atoms in the goal). Property *b* implies that a failed clause try need not be tried again.

We say that a language is *success stable* [157] if it satisfies the following property *c*:

c) *Success is stable*:

If $try(Goal, Clause) = \theta \circ \theta'$, for some θ and θ'
then $try(Goal\theta, Clause) \notin \{suspend, fail\}$.

Most languages discussed in this survey, including FCP(|), are success stable if the guard primitives *unknown* and *var* are excluded. Exceptions will be noted when introduced.

[16] See [170] for a description of the guard predicates and other primitives in a practical system.

Year	Language	Section Reference
73	Prolog	2.5 [171]
78	CSP	[86,87]
	IC-Prolog	[36]
81	Relational Language	[20]
83	Concurrent Prolog	18.3 [160]
	PARLOG 83	[21]
85	Flat Concurrent Prolog (FCP(?))	15 [128]
	GHC	18.1 [197,198,199]
	CP(↓,|)	18.2 [150]
86	Flat GHC	10
	PARLOG 86	18.1 [66]
	Oc	17 [81]
	P-Prolog	16 [209,210]
	CP(%)	[143]
87	Flat PARLOG	11 [49,66,106]
	ALPS	13 [124]
	Doc	17 [82]
88	FCP(|)	4 [this paper]
	FCP(:)	14 [100,156]
	FCP(:,?)	16 [100]

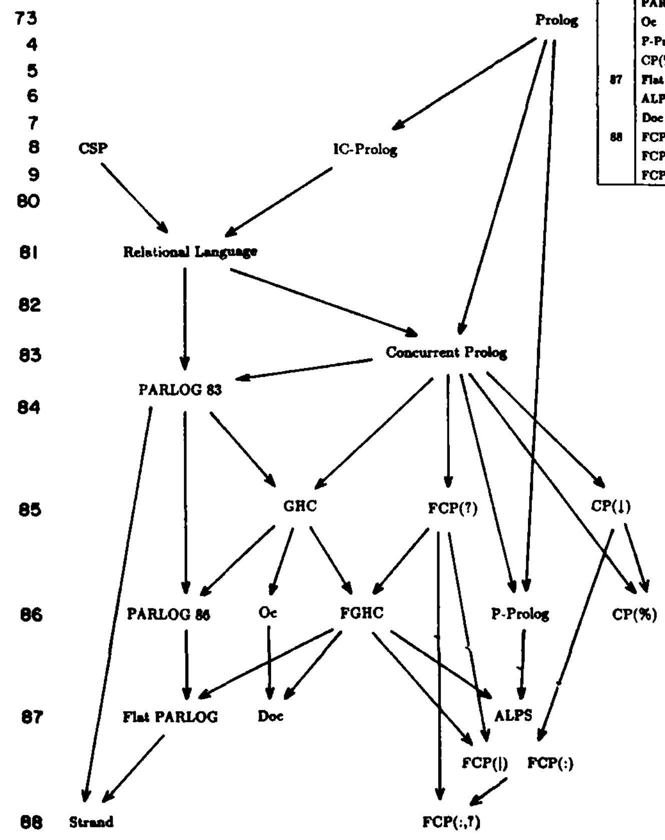


Figure 6: A historical chart of concurrent logic languages

The non-flat languages are described only informally. Transition system for non-flat languages were defined by Saraswat [154,157] and Levy [114].

The notion of language embedding as described in the previous section presupposed a definition of the observables of the source and target language. As discussed in Section 3, since concurrent logic languages employ don't-care nondeterminism, their observables record the results of failing and deadlocked computations, in addition to the results of successful ones. The observables of a concurrent logic program, in any of the languages surveyed, record the initial state and an abstraction of the final state of every computation.

Compositional semantics for concurrent logic programs that are fully abstract with respect to these observables were investigated by [59,65]. Other investigations of the semantics of concurrent logic languages include [9,44,5930,110,131,204]. However, since the work on the semantics of concurrent logic languages is in a state of flux we do not survey it here.

10. Flat GHC: A Language With Non-Atomic Unification

Flat GHC is the flat subset of the language Guarded Horn Clauses [198,199] (see Section 18). Flat GHC, augmented with a control meta-call primitive discussed in Section 10.3 below, is the basis of Kernel Language 1 [55], the core language of the parallel computer system developed at ICOT as part of the Fifth Generation Project [195].

We consider two variants of Flat GHC. One called $FGHC_{av}$, for Flat GHC with atomic variables, and the other called $FGHC_{nav}$, for Flat GHC with non-atomic variables.

$FGHC_{av}$ is derived from the original definition of GHC [198], and it is quite similar to FCP(|). The difference is that in $FGHC_{av}$ a unification specified by the goal $T_1 = T_2$ need not be carried out atomically. Saying it differently, a program in $FGHC_{av}$ cannot specify that a compound unification is to be carried out as an atomic operation. We have found only one implication of this difference in terms of expressiveness: $FGHC_{av}$ requires slightly more elaborate code than FCP(|) to implement the short circuit technique.

$FGHC_{nav}$ has an even finer notion of atomic actions. Intuitively, in $FGHC_{nav}$ even the instantiation of a variable to a value need not be done atomically, and several occurrences of the same variable can be instantiated to different (conflicting) values simultaneously. If such a conflict occurs it is eventually detected and results in failure. However, in $FGHC_{nav}$ there are intermediate states of the computation in which the same variable may have different values, whereas in $FGHC_{av}$ (and in FCP(|)) this cannot happen.

This property of $FGHC_{nav}$ is a consequence of the principle of *anti-substitutability* [199, 197], also called *logical referential transparency*[17]. The principle states, informally, that an occurrence of a variable X can be replaced in any context by a new variable X', provided the equality $X = X'$ is "added" to the context. The principle is motivated by semantic elegance, and it justifies a wide range of program transformations [204]. Operationally, it allows the "decoupling" of different occurrences of the same variable, and instantiating them to different values. In such a case the inconsistency between the instantiations is detected eventually, and failure results.

The main difference in terms of expressiveness between $FGHC_{av}$ *and* $FGHC_{nav}$ *is that in* the latter the short-circuit technique cannot be used to detect the successful termination of a computation. The reason is that the closing of the short circuit, both in the original version described for FCP(|), and the variant described for $FGHC_{av}$ below, cannot guarantee that only consistent instantiations have been made. It is still possible that two occurrences of some variable in the computation were instantiated to inconsistent values, which would result in failure past the closing of the circuit.

It seems that without additional facilities, such as the control meta-call discussed below,

[17] By Graem Ringwood.

detection of successful termination of a computation cannot be specified in FGHC$_{nav}$. Saying it differently, FGHC$_{nav}$ cannot reflect on successful termination, unlike FCP(|) and FGHC$_{av}$.

The initial informal description of GHC [198] takes the atomic variables approach, as well as the treatment of GHC in [157]. Subsequent theoretical work on GHC embraced the anti-substitutability principle [198], and thus imply non-atomic variables. The practical work at ICOT, however, still adheres to atomic variables: the KL1 language designed and implemented at ICOT is essentially FGHC$_{av}$ augmented with a control meta-call [90].

Although the impelementation work on GHC and KL1 adopted the notion of "flatness" employed in this paper, Flat GHC was defined formally only recently [204] under the name "Theoretical Flat GHC". The notion of flatness used there is a bit different from ours.

In the following we relate FCP(|), FGHC$_{av}$ and FGHC$_{nav}$ with regard to the short circuit technique, and show simple embeddings of FGHC$_{av}$ in FCP(|), and of FGHC$_{nav}$ in FGHC$_{av}$. The syntax and the try function are the same for FGHC$_{av}$ and FGHC$_{nav}$. The difference is captured in an additional *Anti-substitute* transition for FGHC$_{nav}$ described below.

10.1 The language FGHC$_{av}$

Syntax

Definition: An FGHC$_{av}$ *program* is a finite sequence of guarded clauses that include the unit clause:

$$X = X.$$

and the clauses:

$$f(X_1,X_2,\ldots,X_n) = f(Y_1,Y_2,\ldots,Y_n) \leftarrow X_1{=}Y_1,\ X_2{=}Y_2,\ldots,X_n{=}Y_n$$

for every function symbol f/n, $n > 0$, occurring in some clause whose head predicate is different from '='[18]. ∎

Semantics

The fact that unification need not be done atomically is captured by the equality clauses, which allow a compound unification to be performed piecemeal.

The FGHC$_{av}$ try function is defined as follows. Let $C = (X_1{=}X_2 \leftarrow \ldots)$ be a renaming of a unification clause in P.

$$try_{FGHC}(T_1{=}T_2,C) = mgu((T_1,T_2),(X_1,X_2)).$$

The try function for the other clauses of P is defined as for FCP(|).

Notes:

1) The semantics of the unit clause $X = X$ in FGHC$_{av}$ is identical to that of FCP(|), since $mgu((T_1,T_2),(X,X)) = mgu(T_1,T_2)$, if X does not occur in T_1 and T_2.
2) The atomic operation in FGHC$_{av}$ is assigning a variable to a variable, or assigning a term whose arguments are distinct variables to a variable. The first action is allowed by the unit unification clause; the second by the other clauses. We do not prevent larger atomic actions; we simply do not require them, by permitting smaller ones.

Comparison of FGHC$_{av}$ with FCP(|)

~~The main implication of the lack of~~ atomic unification in FGHC in terms of expressiveness is that FGHC$_{av}$ cannot use the short-circuit technique as specified to detect the termination of a computation. In FCP(|) one can perform the unification of the underlying computation $X =$

[18] We assume here that an initial goal does not contain function symbols which do not occur in the program. Alternatively, an equality clause has to be added for every function symbol allowed in a goal, or a general recursive definition of unification, using the Prolog-like predicates *functor* and *arg* should be used.

Y and close the short circuit L–R atomically, within the same compound unification $(X,L) = (Y,R)$. In FGHC$_{av}$ one needs first to perform the unification $X = Y$, wait for it to complete using matching, and only then close the short-circuit[19]. This can be achieved by the procedure *unify_and_close_sc*$(X,Y,L\text{–}R)$, defined using the auxiliary procedure *match_and_close_sc* as follows:

```
unify_and_close_sc(X,Y,L-R) ←
     X=Y, match_and_close_sc(X,Y,L-R).

match_and_close_sc(X,X,L-R) ←
     L=R.
```

Note that to detect the termination of an underlying computation, *unify_and_close_sc* must be used instead of '=' throughout the underlying program.

The FCP(|) termination detecting meta-interpreter shown in Section 7.7 above is also an FGHC meta-interpreter. However, the unification performed in the body of the clause:

```
reduce(X=Y,L-R) ← (X,L)=(Y,R).
```

behaves differently in FCP(|) and FGHC$_{av}$. The modified version of the short circuit can be used in an FGHC$_{av}$ termination detecting meta-interpreter, by replacing the above clause with the clause:

```
reduce(X=Y,L-R) ← unify_and_close_sc(X,Y,L-R).
```

The difference between FCP(|) and FGHC$_{av}$ can be observed by composing a program that does the compound unification $f(X,Y) = f(a,b)$ with a program that matches either $X = a$ or $Y = b$, then unifies the other variable with c. Such a program is the same in FCP(|) and FGHC:

```
test(a,Y) ← Y=c.
test(X,b) ← X=c.
```

If this FCP(|) program were to execute using the goal:

```
test(X,Y), f(a,b)=f(X,Y)
```

the terminal state would never contain a substitution in which $X := c$ or $Y := c$. As an FGHC$_{av}$ program, some executions will have $X := c$ and some $Y := c$, since FGHC$_{av}$ cannot specify that the two unifications $X = a$ and $Y = b$ be carried out atomically.

An attempt to establish the difference in power between languages with and without atomic unification was made by Saraswat [155].

An embedding of FGHC$_{av}$ in FCP(|)

FGHC$_{av}$ can be naturally embedded in FCP(|) using the following compiler. The compiler translates an *FGHC$_{av}$ program P to FCP(|) clause-wise. In every clause, it replaces every body goal* $X = Y$ with the goal *unify*(X,Y), where *unify*/2 is a predicate not occurring in P. It adds to the resulting program the clause:

```
unify(X,Y) ← X=Y.
```

and for every function symbol f/n occurring in P, the clause:

```
unify(f(X1,X2,...,Xn),f(Y1,Y2,...,Yn)) ←
     unify(X1,Y2), unify(X2,Y2), ..., unify(Xn,Yn).
```

This completes the description of the compiler.

Like in the definition of the semantics of FGHC$_{av}$, the effect of the definition of *unify*/2 is that a compound unification can be carried out either atomically, using the first clause, or non-atomically, using the other clauses.

[19] This method is due to E.D. Tribble

10.2 The language $FGHC_{nav}$

Syntax

The syntax of $FGHC_{nav}$ is the same as that of $FGHC_{av}$.

Semantics

The difference between $FGHC_{av}$ and $FGHC_{nav}$, namely the anti-substitutability principles, can be modelled in our framework by extending the transition system of $FGHC_{av}$ with the following transition:

- Anti-substitute:

  $\langle G;\theta\rangle \xrightarrow{\text{Anti-substitute}} \langle G',X=Y;\theta\rangle$
  where Y is a variable that does not occur in G, and G' is obtained by replacing one occurrence of X by Y in G.

This transition directly models the principle of anti-substitutability. However, as stated, it allows almost any $FGHC_{nav}$ program to diverge, by alternating the introduction and elimination of the equality goals using Anti-substitute and Reduce_{FGHC}. To prevent this, additional complicated fairness conditions need to be incorporated.

Note that the Anti-substitute transition may cause a conditional answer substitution to contain inconsistent assignments. We have defined substitutions to be functions, *i.e.* have a single value for a variable. This has to be modified in order to specify observables for $FGHC_{nav}$.

The difficulty in modelling the semantics of $FGHC_{nav}$ can be attributed to the need to accommodate inconsistent constraints on the values of variables. The method proposed by Maher [124] and further developed by Saraswat [156,157], suggest another method for modelling this. The method is to separate the goal atoms into "pools", each containing its own binding environment, and add explicit transitions which communicate equality constraints between pools. Failure occurs as soon as one of the pools detects inconsistency.

Comparison of $FGHC_{av}$ with $FGHC_{nav}$.

$FGHC_{av}$ could use the short-circuit technique to detect successful termination of a computation, albeit with some additional effort. The technique is not applicable in $FGHC_{nav}$. It is possible that the unifications executed by the monitored computation are inconsistent, without this inconsistency detected prior to the closing of the short circuit. Thus, unlike in FCP(|) or $FGHC_{av}$, the closing of the short-circuit is not a reliable indication that the computation has not failed. Technically, the Anti-substitute transition incorporates in the underlying computation unifications which are not threaded via the short-circuit. Even if the short-circuit closes, the new unifications introduced by the Anti-substitute transitions can subsequently fail.

An embedding of $FGHC_{nav}$ in $FGHC_{av}$

An embedding of $FGHC_{nav}$ in $FGHC_{av}$ consists of a clause-wise compiler and the identity viewer. For each clause, the compiler iteratively performs anti-substitution to any variable that occurs more than once in clause body atoms other than equality, until no such variables are left. By doing so, the compiler "decouples" every variable that may be used for communication, and adds equalities to clause bodies. These equalities will eventually unify all decoupled occurrences of each variable.

10.3 The meta-call construct

The ability to reflect on the termination and failure of a computation is essential to a systems programming language, but $FGHC_{av}$ (and FCP(|)) cannot do the latter, and $FGHC_{nav}$ can do neither, without reifying unification. The problem can be solved in two different ways. One is to strengthen the basic mechanisms of the language. Atomic variables are sufficient to reflect on termination. To reflect on failure, atomic test unification is needed, as incorporated in the stronger variants of FCP(|) shown in Sections 14, 15, 16 below.

Another solution, which was taken by the developers of both GHC and PARLOG, is to add to the language a meta-level construct, which has "built-in" reflection and control capabilities. There are several variations on the construct, originally proposed by Clark and Gregory [22]. One variant, which is referred to in the following as the *control meta-call*, has the form *call(Goal,Signals,Events)*, where *Signals* is a stream of {*suspend, resume, abort*}, and *Events* is a stream of {*suspended, resumed, failed*(*Goal*), *halted, aborted*}, the last three being terminal events.

The intuitive semantics of the control meta-call is as follows. A computation of a goal G is started under the control meta-call using the goal *call*(G,*In*,*Out*). If some goal atom G' in the computation fails, the message *failed*(G') appears on the *Out* stream. If the computation terminates, the message *halted* appears on *Out*. To suspend the computation, the message *suspend* is sent to the *In* stream, and when suspension occurs the acknowledgement message *suspended* appears on *Out*. Similarly, to resume or abort the computation the message *resume* or *abort* is sent on *In*, and the corresponding acknowledgement message *resumed* or *aborted* appears on *Out*. Using the control meta-call, a process in the language can start a computation and monitor it.

We refer to the language FGHC_{av} augmented with the control meta-call as KL1 [55]. The actual meta-call implemented as part of PIMOS [18], the KL1 operating system, also includes resource management facilities: a computation is allocated some CPU time and some memory, and when either of these is consumed it announces resource overflow and suspends. It can be resumed by providing it with additional resources.

The control meta-call eliminates much of the freedom of non-atomic variables. For example, it can be used to detect the successful termination of unification, a capability not present in FGHC_{nav}. Hence its implementation restricts the kind of algorithms that can be used in a distributed implementation of the language; in particular, the algorithms must incorporate some form of distributed termination detection.

In comparison with the meta-interpreters of FCP(|) shown in Section 7.7, the meta-call construct reflects on failure, whereas an FCP(|) meta-interpreter cannot. On the other hand, an FCP(|) meta-interpreter can produce snapshots, whereas the standard meta-call constructs cannot (although Gregory *et al.* [67] have proposed an enhanced control meta-call that does). We will come back to the meta-call when we discuss FCP(:) in Section 14.

Yet a third approach is to construct a meta-interpreter that reifies unification, and extend it in various ways. A first step in this direction was taken by Tanaka [182].

11. Flat PARLOG: FGHC Extended With Sequential-Or and Sequential-And

The PARLOG language [21], described in Section 18, preceded GHC, but went through several evolutions that made it closer to GHC [24,66,145]. In the earlier definition [21], referred to as PARLOG83 by [145], the output mechanism was assignment, rather than unification. In the latter definition [66], refered to as PARLOG86 by [145], the output mechanism was (non-atomic) unification, as employed by GHC. PARLOG consists of two sublanguages: the single-solution subset, and the all-solutions subset. The latter is essentially Or-parallel Prolog. Here we concentrate on the flat subset of the former. The non-flat language is discussed in Section 18. Presently, the main difference between the computational models of the single-solution subset of PARLOG and GHC is the sequential-Or and sequential-And constructs of PARLOG. In addition, PARLOG offers a surface syntax which contains mode declarations. For example, using modes, the PARLOG *append* program could be specified as follows:

```
mode append(?,?,↑).

append([X|Xs],Ys,[X|Zs]) ← append(Xs,Ys,Zs).
append([ ],Ys,Ys).
```

This program is then translated to PARLOG standard form:

```
append(Xs,Ys,Zs) ←
    Xs ⇐ [X|Xs'] | Zs=[X|Zs'], append(Xs',Ys,Zs').
append(Xs,Ys,Zs) ←
    Xs ⇐ [ ] | Ys=Zs.
```

where ⇐ is PARLOG's input matching primitive. This program is operationally identical to the Flat GHC program:

```
append([X|Xs],Ys,Zs) ← Zs=[X|Zs'], append(Xs,Ys,Zs').
append([ ],Ys,Zs) ← Ys=Zs.
```

Several proposals were made for a "flat" subset of PARLOG [66,106,49]. The Flat PARLOG of Foster and Taylor [49] is essentially Flat GHC with mode declarations as surface syntax. Recently, a language called Strand [188] was derived by Foster and Taylor from their Flat PARLOG language by restricting the output mechanism to be assignment, rather than unification. Strand is essentially a flat version of PARLOG83, with sequential-And and sequential-Or eliminated. PARLOG83 and Strand are not success stable.

Our definition of Flat PARLOG is based on the $KP_{AND\ Tree}$ language of Gregory [66]. The language is Flat GHC augmented with sequential-Or and sequential-And. We investigate each of the two extensions to Flat GHC separately, denoting the resultant languages FP(;) and FP(&). For the sake of uniformity we use the Flat GHC syntax instead of the PARLOG standard form syntax. The translation from the Flat PARLOG syntax to the Flat GHC syntax is straightforward.

In the following we refer to the language combining FP(;), FP(&), and the control meta-call as Flat PARLOG. Although the subject is not discussed explicitly in the PARLOG papers, we assume that PARLOG has atomic variables, and hence consider Flat PARLOG to be an extension of $FGHC_{av}$ rather than of $FGHC_{nav}$.

11.1 The language FP(;)

The language FP(;) allows the specification of sequential-Or clauses $C_1;C_2;\ldots;C_n$, where each disjunct C_i is an ordinary guarded clause. The idea of a sequential-Or clause is that the guarded clause C_i can be selected only if the clause tries of the clauses $C_1,\ldots,C_{i-1}$ fail. The connective ';' is called *sequential-Or*. The similarity of sequential-Or clauses to if-then-else constructs in procedural languages and to conditionals in Lisp is apparent.

Syntax

Definition: Sequential-Or clause, program.

- A *sequential-Or clause* is a guarded clause or has the form C_1 ; C_2 where C_1 is a guarded clause and C_2 is a sequential-Or clause.
- An FP(;) *program* is a set of sequential-Or clauses, augmented with unification clauses as in FGHC. ∎

Semantics

The $try_{FP(;)}$ function is defined as follows. For a conditional clause $C_1;C_2$

$$try_{FP(;)}(A,C_1;C_2) = \begin{cases} try_{FGHC}(A,C_1) & \text{if } try_{FGHC}(A,C_1) \neq fail \\ try_{FP(;)}(A,C_2) & \text{if } try_{FGHC}(A,C_1) = fail. \end{cases}$$

For a guarded clause C

$$try_{FP(;)}(A,C) = try_{FGHC}(A,C).$$

An embedding of FP(;) in $FGHC_{av}$ with *otherwise*

The embedding consists of a clause-wise compiler. Its method of compiling sequential-Or into *otherwise* (introduced in Section 7.6) is similar to the one used by Codish and Shapiro [29] to translate a non-flat language into a flat one.

The general idea is to reify clause selection, by explicitly programming the commitment operation. Each sequential-Or clause $C_1;C_2;\ldots;C_m$ is translated into a different procedure consisting of m guarded clauses by adding *otherwise* to the clauses $C_2,\ldots,C_m$. This ensures that a disjunct can succeed only if all previous disjuncts fail. The head predicates of clauses resulting from each disjunctive clause are renamed to form the new procedure.

A call to the original procedure is translated into a conjunctive call, one goal for each of the new procedures. The single-round mutual exclusion protocol shown in Section 7.4is used to ensure that at most one of these goals would "commit", i.e. proceed to execute the body of the selected disjunct of that sequential-Or clause. The other goals terminate quietly without causing any effect.

More specifically, an FP(;) procedure of the predicate p/n with k sequential-Or clauses is translated into $2k$ FGHC$_{av}$ procedures as follows. The i^{th} conditional clause $C_1;C_2;\ldots;C_m$ is translated into the two FGHC$_{av}$ procedures *test_p$_i$/k+1* and *commit_p$_i$/k+1*. Each disjunct $C_j = (p(T_1,T_2,\ldots,T_k) \leftarrow G \mid B)$ is translated to the clauses:

```
test_p_i(T_1,T_2,...,T_k,Commit) ←
    otherwise, G |
    Commit=lock(Reply),
    commit_p_i(Reply,T_1,T_2,...,T_n).
commit_p_i(granted,T_1,T_2,...,T_n) ← B.
commit_p_i(refused,_,_,...,_).
```

And the call to p/n is translated to calls to the *test_p$_i$* procedures using the clause:

```
p(X_1,X_2,...,X_n) ←
    test_p1(X_1,X_2,...,X_n,Commit_1),
    test_p2(X_1,X_2,...,X_n,Commit_2),
    ⋮
    test_p_m(X_1,X_2,...,X_n,Commit_m),
    mutex(Commit_1,Commit_2,...,Commit_m).
```

where the i^{th} clause of *mutex/n* is:

```
mutex(Commit_1,...,lock(Reply),...,Commit_m) ←
    Reply=granted,
    Commit_1=lock(refused),
    Commit_2=lock(refused),
    ⋮
    ...(excluding Commit_i)...
    ⋮
    Commit_m=lock(refused).
```

As the translation shows, there is a close relationship between sequential-Or and *otherwise*, and it can be said that they were both designed to solve the same problem. Which construct to prefer is largely a matter of taste. Both destroy clause-wise modularity and are easily open to abuse, and therefore should be used sparingly. Sequential-Or is more appealing in being general and uniform. *Otherwise* is more restricted (it can be viewed as a special case of sequential-Or [154,156]), as perhaps appropriate for an exceptional construct, and the cases in which it is less convenient than sequential-Or for its purpose are rare.

11.2 FP(&)

The language FP(&) is FGHC$_{av}$ augmented with sequential-And. Adding sequential-And to a language that supports dynamic creation of processes complicates both the definition and implementation of the language. In defining the operational semantics, the state of the computation cannot be represented by a sequence of goals. A tree of alternating sequential-And and parallel-And nodes, whose leaves contain the goals, is required. The definition of a transition is also complicated by the constraint that a goal can be selected only if it can be reached from the root by selecting the left-most branch in every sequential-And node.

Because of this complication, FP(&) does not fit the semantic framework we described. Instead, we define the syntax of FP(&), and provide it with semantics by embedding it in FGHC$_{av}$, using the short-circuit technique.

The compiler of the embedding translates each FP(&) program P into an FP(&) interpreter written in FGHC$_{av}$, augmented with the standard clausal representation of P. Since FGHC$_{av}$ can be embedded directly in FP(&), using the identity compiler and viewer, this shows that the two languages and practically identical from an expressiveness point of view.

Syntax

Definition: FP(&) clause and program.

- An FP(&) *clause* is a formula of the form

  $A \leftarrow G_1,\ldots,G_m \mid B_1,\ldots,B_n. \qquad n,m \geq 0$

  where the A and G_i's are as before, and each B_i has the form:

  $A_1 \& \ldots \& A_k \qquad (k > 0)$

  where each A_i is an atom.
- An FP(&) *program* is a finite sequence of FP(&) clauses. ∎

Semantics

Let P be an FP(&) program. Translate each clause:

$A \leftarrow G \mid B.$

of P into the FGHC$_{av}$ clauses:

clause(A,B″) ← G | B′=B″.

where each unit goal G in B with predicate other than '=' is replaced by *goal*(G) in B' and

A ← reduce(A).

where *clause* and *reduce* are predicates not occuring in P. Call the resulting program P'.

An interpreter I of FP(&) in FGHC$_{av}$, which assumes this representation, is defined as follows:

```
reduce(A) ←
    reduce′(A,done–Done).
reduce′(true,L–R) ← L=R.
reduce′(X=Y,L–R) ← unify_and_close_sc(X,Y,L,R).
reduce′((A,B),L–R) ← reduce′(A,L–M), reduce′(B,M–R).
reduce′(A&B,L–R) ← reduce′(A,done–Done), wait(Done,B,L–R).
reduce′(goal(A),L–R) ← clause(A,B), reduce′(B,L–R).
wait(done,A,L–R) ← reduce′(A,L–R).
unify_and_close_sc(X,Y,L,R) ← See definition in Section 10.
```

The interpreter implements A & B by executing A and suspending the execution of B until A terminates. Recursively nested sequential and parallel And's, which may be created by recursive procedures, are handled correctly, by starting a new short circuit for every sequential component.

The compiler c is defined to map P to $P' \cup I$. The viewer v is the identify function on the predicates of P, and hides the predicates *clause* and *reduce*. The observables of an FP(&) are then defined to be $v([[c(P)]])$.

Not only the direct definition of sequential-And is quite complex, but also its direct implementation. First, without complex data-structures it may take an unbounded amount of time to find the next process to execute — the amount is determined by the depth of nesting of sequential and parallel And's. Second, in a parallel implementation of the language, executing correctly the conjunct A & B requires performing distributed termination detection on A.

The interpreter of FP(&) in FCP(|) solves the two problems by delegating them to the underlying implementation of FGHC$_{av}$: the process suspension and activation mechanism of FGHC$_{av}$ wakes up the *wait* process when its first argument is instantiated to *done*. The short circuit technique combined with the implementation of unification with atomic variables essentially realizes a well-known distributed termination detection algorithm based on distributed counters [158] (see discussion in Section 7.5).

12. P-Prolog$_x$ — Synchronizing Deterministic Logic Programs

In the languages presented so far synchronization was achieved with matching, specified by clause heads: a clause try suspends if its matching with the clause head, or checking the guard, suspend.

An alternative approach to synchronization in concurrent logic programming was proposed by Yang and Aiso [209,210], and incorporated in the language P-Prolog. Although P-Prolog incorporates also an all-solutions Or-parallel component, we do not discuss it here. We focus on its other component, which employs a novel synchronization mechanism called *exclusive guarded Horn clauses*. We refer to this language subset as P-Prolog$_x$.

P-Prolog$_x$ does not use matching for synchronization. It uses goal/clause unification, rather than matching, and employs the following synchronization principle instead: the reduction of a goal with a clause is enabled when it can be determined that the reduction with *all* alternative clauses is failed. In other words, a process is suspended as long as it has more than one clause to reduce with. It reduces if it has exactly one clause to reduce with; it fails when it has none. A process never makes an Or-nondeterministic choice.

The appeal of this synchronization principle is in the following lemma, a variant of which is due to Maher [124]. The lemma implies that the And-nondeterminism of P-Prolog$_x$ does not affect the result of computations.

Lemma: Equivalence of P-Prolog$_x$ computations.
If a P-Prolog program P has a successful computation from a goal G then every computation of P from G is successful and the answer substitutions of all such computations are the same (up to renaming). ∎

Syntax
The syntax of P-Prolog$_x$ is the same as that of FCP(|).

Semantics
We define the P-Prolog$_x$ try function, try_{PP}, using the auxiliary function try'_{PP}. Note that try'_{PP} is essentially try_{LP} augmented with guard evaluation. The program P is an additional parameter of the functions.

$$try'_{PP}(A',(A \leftarrow G|B),P) = \begin{cases} \theta & \text{if } mgu(A,A') = \theta \wedge \text{checking } G\theta \text{ succeeds} \\ fail & \text{if } mgu(A,A') = \theta \wedge \text{checking } G\theta \text{ fails} \\ & \quad \vee\ mgu(A,A') = fail \\ suspend & \text{otherwise} \end{cases}$$

$$try_{PP}(A,C,P) = \begin{cases} \theta & \text{if } try'_{PP}(A,C,P) = \theta \wedge try'_{PP}(A,C',,P) = fail \\ & \text{for every } C' \in P,\ C' \neq C \\ fail & \text{if } try'_{PP}(A,C,P) = fail \\ suspend & \text{otherwise} \end{cases}$$

Discussion

The advantage of P-Prolog$_x$ is that the order of execution of processes is immaterial, since if a goal has a successful computation, then all of its computations are successful and produce the same answer substitution.

The determinism of P-Prolog$_x$ limits it to algorithmic applications, since it cannot implement system programs such as a stream merger and an interrupt handler[20]. Most algorithmic concurrent logic programs can be written in P-Prolog$_x$ quite easily, without the need to distinguish between matching and unification. This implies that some P-Prolog$_x$ programs can be used in more than one 'mode'. Consider, for example, the P-Prolog$_x$ append program:

```
append([X|Xs],Ys,[X|Zs]) ← append(Xs,Ys,Zs).
append([ ],Ys,Ys).
```

This program can be used to append two lists, as usual. However, it can also be used to compute the difference between a list and its prefix, using, e.g., the call:

```
append([1,2,3],Ys,[1,2,3,4,5,6]).
```

This is possible since at most one clause head unifies with the initial goal, as well as with subsequent goals, and hence goal reduction can proceed.

The practical advantage of this 'multiple-mode' ability is questionable. In practice, few logic programs are used in more than one mode. When they do, the two common modes are output generation and testing, which can be employed by all other concurrent logic languages mentioned, rather then inverting the roles of input and output within a single clause, which is unique to P-Prolog$_x$ and its superset ALPS (and is available in a more restricted sense also in FCP(?) and FCP(:,?) introduced below).

Furthermore, P-Prolog$_x$ uses unification in the head. As mentioned in the discussion of FCP(?) in Section 15 below, this generality seems to impede program readability and maintainability, since often the intended mode of use is known and fixed, but is not communicated by the code.

Embedding P-Prolog$_x$ in FCP(|)

The implementation of P-Prolog$_x$ is not trivial. A naive implementation would be to try all clauses whenever a process reduction is attempted; return to the successful clause if only one exists, or suspend on all variables instantiated during clause tries if there were more than one successful clause try. The overhead of this scheme seems unacceptable. An efficient implementation of P-Prolog$_x$ seems to require a complete analysis of all possible call patterns, which is also quite complex.

To establish the relation between P-Prolog$_x$ and other languages in the family, we show here an embedding of P-Prolog$_x$ in FCP(|). The idea of the embedding is as follows. For each goal atom in the source program we create a controlling process, and for each source clause potentially unifiable with this atom we create a reduction process simulating the attempt to reduce the goal atom with the clause. The reduction process operates as follows. If it detects that it cannot perform the simulated goal/clause reduction, it informs the controller. If it receives a permission from the controller to reduce, it simulates the reduction.

The controlling process counts the number of clause try failures, and when all but one clause have failed, it permits the remaining one to try and reduce. This behavior is achieved by the following translation[21].

Each P-Prolog$_x$ clause $A \leftarrow G|B$ is translated into an FCP(|) procedure with three clauses. The purpose of the first clause is to fail as soon as it is determined that the goal atom does not unify with the head of the source clause or the guard fails. It can never succeed. The second clause

[20] Although an *ad hoc* extension to allow this was proposed [210]. Another extension, ALPS, is discussed in the next section.

[21] This translation was developed in collaboration with M. Maher, and benefited from comments by V.A. Saraswat.

informs the controller if the first clause has failed. The third clause reduces if permission is given from the controller.

Specifically, let $C_1, C_2, \ldots, C_k$ be the clauses of the P-Prolog$_x$ procedure p/n. It is translated into $k+1$ FCP(|) procedures, p/n, $p_1/n+2$, $p_2/n+2$, ..., $p_k/n+2$, which use two auxiliary procedures, as follows. The i^{th} clause $p(T_1, T_2, \ldots, T_n) \leftarrow G|B$ of the P-Prolog$_x$ procedure p/n is translated into the FCP(|) procedure:

```
    p_i(T_1,T_2,...,T_n,_,foo) ← G | true.
    p_i(_,_,...,_,Failed,_) ← otherwise | Failed=failed.
    p_i(X_1,X_2,...,X_n,go,_) ← G | (X_1,X_2,...,X_n)=(T_1,T_2,...,T_n), B.
```

The FCP(|) procedure p/n is defined as follows:

```
    p(X_1,X_2,...,X_n) ←
        p_1(X_1,X_2,...,X_n,S_1,_),
        p_2(X_1,X_2,...,X_n,S_2,_),
        :
        p_k(X_1,X_2,...,X_n,S_k,_),
        xor_k(S_1,S_1,...,S_k),
```

where xor_k is defined as follows:

```
    xor_k(Go,failed,failed,...,failed) ← Go=go.
    xor_k(failed,Go,failed,...,failed) ← Go=go.
    :
    xor_k(failed,failed,...,failed,Go) ← Go=go.
```

with *Go* on the diagonal, and *failed* anywhere else.

The translated program operates as follows. The procedure p/n spawns k parallel clause processes p_i, one for each of the original p/n clauses, plus a xor_k process. If the i^{th} clause process fails it unifies the S_i variable with *failed*. The xor_k process counts $k-1$ failures, and unifies *go* with remaining variable, which enables the remaining clause process to reduce if it has not failed yet Note that the FCP(|) program fails whenever the source P-Prolog$_x$ program fails.

The translation assumes that the unification implied by P-Prolog$_x$'s Reduce transition is atomic. If it is non-atomic, then the exact same embedding can be used with FGHC$_{av}$ or FGHC$_{nav}$ as the target language, depending on the kind of nonatomicity allowed.

13. ALPS — An Integration of P-Prolog$_x$ and FGHC

ALPS was proposed by Maher [124] as an algorithmic concurrent logic programming language. ALPS goal reduction rule states that a goal can be reduced with a clause if either this is the only candidate clause left (the P-Prolog$_x$ rule), or the reduction does not instantiate variables of the goal (the FGHC and FCP(|) rule).

In particular, the FGHC unification primitive is definable in ALPS using the single unit clause:

$$X = X.$$

The reduction of the goal $T_1 = T_2$ with this clause is enabled if T_1 and T_2 are unifiable, using the P-Prolog rule, since this is the only candidate clause. Unlike FCP(|), and like FGHC, the unification specified by such a goal need not be carried out atomically. In particular, the transition system of ALPS defined by Maher realizes non-atomic variables, as in FGHC$_{nav}$.

ALPS was defined in the general setting of constraint logic programming [92]; we address this aspect of the language in Section 21.

Embedding $FGHC_{nav}$ and P-$Prolog_x$ in ALPS

$FGHC_{nav}$ can be embedded in ALPS using a compiler that duplicates each clause, and the identity viewer. Clause duplication prevents the resulting ALPS program from "eagerly" reducing using the determinacy rule, since no goal is ever determinate[22]. P-$Prolog_x$ can be embedded using the embedding into FCP(|), shown in the previous section, assuming unification need not be carried out atomically. ALPS can be embedded in FGHC much the same way that P-$Prolog_x$ was embedded in FCP(|).

Discussion

The transition rules of ALPS are more 'eager' than those of FGHC. This means that some programs which deadlock as FGHC programs may proceed as ALPS programs. The practical implications of this difference are yet be determined. The benefits in terms of added expressiveness are unclear, and the comment on P-$Prolog_x$ apply here as well. In addition, the difficulties in efficient implementation of the ALPS language, compared with FGHC, seem substantial.

14. FCP(:) — FCP(|) Extended With Atomic Test Unification

In FCP(|), FGHC and Flat PARLOG, a program can perform only matching prior to clause selection. In the next set of languages shown, FCP(:), FCP(?), and FCP(:,?)[23], a program can perform unification as part of the test for clause selection, prior to commitment. If the unification fails, it should leave no trace of its attempted execution; in other words, the unification attempt should be atomic. We call unification which is tried before commit *atomic test unification*. In FCP(|), atomic unification is a special predicate. In FCP(:) and FCP(:,?) it is definable, and in this sense these languages are natural generalizations of FCP(|).

The first flat language to combine input matching and atomic test unification is Saraswat's FCP(↓,|) [150,154]. This idea was generalized by Saraswat in the Ask-and-Tell framework [156], which gave rise to the languages cc(↓,|) [156,157] and the similar language FCP(:) [100] described below.

14.1 The language FCP(:)

Syntax

Definition: FCP(:) clause and program.

- An FCP(:) *clause* has the form:

  $A \leftarrow Ask : Tell \mid Body.$

  where *Ask* and *Tell* are possibly empty conjunctions of atoms, *Ask* atoms have guard test predicates, and *Tell* contains only equality atoms. If *Tell* is empty, the colon is omitted.
- An FCP(:) *program* is a sequence of FCP(:) clauses. ∎

Semantics

The effect of a clause try of a goal A with an FCP(:) clause with an empty tell part is the same as in FCP(|). If the tell part is not empty, the effect is as follows. First, the goal/head input matching and the guard checking are performed. If they fail or suspend, the clause try fails or suspends, respectively. If they succeed, then the unification specified by the tell is performed, which can either succeed or fail, but not suspend. If it succeeds, the result of the clause try is

[22] The clause duplication method is due to M. Maher.

[23] More precise but also more cumbersome names for these languages are, respectively, FCP(:,|), FCP(?,|) and FCP(:,?,|).

the substitution combining the ask substitution and the tell substitution. If it fails, the clause try fails.

Definition: Try function for FCP(:).

- Let $Tell = (X_1{=}Y_1,\ldots,X_n{=}Y_n)$ be a conjunction of equality atoms. We define $mgu(Tell) = mgu((X_1,\ldots,X_n),(Y_1,\ldots Y_n))$, and the try function to be:

$$try_{FCP(:)}(A,(A' \leftarrow Ask{:}Tell|B)) = \begin{cases} \theta \circ \theta' & \text{if } match(A,A') = \theta \wedge \text{checking } Ask\theta \text{ succeeds} \\ & \quad \wedge\, mgu(Tell\theta) = \theta' \\ fail & \text{if } mgu(A,A') = fail \vee \\ & mgu(A,A') = \theta \wedge \text{checking } Ask\theta \text{ fails} \vee \\ & match(A,A') = \theta \wedge \text{checking } Ask\theta \text{ succeeds} \\ & \quad \wedge\, mgu(Tell\theta) = fail \\ suspend & \text{otherwise} \end{cases}$$

Embedding of FCP(|) in FCP(:)

The embedding of FCP(|) in FCP(:) is trivial. All the compiler does is to replace the unification clause $X = X$ by the clause

```
X=Y ← true : X=Y | true.
```

This clause is necessary since '=' is a primitive in FCP(|) but not in FCP(:).

14.2 Programming in FCP(:)

Atomic test unification enables numerous programming techniques not available in any of the weaker languages introduced so far. These include multiple writers on shared variables, which can be used to realize sophisticated synchronization protocols and blackboard-like shared data structures; the ability to reflect on failure of unification, which enables the construction of failsafe meta-interpreters that can be used to realize the control meta-call; the ability to record the logical time in which a unification occurs, which is essential for computation replay and hence essential to concurrent algorithmic debugging; and the ability to simulate Prolog's test unification, and hence the ability to naturally embed Or-parallel Prolog and similar languages. We discuss these techniques below.

Mutual exclusion and multiple-writer streams

Using atomic test unification, single-round mutual exclusion can be achieved with less machinery than needed in FCP(|). Let $p_1,\ldots,p_n$ be the processes wishing to participate in a single-round mutual exclusion protocol, with unique identifiers $I_1,\ldots,I_n$. Add to each process an argument, and initialize all processes with this argument being the variable ME. Each process p_k competing for a lock attempts nondeterministically unify its identifier I_k with ME, or to check that ME is already instantiated to some $I \neq I_k$.

A schematic description of each process is as follows. The k^{th} process call is $p(ME,I_k,\ldots)$.

```
p(ME,I,...) ← true : ME=I | ... lock granted ...
p(ME,I,...) ← ME ≠ I | ... lock denied ...
```

This technique is not a substitute to the multiple-round mutual exclusion protocol shown in Section 7. However, in the special case that in each round the number of competing processes decreases by one, it can be generalized, as follows.

Assume a set of processes $p_1,\ldots,p_k$, where each p_i may wish to deposit a message m_i on a shared stream Ms. Furthermore, assume that the messages are pairwise not unifiable. One solution is to create a merge network for all these processes. However, if the number of processes actually wishing to deposit their message on the stream is much smaller than k (as is the case with exceptional message streams), this solution is very wasteful. A more efficient solution in this case is to extend the single-round mutual exclusion protocol above to streams, as follows. When wishing to deposit a message on *Ms*, the process nondeterministically attempts to do so, or to check that another message is already there. In the second case it calls itself recursively with the tail of the stream. Assume each process p_i is called with *Ms* as its first argument and m_i as its second, the code of a process is as follows:

```
p(Ms,M,...) ← true : Ms=[M|_] | ... message sent; do other things...
p(Ms,M,...) ← Ms=[_|Ms'] | p(Ms',M,...).
```

Using this protocol, if the number of messages to be placed on *Ms* is finite, every process wishing to place a message on *Ms* will eventually do so (assuming And-fairness).

The dining philosophers

The seminal problem of mutual exclusion is that of the dining philosophers [37]. In this problem n philosophers are sitting at a round table, with one fork between each two philosophers. To eat, a philosopher requires two forks. Each philosopher goes through a cycle of eating and thinking. The problem is to provide the philosophers with an algorithm that guarantees that they will not deadlock, and that no philosopher will starve.

Using atomic test unification on multiple-writer streams it is easy to specify a deadlock-free behavior for philosopher:

```
phil(Id,[eating(LeftId,done)|Left],Right) ←       % Left is eating, wait till
    phil(Id,Left,Right).                          % he is done.
phil(Id,Left,[eating(RightId,done)|Right]) ←      % Right is eating wait till
    phil(Id,Left,Right).                          % he is done.
phil(Id,Left,Right) ←                             % Atomically grab both forks
    true : Left=[eating(Id,Done)|Left'],
    Right=[eating(Id,Done)|Right'] |
    ... eat, when done unify Done=done,
    then think, then become:
        phil(Id,Left',Right').
```

The program is independent of the number of philosophers dining. A dinner of n philosophers can be specified by the goal:

```
phil(1,Fork1,Fork2), phil(2,Fork2,Fork3),..., phil(n,Forkn,Fork1).
```

whose execution results in each of the *Fork* variables being incrementally instantiated to a stream of terms *eating(Id,done)*, with the *Id*'s on each *Fork* reflecting the order in which its two adjacent philosophers use it. For example, a partial run of this program on a dinner of 5 philosophers provided the substitution:

```
Fork1 = [eating(1, done), eating(5, done), eating(1, done), eating(5, _) | _]
Fork2 = [eating(1, done), eating(2, done), eating(1, done), eating(2, _) | _]
Fork3 = [eating(3, done), eating(2, done), eating(3, done), eating(2, _) | _]
Fork4 = [eating(3, done), eating(4, done), eating(4, done), eating(3, done),
        eating(4, done) | _]
Fork5 = [eating(4, done), eating(4, done), eating(5, done), eating(4, done),
        eating(5, _) | _]
```

The run was suspended midstream in a state in which *Fork4* is free and the the 2nd and 5th philosophers are eating. Up to that point each of the philosophers ate twice, except 4 which ate three times.

This program is much simpler then the Parlog86 program for the dining philosophers in [145]. The key to its simplicity is indeed the ability of FCP(:) to specify atomic test unification: a philosopher atomically tries to grab both forks, excluding other philosophers from grabbing them. The mutual exclusion is obtained by unifying the head of the *Fork* stream with a term containing the unique *Id* of the philosopher.

The deadlock-freedom of the program is guaranteed by the language semantics. The program can be further enhanced to achieve starvation freedom as well.

The duplex stream protocol

Processes placing messages on a shared stream need not be competing; they can also cooperate, and use the shared stream for both communication and tight synchronization.

For example, consider a stream producer and a stream consumer, wishing to participate in the following interaction. When the consumer reads the stream, it wants to read all the messages produced so far by the producer. The producer produces messages asynchronously, but wishes to know whenever all messages it has produced so far have been read. This can be achieved using the following duplex stream protocol [152]. The producer places a message *M* on the stream wrapped as *write(M)*. The consumer, when reaching the end of the stream, places on it a *read* message. From the consumer's point of view, successfully placing a *read* on the stream indicates that it has read all messages produced so far. From the producer's point of view, failing to place a *write(M)* message, due to the existence of a *read* message, is an indication that all previous messages have been read. This is realized by the following code, where *produce(M,Ms,Ms',Status)* places the message *M* on *Ms*, returning the remaining stream *Ms'*, and *Status=new* if all messages previous to *M* have been already read, *Status=old* otherwise. *consume(Ms,Ms',Rs)* returns in *Rs* the messages ready in *Ms*, and in *Ms'* the remaining stream.

```
produce(M,Ms,Ms',Status) ← true : Ms = [write(M)|Ms'] | Status=old.
produce(M,[read|Ms],Ms',Status) ← Ms=[write(M)|Ms'], Status=new.

consume([M|Ms],Ms',Rs) ← consume'([M|Ms],Ms',Rs).

consume'(Ms,Ms',Rs) ← true : Ms=[read|Ms'] | Rs=[ ].
consume'([write(M)|Ms],Ms',Rs) ← Rs=[M|Rs'], consume'(Ms,Ms',Rs').
```

consume is two-staged so that it would not place a *read* message on an initially empty stream.

If the producer waits every so often for the consumer to catch up, then *consume* always terminates.

The duplex protocol gives rise to a much more efficient and more flexible bounded-buffer protocol than the FCP(|) protocol shown in Section 7.3. It is more efficient, since there is no acknowledgement for every message, only one per 'batch'. It is more flexible, since the producer can change its mind on how many messages to send without an acknowledgement, without consulting or affecting the consumer, and with no need to change 'buffer-size'.

CSP with both input and output guards

To demonstrate the power of atomic test unification, we show an FCP(:) simulation of CSP with output guards [87]. CSP with output guards is notoriously difficult to implement, and hence Occam [91], the practical realisation of CSP, adopts only input guards. It is interesting to note that a logic programming language with matching is sufficient to simulate CSP with input guards, but a language with both matching and atomic test unification seems to be required to simulate CSP with both input and output guards.

Consider two sets of processes $p_1,\ldots,p_n$, $c_1,\ldots,c_n$, wishing to participate in the following interaction. Some (possibly all) of the p_i's wish each to interact with exactly one of the c_i's, but they do not care which. Some (possibly all) of the c_i's wish each to interact with exactly with one of the p_i's, but they do not care which. We would like a protocol, which, if there are $i \leq n$ p's and $j \leq n$ c's willing to interact, then $min(i,j)$ pairs will do so. The protocol should be independent of i and j, and allow i and j to increase dynamically.

The protocol is as follows [148]. Each p willing to interact sends to all the c's the incomplete message *hello*(X). All messages sent by the same p have the same variable X, and the variables in messages sent by different *p*'s are distinct. Each *c* willing to interact does the following: it nondeterministically and atomically selects one of its incoming *hello*(X) messages and unifies X with its unique *Id*.

The program for the case of two *p*'s and two *c*'s is as follows:

```
p(X,ToC1,ToC2) ← ToC1=hello(X), ToC2=hello(X).

c(Id,hello(X1),_) ← true : Id=X1 | true.
c(Id,_,hello(X2)) ← true : Id=X2 | true.
```

The initial process network is:

$$p(X_1,M_{11},M_{12}),\ p(X_2,M_{21},M_{22}),\ c(a,M_{11},M_{21}),\ c(b,M_{12},M_{22}).$$

This process network terminates, and at the end of its execution exactly one of X_1 and X_2 will be instantiated to *a*, and the other *b*.

In this example the two *p*'s and two *c*'s were both willing to interact. However, the definition of *p* and *c* is applicable also in the more general case, in which less are willing to interact on each side, or that processes are added dynamically.

This demonstration of the power of atomic test unification also indicates that the distributed implementation of atomic test unification is far from being trivial. It is discussed in Section 20.

Otherwise and reflection on failure

In FCP(|) it is possible to prevent failure of user-defined processes, by appending to each procedure *p* the clause:

```
p(...) ← otherwise | ... report failure ...
```

However, there is no way to prevent the failure of the primitive unification process '='.

In FCP(:), on the other hand, since unification is definable, it is possible also to define failsafe unification using the clauses:

```
X = Y ← true : X=Y | true.
X = Y ← X ≠ Y | ... report failure of unification ...
```

More generally, it is possible to define a failsafe FCP(:) meta-interpreter, which, instead of failing when the interpreted program fails, simply reports the failure. To achieve this we modify the clause representation of the interpreted program, by appending to it the clause:

```
clause(A,B) ← otherwise | B = failed(A).
```

Using this representation, a termination detecting failsafe meta-interpreter for FCP(:) is defined as follows:

```
reduce(A,Result) ←
    reduce'(A,[ ]-Result).

reduce'(true,L-R) ← L = R.
reduce'((A,B),L-R) ← reduce(A,L-M), reduce(B,M-R).
reduce'(failed(A),L-R) ← R=[failed(A)|L].
reduce'(goal(A),L-R) ← clause(A,B), reduce'(B,L-R).
```

On a call *reduce(A,Result)*, *Result* is instantiated to the (possibly empty) stream of goals failed during the com, ıtation. The stream is closed when the computation terminates.

14.3 Embedding KL1 and Flat PARLOG in FCP(:)

The inability to reflect on failure without reifying unification made all the previous languages unable to implement the control meta-call efficiently. Therefore to make them practical this construct has to be introduced as a primitive into the language as discussed in Section 10.3.

We show how the control meta-call can be implemented in FCP(:), and thus provide an embedding of KL1 in FCP(:). Combined with the techniques used to embed FP(;) and FP(&) in FGHC$_{av}$, discussed in Section 11, the implementation of the control meta-call can be enhanced to provide an embedding of Flat PARLOG in FCP(:).

An implementation of the control meta-call in FCP(:)

The meta-call implementation consists of two components: a meta-interpreter, which can produce events and is sensitive to interrupts, and a computation monitor, which provides the user interface.

The meta-interpreter requires the same clause representation of the FCP(|) interruptible meta-interpreter shown in Section 7.7, augmented with the *otherwise* clause shown above and an interrupt-sensitive clause. Each FCP(:) clause (including the unification clause)

```
A ← Ask : Tell | B.
```

is translated into:

```
clause(A,X,Is) ← Ask : Tell | X=B'.
```

where B' is B transformed as in previous meta-interpreters, and two clauses are appended:

```
clause(A,B,Is) ← otherwise | B=failed(A).
clause(A,B,[I|Is]) ← A=B.
```

The first reports failure of a reduction attempt. The second aborts the attempt when sensing an interrupt. Note that the order of the last two clauses is important: if they were switched, then the meta-level process *clause(A,B,Is)* executing a failing object-level process *A* will suspend on the interrupt stream *Is* rather then reporting failure. This is another demonstration of both the subtlety and power of *otherwise*.

Using this representation, the following meta-interpreter achieves the desired behavior:

```
reduce(true,Is,Ss,L-R) ← L=R.
reduce((A,B),Is,Ss,L-R) ← reduce(A,Is,Ss,L-M), reduce(B,Is,Ss,M-R).
reduce(goal(A),Is,Ss,L-R) ←
    clause(A,B,Is), reduce(B,Is,Ss,L-R).
reduce(failed(A),Is,Ss,L-R) ←
    write(failed(A),Ss), L=R.
reduce(A,[I|Is],Ss,L-R) ←
    serve_interrupt([I|Is],A,Ss,L-R).

write(M,Ms) ← true : Ms=[M|_] | true.
write(M,[M|Ms]) ← write(M,Ms).
```

The differences between this and the snapshot meta-interpreter of FCP(|) shown in Section 7.7 above are the additional signals stream *Ss*, the clause added for handling process failure and the lack of a special clause for unification. The latter is not needed since the clause defining '=' is a normal FCP(:) clause which requires no special treatment. Failure is handled by placing an appropriate message on the signals stream, using the multiple-writer stream protocol. This is an example where creating a merger for each forked process would have had an unacceptable overhead. Assuming either low rate of process failure, or that the computation is suspended by the controller as soon as failure is detected, the multiple-writer protocol would exhibit a much better performance.

Note that if two unifiable processes fail, only one message is produced on the signals stream. This oddity can be solved either by allocating unique identifiers to the meta-interpreter processes (which is inelegant and quite expensive), or, in FCP(:,?), using the anonymous mutual exclusion protocol, discussed in Section 15.

An alternative solution which does not use a multiple-writer stream is to use the short-circuit in order to report failure, as in the failsafe FCP(:) meta-interpreter shown above. The disadvantage of *this approach* is that the list of failed goals will be seen only upon termination of the computation. A computation monitor, which suspends the computation as soon as a failed goal is sensed, cannot be programmed using this technique.

The definition of the computation monitor should be quite obvious now. Its top level is the same as the meta call, *call(Goal,Signals,Events)*. It invokes the meta-interpreter, keeping hold of the ends of its short circuit streams. It serves signals coming from the outside by forwarding them to the meta-interpreter, via the interrupt stream, and reports on events that happen during the computation by placing them on the *Events* stream.

The meta-interpreter given serves as a specification of the required functionality of the control meta-call. This functionality can be implemented by source to source transformation. The transformation presently employed in the Logix system [84] which achieves this functionality results in about 30% increase in runtime and 80% increase in code size. In [46], Foster reports an experimental study that quantifies the cost of direct support for metacontrol functions, and compares this with the cost of support by program transformation. The same paper describes extensions to an existing abstract machine [49] required to support these functions. This study indicates that direct support for the control meta-call need not be expensive, nor require complex implementation mechanisms.

Discussion: atomic test unification vs. non-atomic unification

It is a subject of ongoing debate whether it is preferable to have a stronger language which can embed meta-level functions such as the control meta-call, or to have a weaker language and provide specific meta-level functions as language extensions.

The issue seems to be a tradeoff between simplicity at the implementation level versus elegance and expressiveness at the language level. On one side of the debate are Flat GHC and Flat PARLOG, with non-atomic unification. On the other side are FCP(↓,|), FCP(:), FCP(?), and FCP(:,?), languages, with atomic test-unification.

The main arguments for a weaker language, with non-atomic unification and a built-in control meta-call, are:

- The base language is simpler to implement;
- The specialized meta-level construct can be added with less overhead than via a general-purpose language mechanism.
- The base language has simpler formal semantics, and is therefore better amenable to theoretical treatment such as verification and transformation.
- Atomicity of unification is not assumed by the theory of (pure) logic programming. Therefore, it is important to write programs without relying on atomic unification whenever possible, and a language with non-atomic unification encourages it. The resulting programs allow better declarative reading[24].

The main arguments for a stronger language, which has atomic test unification and can implement meta-level constructs via interpretation and transformation are:

- Providing semantics for any specific meta-level construct as part of the base language is both complicated and *ad hoc* (we know of no formal semantics for the control meta-call or similar constructs, other then the one implied by the semantics of FCP(:) combined with the definition of the control meta-call).
- The need for stronger meta-level constructs is continuously evolving (e.g. live and frozen snapshots, sophisticated debuggers, etc. which are not provided by the control meta-call). If these needs are met at the language definition level, rather than by interpretation and transformation, the language semantics as well as implementation have to be continuously modified.
- When atomic test unification is not employed, there is little or no runtime penalty compared

[24] These last two points were communicated by K. Ueda.

to implementations of the weaker languages.

- Should the efficiency of a direct implementation of a certain meta-level function be required, it can be provided without affecting the language semantics. Such a direct implementation can be viewed as a (possibly hand-coded) specialisation of a function that could be provided by the language itself.
- There are other applications in which the added strength of atomic test unification is employed, such as embedding Or-parallel Prolog [165], and debugging (see below).
- It is not obvious at present that the semantics of the weaker languages is indeed simpler.

Recently, Saraswat has proposed combining both atomic test unification and non-atomic unification in a single language [157]. Such a language inherits the complexities of both approaches, and it is not clear at present what performance gains it allows.

14.4 Computation replay and debugging

One type of bug which is most difficult to diagnose in concurrent programs are transient, or lurking, bugs. Once a bug occurs in a sequential deterministic language, it is possible to repeat the computation and analyze it with various tools. This is not always possible in a concurrent program, unless special measures are taken. Specifically, all communication and all nondeterministic (scheduler and program) choices made during a computation must be recorded, so that if an erroneous behavior is observed, the computation can be repeated.

We show an FCP(:) meta-interpreter that records scheduler and program (i.e. And- and Or-nondeterministic) choices made by the interpreted program. This information is sufficient in order to reconstruct closed (non-reactive) computations, in which all communication happens internally. The meta-interpreter computes a tree data-structure called a *trace*, which reflects the process reductions occurred in the computation. Each node in the trace contains the pair (*Time,Index*), with the time in which the process in that node reduced, and the identity of the clause used for reduction. Given an initial goal and a trace of its computation, the computation can be repeated by redoing the process reductions specified by the trace in the order specified by the *Time* field of each node, and for each reduction selecting the clause specified by the *Index* of its node.

To construct such a trace, we assume that the underlying machine maintains logical clocks [107], and that the language provides a new primitive, *time*(*Time*), which unifies *Time* with the present value of the local logical clock. The clause representation is modified, to provide additional information on the clause reduction: the logical time in which it took place, and the identity of the clause chosen[25].

The i^{th} clause $A \leftarrow Ask : Tell \mid B$ of the program is transformed into the clause:

```
clause(A,X,Index,Time) ← Ask : Tell, time(Time) | X = B', Index = i.
```

Using this representation, a meta-interpreter that constructs a trace is defined as follows:

```
reduce(true,true).
reduce((A,B),T) ← T=(T1,T2), reduce(A,T1), reduce(B,T2).
reduce(goal(A),T) ← T=trace(Index,Time,SubTrace),
    clause(A,B,Index,Time), reduce(B,SubTrace).
```

A computation reconstructor, which repeats a computation given an initial goal and a trace, can be written quite elegantly using incomplete data-structures. It first serializes the trace using the *Time* field, then executes the reductions in order, one by one. We do not show it here.

Given the ability to reconstruct a computation, algorithmic debugging techniques [159] can be applied to concurrent programs as well. See [176,118,120] for details.

[25] Inability to record the time in which a unification occurs is what prevents the weaker languages shown from replaying computations.

14.5 An embedding of Or-parallel Prolog in FCP(:)

The question of how to provide the capabilities of Prolog in a concurrent logic programming languages has received considerable attention since the beginning.

One approach was pursued by PARLOG [24,25,66], namely to provide two sublanguages with an interface: the single-solution sublanguage, which is the counterpart of other concurrent logic programming languages, and the all-solutions sublanguage, which is essentially an all-solutions Or-parallel Prolog. A stream-like interface allows single-solution programs to invoke and control all-solution programs.

Another approach was to embed Prolog in a concurrent logic language. The first success in this direction was Kahn's Or-parallel Prolog interpreter in Concurrent Prolog, discussed in Section 18. However, this interpreter relies in an essential way on the non-flat nature of Concurrent Prolog. Initial attempts by Ueda [200, 201] and Codish and Shapiro [29], were successful in producing efficient translations when the mode of unification of the source Prolog program could be determined at compiler time. A more general, but less efficient, solution is described in [165], in the form of an Or-parallel Prolog interpreter written in FCP(?), a language introduced in Section 15. Although originally written in FCP(?), the interpreter does not exploit properties of it not available in FCP(:), and can be easily converted to this language. This implementation is not as direct as the interpreter in Concurrent Prolog, but is still quite simple. Furthermore, if the mode of subprograms can be determined, the interpreter can be gracefully interfaced to programs implemented using the transformations proposed by Ueda. The execution algorithm employed by the interpreter was proposed independently for other purposes [3,4,28]; nevertheless, its practicality is still under debate.

This embedding employs atomic test unification to implement Prolog's unification. Hence, unlike the disjoint-sublanguages approach, or the mode-based compilation, which is applicable to any concurrent logic language, the embedding approach is not applicable to languages such as FCP(|), (Flat) GHC, and (Flat) PARLOG. Should the execution algorithm employed by the embedded-language approach prove efficient in practice, its advantage over the disjoint-sublanguages approach would become apparent, especially in the presence of specialized hardware for the execution of concurrent logic languages [5,74].

A variant of the algorithm can be implemented also in Flat PARLOG or KL1, using the control meta-call. However, such an implementation would be hopelessly inefficient, since it would require a new meta-call at every choice point, and cannot prune alternatives using test unification as done in direct implementations of Prolog or in the Prolog interpreter in FCP(?).

Another approach, pursued by Saraswat [150,153,157] and Yang and Aiso [209,210] was to incorporate in concurrent logic languages don't-know nondeterminism. As the resulting languages cannot specify reactive concurrent systems, it is not an extension or a substitute for concurrent logic languages. Assuming that an underlying reactive concurrent logic language is still desired, the problem of integrating if with a parallel don't-know nondeterministic logic languages is much the same as that of integrating Prolog: it can either be implemented separately, with some all-solutions interface, as in the two sublanguages approach, or it can be compiled into a concurrent logic language, as in the embedding approach. This is discussed further in Section 21.

15. FCP(?) — Dynamic Synchronization With Read-Only Variables

The language Concurrent Prolog [160] introduced a different approach to synchronization, using read-only variables and read-only unification. The approach is preserved in its flat subset Flat Concurrent Prolog [128], also known as FCP, and called throughout the paper FCP(?) (read "FCP read-only").

FCP(?) assumes two types of variables, writable (ordinary) variables, and read-only variables, and uses read-only unification, which is an extention of ordinary unification, to unify terms containing read-only variables. The *read-only operator*, $?$, is a mapping from writable to read-only variables. When applied to a writable variable X, the read-only operator yields a *corresponding read-only variable* $X?$. The read-only operator is the identity function on terms other than writable variables.

In the absence of read-only variables read-only unification is just like ordinary unification. However, a read-only variable $X?$ cannot be unified with a value. An attempt to unify $X?$ with a term other than a writable variable suspends. When the writable variable X is instantiated to some value T (by some concurrent unification) its corresponding read-only variable $X?$ receives the value $T?$. This may release a unification suspended in an attempt to unify $X?$ with some value.

Whereas synchronization with matching is specified clause-wise and statically, synchronization with read-only unification is specified term-wise and dynamically. Read-only unification can be used to achieve various forms of dynamic synchronization, not acheivable otherwise.

15.1 The language

Syntax
The syntax of FCP(?) is the same as FCP(|), except that a clause may contain read-only variables.

Semantics
The semantics of the language is similar to FCP(|), except that goals may contain read-only variables, and the goal and the clause head are unified using read-only unification instead of matching.

Definition: Admissible substitution, read-only extension, read-only mgu.

- A substitution θ is *admissible* if $X?\theta = X?$ for every variable X.
- The *read-only extension* of an admissible substitution θ is the unique idempotent substitution $\theta_?$ satisfying $X(\theta_?) = X\theta$ and $X?\theta_? = (X\theta)?$ for every writable variable X.
- The *read-only mgu*, $mgu_?$, of two terms T_1 and T_2 is defined by:

$$mgu_?(T_1,T_2) = \begin{cases} \theta_? & \text{if } mgu(T_1,T_2) = \theta,\ \theta \text{ admissible} \\ fail & \text{if } mgu(T_1,T_2) = fail \\ suspend & \text{otherwise} \end{cases} \qquad \blacksquare$$

For example, $\{X \mapsto a\}$ is admissible but $\{X? \mapsto a\}$ and $\{X \mapsto a, X? \mapsto a\}$ are not. The read-only extension of $\{X \mapsto a, Y \mapsto Z\}$ is $\{X \mapsto a, X? \mapsto a, Y \mapsto Z, Y? \mapsto Z?\}$. $mgu_?(f(X,Y), f(a,Z)) = \{(X \mapsto a, X? \mapsto a, Y \mapsto Z, Y? \mapsto Z?\}$, and both $mgu_?(X?,a)$, $mgu_?(f(X,X?), f(a,a))$, and $mgu_?(f(X,X?), f(a,b)) = suspend$[26].

The try function of FCP(?) is the same as that of FCP(|), except that it uses $mgu_?$ instead of *match*, and it returns *suspend* if the read-only unification of the goal and the head is inadmissible due to read-only goal variable, and *fail* if it fails or is inadmissible due to a read-only clause variable only (since the latter state is stable).

15.2 FCP(?) programming techniques

Standard programming techniques
All standard programming techniques shown for FCP(|) and FCP(:) are realizable also in FCP(?). However, for most of the simple synchronisation tasks, the generality and the dynamic nature of read-only unification turns out to be more of a burden than an asset. Since read-only unification is an extension of unification, using it for goal/clause unification is closer to the original model of

[26] This definition of read-only unification is different from the original one [160], in that it is order-independent and disallows "self-feeding", i.e. the success of $f(X,X?)=f(a,a)$. The revision was influenced by criticism of the earlier definition [152,197], and by the language CP(%) of Ramakrishnan and Silberschatz [143].

logic programming and Prolog. Nevertheless, in concurrent logic programming, matching is used more often than unification. The default in FCP(?) encourages programmers to use unification even when matching is needed, and instead restrict the use of the procedure by placing read-only variables in the caller. For example, consider the FCP(?) procedure *append*.

```
    append([X|Xs],Ys,[X|Zs]) ← append(Xs?,Ys,Zs).
    append([ ],Ys,Ys).
```

The procedure is almost identical to the logic program (and Prolog program) *append*. The only difference is the read-only annotation in the recursive call. Nevertheless, this program has awkward behavior. Although its head specifies unification, the intention is that the first argument be matched. The program ensures this for recursive calls, but not for the initial call. If the initial goal is *append(Xs,Ys,Zs)* rather than *append(Xs?,Ys,Zs)*, the first (or second) clause can be chosen erroneously. Placing this responsibility on the caller is a source of non-modularity and bugs. In addition, matching can be compiled more efficiently than unification [99]. Without global analysis, which infers that the caller always places a read-only variable in the appropriate position [30,186], an FCP(?) program would compile less efficiently than a corresponding program in a language with input unification.

A later definition of FCP [170] allowed both a matching predicate =?= and unification in the guard. Using a matching guard, the recursive clause of append could be specified as:

```
    append(Xs,Ys,[X|Zs]) ← Xs =?= [X|Xs'] | append(Xs',Ys,Zs).
```

However, since this syntax is more verbose than the default one, programmers would still use the previous style, resulting in programs which are both more error prone and less efficient. In addition, it turned out to be difficult to define cleanly the try function for guards which contained a free mix of matching and unification predicates [173].

It seems, therefore, that the approach taken by the other flat languages, namely to use matching as the default, is better. The language FCP(:,?), discussed in the next section, attempts to unify the expressiveness of FCP(?) with the more convenient and efficient programming style of the other languages.

Test-and-set

One use of read-only variables is to implement various forms of a test-and-set operation. A variable can be tested to be a variable and then set to a non-variable term T in two stages: First unify it with a new read-only variable $X?$ and if successful unify T with X:

```
    test_and_set(X?,T) ← X=T.
```

The definition of read-only unification implies that the clause try will succeed with the goal *test_and_set(X,T)* if and only if X is a variable at the time of the try. The technique directly generalizes to simultaneous test-and-set of several variables.

The ability to implement test-and-set implies that FCP(?) is not success stable. For example, *test_and_set(X,a)* succeeds with X instantiated to *a*, but *test_and_set(a,a)* fails.

We note that test-and-set can be also realized in FCP(:), augmented with the *var* guard primitive, but not in any of the weaker languages.

Anonymous mutual-exclusion, multiple-writer streams and distributed queues

The ability to test-and-set can be used to implement anonymous mutual exclusion, that is, mutual exclusion without unique identifiers. For example, a multiple-writer stream, which preserves message multiplicity even in the presence of unifiable messages (in contrast to the FCP(:) program shown in Section 14 above) can be defined as follows:

```
    write(M,[X?|Ms],Ms) ← M=X.
    write(M,[_|Ms],Ms') ← write(M,Ms,Ms').
```

The third argument Ms' can be used to place subsequent messages on the stream. It ensures that the next message is placed after the previous one, so a writer can ensure that its own messages are ordered. Even if a writer placing several messages on a stream does not care for their order, he could still use Ms' instead of Ms for subsequent messages, to increase efficiency.

Using this procedure, placing n messages by n writers on one stream requires $O(n^2)$ steps. By introducing a special abstract data type, called *mutual-reference* [168], the three argument *write* operation specified by the above program can be implemented by a destructive assignment so that the cost of sending n messages is $O(n)$. The implementation is also 'better' than the specification in another respect: assuming And-fairness it guarantees that every *write* operation will eventually complete, even in the presence of an unbounded number of writers, a property not guaranteed by the program above. Mutual-references are the standard technique for realizing efficient stream mergers. Whenever we use a multiplicity-preserving multiple-writer stream in a program we assume it is implemented efficiently and fairly using mutual-references.

Another application of anonymous mutual exclusion is a distributed queue [165]. In it, client processes are at the leaves and queue processes are at the internal nodes of a process tree. Each *enqueue*(X,ME) or *dequeue*(X,ME) request is sent up the tree from the leaf process which generated it, with X carrying the element to a queue or dequeue and ME being a new mutual exclusion variable. If a queue process at a node can satisfy the request by matching it with a locally stored corresponding request, it does so. Otherwise it keeps a copy of the request in its local queue, and also sends a copy of it to its parent. A request is matched with a corresponding request by atomically testing the ME fields of the two requests to be variables and setting them to some value. When attempting to match the requests, the queue process also nondeterministically checks whether the ME field of any of the requests has been set by another queue process; this indicates that the request has been satisfied by some other queue, and so it is discarded.

Such a distributed queue can be used for dynamic load balancing, where workers off-load work by enqueing, and request work by dequeing [191]. It is very suitable for this application since requests are satisfied locally whenever possible, but eventually get to the most global queue (the root queue) if necessary.

Protected data-structures

Another important application of read-only variables is to protect processes communicating across trust boundaries. Consider an operating system process interacting with a possibly faulty user process via an incomplete message protocol, or by incrementally producing some data structure.

If the user process does not obey the protocol, and instead of waiting for the operating system process to instantiate some variable it instantiates this variable to some erroneous value, it may cause the operating system process to fail.

Several proposals were made to solve this problem. One is to restrict the type of communication protocols allowed between user processes and system processes, and provide user processes only with complete data-structures, with no 'holes' to mess with. This solution greatly decreases the flexibility of the interaction, and puts a heavy synchronisation and termination detection burden on the operating system.

Another solution is to isolate the components of the operating system interacting with user processes, and provide them with robust failure handling mechanisms. This solution also seems infeasible, since incomplete data structures can be passed asynchronously between system components, and therefore user processes may share variables with arbitrarily 'deep' operating system components.

Another solution, adopted by the operating system designed by ICOT, is to use specialized filter processes to monitor user-system interaction. These processes forward back and forth instantiations done by the interacting processes, as long as the user processes obey the protocol which the operating system expects. When a violation by the user is detected, the filter does not pass it further to the system. Foster [48] describes three techniques for achieving robustness in operating systems implemented in languages that do not support read-only variables: at-source (by transformation of user programs), en-route (by filters) or at-destination (by making system programs fail-safe). The second technique is shown to be generally the most effective.

Read-only variables allow a simpler solution [76]. An operating system component which produces a data-structure incrementally can protect the incomplete part of the data structure from outside intervention. This is done by making it read-only to its readers, and keeping the writable access to oneself. This is achieved by placing a read-only variable $X?$ in every 'hole' in the data structure, and keeping X. For example, a protected-stream producer can be defined as follows:

$$p([X|Xs?],\ldots) \leftarrow p(Xs',\ldots).$$

If, when $p(Xs,\ldots)$ is invoked, it has the only writable occurrence of its first argument Xs, this invariant will hold in all future iterations of the process, and no consumer can interfere with the stream production. If the *Message* itself is also produced incrementally, it could also be protected using the same technique.

<u>Discussion</u>

The advantages of read-only unification over matching is that it is a generalization of unification, rather than a special case of it: read-only unification in the absence of read-only variables is just unification. Hence read-only unification achieves both communication and synchronization with a single notion. Second, read-only unification is symmetric: unlike matching, it does not distinguish between the goal and the clause, and the read-only unification of any two terms behaves alike. Third, it is dynamic. Read-only variables can be embedded in any data-structure, hence synchronisation can be associated with data, not only with procedures.

Some of the disadvantages of read-only unification come from its strenght: Not being success stable makes it harder to analyze statically FCP(?) programs, and often makes FCP(?) less readable compared to programs using input matching. Its non-monotonic nature makes it more difficult to analyze theoretically, compared to languages which use only input matching and unification. Finally, it has some points of singularity (e.g. the unification of X with $X?$), which do not seem to have acceptable intuition behind them.

An alternative concept, called locks, was proposed by M. Miller and E.D. Tribble and formalized by Saraswat [156]. Its motivation was to provide more reasonable semantics to the unification $X=X?$. In FCP(?), this unification subtracts the writing capability from X, making it read-only. In the alternate proposal, its effect it to make both X and $X?$ writable. The ability of a read-only variable to become writable gives rise to both additional complications and additional programming techniques, though it has not been pursued to completion.

16. FCP(:,?) — An Integration of FCP(:) and FCP(?)

The language FCP(:,?) [100] attempts to integrate the convenience and efficiency of matching with the expressiveness of atomic test unification and read-only variables. In addition, it has the added pragmatic advantage over FCP(?) of being a superset of Flat GHC, FCP(|), and FCP(:), in the sense that every program in these languages would execute correctly as an FCP(:,?) program.

FCP(:,?) is as strong as any other language in the family, in the sense that there are natural embeddings of all languages in the family into it. It is the target language of the implementation effort at the Weismann Institute [99].

<u>Syntax</u>

The syntax of FCP(:,?) is the same as that of FCP(:), except that the tell and body parts may contain read-only variables.

<u>Semantics</u>

The semantics of the language is also the same as FCP(:), except that in the tell part read-only unification is used instead of ordinary unification. This is reflected in the try function of FCP(:,?), which is the same as that of FCP(:), except that it uses *mgu?* instead of *mgu*, and returns *suspend* if the read-only unification in the *tell* part suspends on a read-only goal variable, and *fail* if it fails or suspends on a read-only clause variable (since the latter kind of suspension is stable).

Programming in FCP(:,?)
As mentioned above, any FGHC, FCP(|) or FCP(:) would execute correctly as an FCP(:,?) program. The FCP(?) programs shown in the previous section easily translate into FCP(:,?). For example, the multiple-writer stream is written as follows:

```
write(M,Ms,Ms') ← true : Ms=[X?|Ms'] | M=X.
write(M,[_|Ms],Ms') ← write(M,Ms,Ms').
```

and the protected stream producer as follows:

```
p(Xs,...) ← true : Xs=[Message|Xs'?] | p(Xs',...).
```

17. Doc — "$X = X$ Considered Harmful"

The language Doc (Directed Oc) by Hirata [82], is a successor to Oc [81,83]. Oc is essentially $FGHC_{nav}$ with no guards. Doc is a further restriction, which follows the motto "$X = X$ considered harmful". Doc is a concurrent logic programming language in which every variable has at most one writer and at most one reader, i.e. one process which instantiates a variable, and one process that matches it. This restriction is enforced syntactically, by annotating each variable occurrence as either a writable or a read-only, and requiring that each variable may occur at most once in each mode in a clause.

The motivation for this restriction is that the cost of broadcasting information in a distributed environment may be too expensive to be supported at the language level.

Discussion
Although the removal of variable-to-variable unification from logic programming seems a rather drastic proposal, its effect is not fatal, and the resulting language is still usable. The techniques available in Doc (except for protected data structures) are a subset of those available in $FGHC_{nav}$. In particular, the short-circuit technique and any of the techniques relying on atomic unification are not available in Doc. Furthermore, broadcasting is not available in Doc, and should be implemented by an explicit distributor process, which receives a message and distributes it separately to each recipient. In addition, Doc's read-only annotation is a reminiscent of the read-only variable, and indeed it can employ the protected data-structures technique; actually, a Doc process *must* protect any incomplete structure it intends to produce, by the syntactic restrictions of the language. Because of the ability to specify protected data-structures, it seems that Doc cannot be embedded in a language that does not contain the equivalent of read-only variables.

An embedding of Doc in broadcast-free FCP(?)
The similarity of Doc's annotations to writable and read-only variables in FCP(?) is apparent. Indeed, it is natural to consider a subset of FCP(?), which may be called[27] *broadcast-free* FCP(?), in which every variable may occur at most once read-only and at most once writable in each clause. Doc programs can be trivially translated into broadcast-free FCP(?).

This translation is valid, in the sense that every computation of the resulting FCP(?) program corresponds to a possible computation of the source Doc program. However, the translation is not an embedding in the sense used so far. Since the read-only unification used in FCP(?) is atomic,

[27] The name was suggested by V.A. Saraswat.

some executions of a Doc program cannot be realized by the corresponding FCP(?) program. This can be remedied by further "decoupling" variables in the clause, as done in the embedding of $FGHC_{nav}$ in $FGHC_{av}$ in Section 10, which masks the atomicity of unification of FCP(?). For each variable X that occurs both writable and read-only in a clause, replace $X?$ by a new variable $Y?$, and add the goal *send*$(X?,Y)$ to the body of the clause. *send* is defined as follows. For every function symbol f/n in the program, $n \geq 0$, *send* has the clause:

```
send(f(X1,X2,...,Xn),f(Y1?,Y2?,...,Yn?)) ←
    send(X1?,Y1), send(X2?,Y2),..., send(Xn?,Yn).
```

We note that broadcast-free FCP(?) is still stronger than Doc, since it provides a variant of the short circuit technique. In this variant a ground message is sent around the circuit in a particular direction. Its arrival at the other end indicates termination.

18. Non-Flat Concurrent Logic Programming Languages: PARLOG, GHC, Concurrent Prolog, and CP(↓,|)

A concurrent logic programming language is non-flat if the guard of a clause may contain program defined predicates. Several of the flat languages described above — Flat GHC, Flat PARLOG, FCP(|,↓), and FCP(?) — were actually derived from their non-flat ancestors simply by restricting the guard to contain predefined predicates only.

The ability to define guard predicates implies that guard computations may be unbounded and, in general, may fail to terminate. Nevertheless, as in flat languages, a clause try is an atomic operation: it *succeeds*, *suspends*, or *fails*, and if it suspends or fails it leaves no trace of its attempt.

Two approaches were taken to ensure atomicity of a clause try; they are also reflected in the corresponding flat languages. One approach is to forbid guard computations from assigning goal variables. This way several clauses can be tried in parallel for the same goal without interference. This approach is taken by PARLOG and GHC, and is reflected in their flat subsets in the restriction that guards can only do matching, not unification. The second approach is to allow guard computations to assign goal variables, but to make such assignments visible only upon commitment. This is reflected in the FCP languages, which allow test unification in guards, but require the unification attempt to be atomic.

We discuss the non-flat languages informally. Transition systems for non-flat languages are given by Saraswat [154] and Levy [114].

18.1 PARLOG and GHC

PARLOG and GHC are similar in their requirement that guard computations do not instantiate goal variables, but differ in the way they realize this requirement. In PARLOG, a syntactic compile-time check, called a *safety check* is performed to ensure that the program has no computations in which guards instantiate goal variables [23]. Since the question whether a program is safe is undecidable in general [29], any algorithm for determining safety can only perform an approximate check, and if it correctly rejects all unsafe programs then it is bound to reject some safe programs as well. This leads to the awkward situation in which the set of legal PARLOG programs is either undecidable, or is determined by an algorithm, whose specification may be both quite complex and evolving. The practice of PARLOG programming seems to be that the safety check is not done, and the responsibility of producing safe programs is placed on the programmer's intuition.

The design of GHC [198] was influenced by an earlier design of PARLOG [24], called PARLOG83 in [145], which employed output assignment instead of unification, and by critical examination of Concurrent Prolog [196]. Rather than ruling out the possibility of the guard instantiating goal variables by a syntactic check, GHC ensures this with its synchronization rule. In fact, the sole synchronization rule of GHC states that a unification in the head or the guard that attempts to instantiate a variable in the goal suspends.

The implementation of this synchronization rule in full GHC requires recording for each variable which level in the process tree it 'belongs' to, which imposes considerable complications in the runtime data-structures and algorithms [114,185]. Therefore two subsets of GHC were identified: one is the flat subset, introduced in Section 10, another is the safe subset, defined as follows. A GHC program is *safe* if it has no execution in which a body unification suspends. Note that a Flat GHC program is trivially safe. Of course whether a GHC program is safe is also undecidable.

As in their flat subsets, the main difference between Safe GHC and PARLOG is the availability of sequential-And and sequential-Or in the latter.

Although PARLOG and GHC predate their flat subsets, there are almost no examples which show that the former languages are significantly more expressive than the latter ones. Perhaps the one interesting example is that of unbounded nondeterministic choice, implemented by recursion in the guard. Consider a process c(*In*,...) which has an unbounded list (or stream) of streams *In*. On each iteration, *c* wishes to extract one element from one of the streams, if such an element is ready, and iterate with *In'*, which contains the tail of that stream and the unmodified remaining streams. If all the streams close, the process terminates. Using non-flat guards, the program can be written in GHC as follows:

```
c(In,...) ←
    get(X,In,In') |
    ... do something with X ...
    c(In',...).
c(In,...) ←
    halt(In) | true.

get(X,[[X'|Xs]|In],In') ← In'=[Xs|In], X=X'.
get(X,[Xs|In],In') ← get(X',In,In'') | X=X', In'=[Xs|In''].

halt([[ ]|In]) ← halt(In).
halt([ ]).
```

The intermediate variables X' and In'' are needed to ensure that the recursive call of *get* does not suspend because of an attempt to instantiate the goal variables X or In'.

Note the difference between *get* and *halt*. Both are recursive, but *halt* iterates in the body, since it tests for a conjunctive condition (all streams have terminated), whereas *get* iterates in the guard, since it tests for a disjunctive one (there is an element on one of the streams).

The program cannot be specified directly in a flat language, since it requires nondeterminism of unbounded degree in process reduction. However, its purpose can usually be achieved using a merge network, which is specifiable in any flat language.

Embedding Safe GHC in FCP(:)

Safe GHC can be embedded in FCP(:) using a technique for compiling Or-parallelism into And-parallelism, developed by Codish and Shapiro [29]. The idea is to spawn And-parallel processes to evaluate Or-parallel guards, and thread these processes using two short-circuits: a success circuit, which reports the success of one of the guards, and a failure circuit, which reports the failure of all guards. The hierarchical And/Or tree is implemented by a hierarchy of success and failure circuits. The power of FCP(:) is needed since the method requires reflection on the failure of unification.

A mutual exclusion protocol ensures that at most one guard can commit for each goal. Although the mutual exclusion protocol used in the original embedding [29] relies on atomic unification (Section 14), the less efficient single-round mutual exclusion protocol (Section 7.4) can be used as well. The technique was later enhanced by Levy and Shapiro [116], into a compiler from Safe GHC to FCP(?).

The technique cannot be used to embed (unsafe) GHC in a flat language, since a correct implementation of GHC requires recording the guard in which a variable is allocated. This problem is further discussion in Section 20.

Embedding PARLOG in FCP(:)

The technique for compiling Or-parallelism into And-parallelism can be combined with the FCP(:) implementation of the control meta-call to form an embedding of Safe GHC + the control meta-call in FCP(:). It can be further combined with the techniques for embedding FP(&) and FP(;) in $FGHC_{av}$, to embed PARLOG in FCP(:).

18.2 Concurrent Prolog and CP(↓,|)

Concurrent Prolog [160] is the ancestor of FCP(?). Similarly, the language CP(|,↓) [154,157,151] is the ancestor of FCP(:). Unlike GHC and PARLOG, both allow guard computations to instantiate goal variables. However, to achieve atomicity of a clause try, these instantiations should not be visible outside the calling goal prior to the commitment of the clause. In order to perform Or-parallel clause evaluation in Concurrent Prolog, a 'multiple-environments' mechanism is necessary. This mechanism allows competing clauses to make temporary and hidden instantiations to goal variables, which become permanent and visible only upon commitment. Several approaches to the construction of such a mechanism were investigated [114], but none have lead to satisfactory results. The difficulty in constructing such a mechanism can be understood by examining the power of Concurrent Prolog. It can specify almost trivially an Or-Parallel Prolog interpreter, which simulates the don't-know nondeterminism of Prolog by recursion in guards.

An embedding of Or-parallel Prolog in Concurrent Prolog

The Or-parallel Prolog interpreter assumes that the Prolog program is represented by the Concurrent Prolog procedure, *clauses/2*, which returns on the call *clauses(A,Cs)* the list of clauses *Cs* potentially unifiable with the goal *A*. In principle *Cs* can be the entire Prolog program, but indexing on procedure names or even on goal arguments can be used to reduce the number of clauses returned. Each Prolog clause $A \leftarrow B_1,\ldots,B_k$ is translated into a term in the list *Cs* of the form $(A \leftarrow [B_1,\ldots B_k|Bs]\backslash Bs)$. Note that it represents the (possibly empty) body by a (possibly empty) difference-list of goals. Given this translation, an Or-Parallel Prolog interpreter can be written in Concurrent Prolog as follows[28]:

```
solve([ ]).
solve([A|As]) ←
    clauses(A,Cs), resolve(A,Cs?,As?).

resolve(A,[(A←Bs\As)|Cs],As) ←
    solve(Bs) | true.
resolve(A,[_|Cs],As) ←
    resolve(A,Cs,As) | true.
```

The interpreter as defined can return only one answer to a goal. This limitation, however, is shared also by Prolog meta-interpreters. To collect all solutions to a goal, a set abstraction is incorporated in Prolog. It is typically implemented by storing the solution found (using a side-effect) and inducing failure. The approaches of Ueda [200,201] and Shapiro [165], in comparison, naturally collect all solutions to a goal.

The simplicity of this interpreter indicates that the implementation of the multiple-environments mechanism of Concurrent Prolog is at least as difficult as the direct implementation of Or-Parallel Prolog. Presently it seems that the added complexity of Concurrent Prolog over its flat subset outweighs its added expressiveness.

[28] This interpreter is due to Kenneth M. Kahn.

PART IV. IMPLEMENTATIONS AND APPLICATIONS

19. Implementations of Concurrent Logic Programming Languages

Considerable effort has been invested in efficient implementations of concurrent logic programming languages, for both sequential and parallel computers.

19.1 Sequential implementations

We consider in depth implementation techniques for flat languages, then mention briefly techniques for non-flat languages.

There are several implementations of flat languages [[illegible],49,89]. All employ some variant of an abstract machine developed by a group at the Weismann Institute, first incorporated in an interpreter for FCP [128], and later refined and integrated with techniques for compiling unification, developed by Warren [206], within a compiler/emulator based implementation [89].

The sequential abstract machine

The key concepts of the machine are as follows. The machine represents the goal by an active queue and a set of suspension lists. Each process in the goal is either in the active queue or in one or more suspension lists. Each suspension list is associated with an unbound variable, and may consist of several processes.

The basic operation of the machine is to dequeue a process from the active queue, and try to reduce it with some clause in the program. This operation is called a *process try*. A process try is composed of a sequence of clause tries. In each clause try the try function of the process and the clause is computed (see Section 4.2). A process try succeeds if one of its clause tries succeeds; it suspends if none succeeds, but at least one suspends; it fails if all clause tries fail. When a process try succeeds the try substitution θ is computed. When a process try suspends a set of suspension variables is computed; a variable is included in the set if its being instantiated in the future may release some clause try from suspension, i.e. cause it to succeed or fail.

If a process try succeeds with a substitution θ then the goals in the body of the successful clause are added to the active queue, and θ is applied to the state of the computation. In addition, processes in suspension lists of variables in the domain of θ are moved to the active queue. If the process try suspends with a suspension set S then the process is added to the suspension lists of each of the variables in S. If the process try fails the machine halts with an error state.

Note that a process can suspend on several variables, and be activated and suspended several times before succeeding or failing. A mutual exclusion mechanism, described below, ensures that a process is activated at most once per suspended process try.

The machine is connected to one or more external input devices, realised by data streams, including a keyboard, and typically has a process consuming each stream. The machine terminates successfully when all external input streams are closed, and there are no processes left. It terminates with deadlock if all input streams are closed and only suspended processes are left.

The machine maintains all dynamic data structures in a single address space, called a *heap*. The heap grows when terms are allocated and processes are created, and shrinks by garbage-collection. The structures in the heap are variables, terms, process records, suspension records, activation records, and programs.

A variable is represented by one memory word, which is either empty or points to a suspension list. When a variable is instantiated to a term, its memory word becomes a reference (pointer) to the term unless the term can be stored in one word (e.g. an integer), in which case it is stored in place of the variable. Other terms are represented using standard techniques. A process with a predicate p/n is represented by $n+2$ words: one for the program counter, which points at the code of the procedure p/n, n words for the process arguments, and one word for chaining the process in

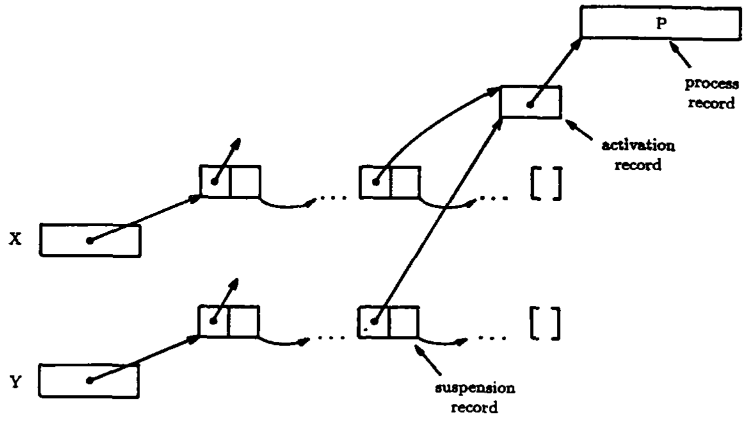


Figure 7: Suspending a process on two variables

the active queue. The active queue consists of chained processes. A suspension list consists of a list of suspension records (which could be list cells). Each suspension record points to an activation record and to the next suspension record if there is one. The activation record realizes the mutual exclusion mechanism which prevents multiple activations of the same process. It either points to a process record or is null, if the process has already been activated. If a process suspends on several variables, the suspension records in the suspension lists of these variables all point to the same process activation record, which in turn points to the process. The first variable to be assigned activates the process by enqueing it to the active queue, and sets its activation record to null. This prevents the other variables from re-activating this process. A process suspended on two variables is shown in Figure 7.

In addition to the heap, the machine has global registers for the active queue front and back, top-of-heap pointer, current process, current program counter, etc. In a language with test unification, the machine also has a trail. The trail is used to record assignments made during test unification in a clause try, so that they can be undone if the test unification subsequently suspends or fails. Unlike the standard Prolog trail, which needs to support deep backtracking, the trail in flat languages needs to support only shallow backtracking, and is reset on every clause try. As a result it can be rather small (e g. 256 words).

The machine employs several optimizations, the most important being tail-recursion optimization[29]. Each dequeued process is given a time-slice t (e.g. $t = 25$). When a process A with time-slice t is reduced to the processes $B_1,\ldots,B_k$, $k \geq 1$, then one of them, say B_1, reuses A's process record (if it is large enough), inherits the time-slice t-1, and is immediately tried if $t > 1$. For the other processes $B_2,\ldots,B_k$ new process records are allocated, and they are enqueued to the back of the active queue. If $t = 1$ then B_1 is also enqueued. This scheme maintains And-fairness while decreasing process switch and memory access (assuming some process arguments are maintained in processor registers during a time-slice).

To increase the chance of a process record being reused, minimal size records are allocated

[29] This name is kept for historical reasons. The optimization applies to any clause, not necessarily recursive, and not necessarily to the tail call.

(e.g. 10 words). In addition, free-lists of process records, suspension records, and activation records are maintained between garbage collections, to improve storage utilisation.

Implementations of non-flat languages

One way to achieve atomicity of a clause try in a non-flat language is to try and reduce goals in some order; when reducing a goal, try each clause in some order; and for each clause guard apply this execution algorithm recursively. This is the algorithm incorporated in the first interpreter for Concurrent Prolog, written in Prolog [160]. Variants of it were implemented on top of Prolog, both for Concurrent Prolog and for GHC and CP(|,↓,&) [203, 153]. This execution algorithm, however, does not satisfy any fairness requirements. For example, an attempt to reduce a faulty process (with a nonterminating guard) may block the rest of the system forever.

Several other executions algorithms for Concurrent Prolog which do not suffer from this problem were investigated [115,127,141]. Their complexity, however, seemed unacceptable, and was partially a motivation for the development of Flat Concurrent Prolog and the simpler non-flat languages, GHC, PARLOG, and CP(|,↓). An abstract machine for PARLOG was developed by Gregory *et al.* [68], and later optimized by Crammond [34]. Its basic design differs from the FCP abstract machine [89] in that it explicitly maintains a process tree. Another abstract machine for GHC was derived from the FCP machine by Levy [113]. Although GHC is simpler than Concurrent Prolog, its implementation still required fairly heavy machinery. Therefore Safe GHC was investigated, and a compiler from Safe GHC to FCP was developed [116].

Compilation of unification

The basic data manipulation of logic languages is unification. Warren [206] has developed a method for compiling unification efficiently by identifying its various special cases which are specified in a clause head, and generating special instructions for them.

Warren's scheme was designed for Prolog's general unification, and is applicable both to FCP's read-only unification [89], and to the input matching employed by FGHC [97] and PARLOG. Using it, an abstract machine along the lines described above can achieve the same uniprocessor performance as the Warren abstract machine for Prolog.

However, for input matching one can do better than Warren's scheme. The input matching component of a set of clauses of the same procedure can be jointly compiled into a decision tree, which combines shared matchings and finds more efficiently the set of applicable clauses [99].

Processor architectures

Two processor architectures specialized for the execution of a concurrent logic programming language, namely FCP(?), were developed. The first architecture, Carmel [74,75], takes the RISC approach. It augments a simple processor architecture with mechanisms to support the expensive or frequent operations of FCP(?). By carefully tuning the instruction set and processor architecture, impressive performance is obtained.

The second architecture, by Alkalaj and Shapiro [5], takes the view that internal concurrency in a processor combined with a carefully designed memory hierarchy is the key to high performance. The architecture consists of several specialized processing units, each with its own memory hierarchy. The reduction and tag processors are at the root of the hierarchy. They are supported by three additional processing units: an instruction processor, a data-trail processor, and a goal-management processor. The instruction processor employs standard techniques for instruction prefetching and caching. The data-trail processor employs a data cache enhanced to support shallow backtracking, required in the implementation of atomic test unification. The goal-management processor manages the top of the process queue in a way analogous to how a RISC processor manages the top of the activation stack. The goal-management processor manages process switching, spawning, activation, and suspension, using a bank of register windows. The execution algorithms of this architecture are specified using an FCP(?) program, by hardware description techniques developed by Suzuki [172] and Weinbaum and Shapiro [208]. The specification forms a working simulator of the architecture. The performance of this architecture is yet to be evaluated. How these two processors can be integrated in a multiprocessor architecture is an open question.

The PSI-II processor was designed for the execution of Prolog, but was re-microcoded to imp˙ ˙ment KL1 [195]. It is the building block of the multi-PSI parallel machine.

19.2 Parallel implementations

We review the concepts behind two types of parallel implementations: distributed and shared-memory. The implementations include a distributed implementation of FCP [190], a distributed implementation of FGHC [90], a distributed implementation of Flat PARLOG [47], and shared-memory implementation of PARLOG [34].

The core operation in these implementations is unification.

Distributed atomic unification

In a distributed implementation each processor executes a variant of the sequential abstract machine, described above, and takes special actions when a clause try involves variables shared with other processors. These actions realize a distributed unification algorithm.

Since non-variable terms are immutable data structures, they can be replicated upon demand throughout a processor network without any special consistency maintenance mechanisms. The writing on a variable, however, needs to be coordinated. In particular, in a language with atomic unification, a unification that involves writing on several variables should either succeed in writing on all of them, or write on none. Hence, from a distributed implementation point of view, an atomic unification is best viewed as an atomic transaction, which may read from and write to several logical variables. Standard database concurrency-control techniques for realizing atomic transactions can be adapted to the particular requirements of unification.

One approach, applicable to a network of processors without shared memory, is as follows. It uses the messages *read*, *lock*, *become_value*, and *become_local*. A variable shared by several processors is represented by a directed tree, with edges pointing towards the root. Each processor sharing a variable stores a node of the tree in its local memory, which contains the address of the node it is pointing to if it is not the root. An occurrence of a variable is called *remote* if it is an internal node in the tree; *local* if it is the root of the tree.

An attempt to read a shared remote variable is called a *read-fault*. A processor executing a process which has had a read-fault sends a *read* request up the tree, and adds the faulting process to the remote variable's suspension list. When a processor storing the root of the tree receives a *read* message, it operates as follows. If the variable has been assigned a term T, a *become_value*(T) message is sent in reply. If the variable is still unbound, the *read* request is stored in the variable's suspension list, and will be replied to when the variable is assigned.

A shared variable can be written only at its root. Write-permission is transferred between processors by changing and redirecting edges in the tree. A processor with a local shared variable (i.e. the root of a shared variable tree) may write on it when it pleases. It ensures that a unification that involves writing on several shared variables is atomic by not responding to messages, including *read* messages, while performing a clause try.

An attempt to write on a remote shared variable is called a *write-fault*. A processor execu.ing a process which has had a write-fault sends a *lock* message up the variable's tree, and suspends the faulting process on the remote variable. The processor receiving this message replies with a *become_value*(T) if the variable has already been assigned a term T, or with a *become_local*(*Reads*) if the variable is still unbound, and changes its local variable to be a remote variable pointing at the sender's variable. *Reads* is the (possibly empty) list of suspended *read* requests on the sender's local variable suspension queue, to which a request from the sender's own variable is added in case it has local processes suspended on it. The receiver of a *become_local*(*Reads*) message changes its variable from remote to local, wakes up all processes suspended on it, and adds the *Reads* to the variable's suspension list.

The scheme as described may result in livelock, if two processors keep sending *lock* requests to each other, and none accumulates enough local variables to perform a process reduction. To prevent this, a 2-phase-locking scheme can be incorporated [190,192]. The scheme requires additional bookkeeping by a write-faulting processor, but not additional messages. We do not describe its details here.

Another question to address is how to handle variable to variable unifications. One approach is to lock (i.e. make local) the two variables when assigning one to the other. This ensures that no cycles are created, but may cause superfluous contention in applications using the short-circuit technique. A second approach is to impose some ordering on variables, and to respect this ordering when unifying two variables. Another approach is not to prevent the creation of cycles, but to break them when they are detected.

Implementation of non-atomic unification and the meta-call construct
In languages without atomic unification, such as GHC, PARLOG, and their flat subsets, simpler algorithms than the one described above apply. For example, when unifying a remote variable X with a term T it is not necessary to bring X locally before assigning it; instead, a message can be sent to X, requesting it to unify with T. If the unification fails, the machine halts with an error state (or simply notes the inconsistency and proceeds).

Since either of these behaviors is not acceptable in a multi-tasking operating system, the meta-call construct, described in Section 10.3, was developed. The implementation of the meta-call construct must be integrated with the distributed unification algorithm in order to detect termination and to correctly ascribe failure. One approach, taken in the distributed implementation of FGHC [90], is to associate with every computation (invocation of a meta-call) a unique identifier, and maintain tables associating computation identifiers with the appropriate streams of the meta-call. When a unification fails, this fact is reported to the computation by placing a message on the appropriate stream. Since the short-circuit technique is not applicable, distributed termination of a computation is detected by maintaining an explicit distributed counter for each computation, at the language implementation level.

Foster [47] describes an alternative approach to the distributed implementation of the control metacall, which avoids the complexity of FGHC's distributed counters. Only uniprocessor computations are supported directly in the implementations and remote structure-to-structure unification operations are performed locally. Acknowledgement messages and message counting on individual nodes hence suffice for termination detection. Termination detection in distributed (multi-node) tasks is programmed in PARLOG using the usual techniques.

A shared-memory implementation
Crammond [34] describes a parallel implementation of PARLOG on a shared-memory multiprocessor. In this implementation each processor has its own data areas, although processors may access each other's areas in order to read the value of a shared variable, to assign a shared variable, or to take work (processes) from each other. A simple locking mechanism is employed, where a processor that modifies a shared object (e.g. a process queue or a shared logical variable) locks it, and a processor attempting to lock a locked object busy waits ("spins") until this object is unlocked. Since PARLOG does not have atomic unification, a processor needs at most one lock at a time, and hence this locking scheme does not result in deadlock. An extension of this implementation scheme to languages with atomic unification would require some concurrency control mechanism similar to the one discussed in this section above for distributed atomic unification.

A simple load balancing scheme is employed in this implementation, where a processor dequeues processes from its own queue as long as it is not empty, and dequeues from some other processor with a nonempty queue if its own queue is empty. Using such a scheme, this implementation obtained a speedup of up to 15 using 20 processors. Alternative load balancing schemes can be incorporated in this implementation with little difficulty.

An analysis of a shared-memory implementation of Flat GHC is reported by Tick [193].

19.3 Process to processor mapping

The question of how to map processes to processors is not unique to concurrent logic programming, and any general approach or solution may be applicable. Approaches to the problem fall into two general categories: methods in which the program itself (or programs associated with it) specify the mapping, and dynamic mapping techniques, incorporating load-balancing algorithms. Hybrid techniques are also possible.

We show how instances of the two approaches can be realized using distributed meta-interpreters. The interpreters are shown in FCP(:), although they could be written in any flat language.

Mapping with Turtle-programs

The use of Turtle-programs for mapping processes to processors was suggested and demonstrated in [163]. Assume that the parallel machine is a (finite or infinite) two dimentional grid. View each process as a LOGO-like Turtle, which has a position on the grid, and a heading. With each process activation (body goal) *P* we can association a LOGO-like Turtle program *TP*, as in *P*@*TP*. The meaning of the call *P*@*TP* is that *P* should have the position and heading obtained by applying *TP* to the position and heading of its parent process, and execute in the processor corresponding to that position. Processes without an associated Turtle program simply inherit their paren't position and heading.

Using this notation, a sequence of processes can be easily mapped on a sequence of processors. For example, consider the *vm* vectro-matrix multiplication program in Section 7.2. Adding the @forward Turtle program to the recursive call to *vm*, cause the inner-product processes *ip* to be placed on a sequence of adjacent processes:

% *vm*(*Xv*, *Ym*, *Zv*) ← multiplying the vector *Xv* by the matrix *Ym* gives the vector *Zv*.

```
    vm(_,[ ],Zv) ← Zv=[ ].
    vm(Xv,[Yv|Ym],Zv) ← Zv=[Z|Zv'],
        ip(Xv,Yv,Z), vm(Xv,Ym,Zv')@forward.
```

Mapping process arrays to processor arrays is just as easy. Consider the matrix multiplication program *mm* in Section 7.2. Adding the @*forward* Turtle program to the recursive call to *mm*, and @*right* to the initial call to *vm* maps the array of *ip* processes to an isomorphic array of processors:

% *mm*(*Xm*, *Ym*, *Zm*) ←
  Zm is the result of multiplying the matrix *Xm* with the transposed matrix *Ym*.

```
    mm([ ],_,[ ]).
    mm([Xv|Xm],Ym,[Zv|Zm]) ←
        vm(Xv,Ym,Zv)@right, mm(Xm,Ym,Zm)@forward.
```

The mapping of additional process structures is discussed in [163]. An alternative mapping strategy is described in [174]. Here show an enhanced distributed meta-interpreter which implements Turtle program mapping].

We assume that the underlying machine is a torus-connected mesh of processors (a virtual torus can be mapped on a two dimensional mesh by placing four virtual processors per physical one). The interpreter consists of a torus of *processor* processes. We assume that these processes are mapped to the underlying processors using the *torus* program shown in Section 7.2.

Each *processor* process has four outgoing streams to its neighbors. Its four incoming streams are merged into one. An interpreted process has a heading, and possibly also a Turtle program. A headed process is represented by a pair (*Goal*, *Heading*) where *Heading* is one of {*north*, *south*, *east*, *west*}. To a headed process (*G*, *H*) a Turtle program *TP* may be attached, as in (*G*, *H*)@*TP*. We assume the process in each processor is called *processor*(*In*, [*In*, *ToNorth*, *ToSouth*, *ToEast*, *ToWest*]), where the first argument is the merger of its neighbors' outgoing streams, and its second argument is the list of its five outgoing streams, one to itself and four to its neighbors. The *processor's* code is as follows:

```
processor([(Goal,Heading)|In],Out) ←
    reduce(Goal,Heading,In),
    processor(In,Out),
processor([(G,H)@TP|In],Out) ←
    route(G,H,TP,Out,Out'),
    processor(In,Out').
```

It receives goals on its input stream. If a goal has a Turtle program it routes it to its appropriate output stream. Otherwise it executes it locally. Its execution may result in new goals, possibly with Turtle programs. They are merged into its input stream, and treated normally. The meta-interpreter reduces goals, maintaining their heading, and when it encounters a goal with a Turtle program it sends it to the processor for routing.

```
reduce(true,_,_).
reduce((A,B),H,Out) ←
    reduce(A,H,Out), reduce(B,H,Out).
reduce(goal(A),H,Out) ←
    clause(A,B), reduce(B,H,Out).
reduce(A@TP,H,Out) ←
    write((A,H)@TP,Out).
```

The router is specified, without showing its code:

```
route(Goal,Heading,TP,Out,Out') ←
    Send Goal according to TP and Heading on the appropriate
    Out stream, with an updated heading and possibly with a residual
    Turtle program, and return the updated streams Out'.
```

The torus of *processor* processes can be mapped on an underlying torus using Turtle programs; but who will interpret these Turtle programs? Booting an initial process network on the processor network is necessary, and can be done using standard techniques. One solution is described in [187].

In this scheme the underlying parallel implementation of the language does not have to support remote process spawning in addition to distributed unification, since it is implemented at the language level by standard message passing between meta-interpreter (or runtime support) processes. Another mapping notation is described in [174].

Mapping with dynamic load-balancing

Dynamic load balancing requires that processors off-load work when they are too busy, and request work when they are idle. A good dynamic load balancing algorithm distributes work evently and with little overhead. If the underlying machine has a notion of locality, i.e. communication costs between processors are not uniform, then a dynamic load balancing algorithm should prefer local distribution of work over global one, when possible.

We show here a simple implementation of dynamic load balancing using a centralized queue. The scheme can be enhanced to use a distributed queue [165], and thus reduce contention and increase locality.

Assume a network of processors, and a *next* mapping command which places the process in the next processor in some processor ordering. A distributed meta-interpreter performing dynamic load balancing can be defined as follows:

```
processors(N,ToQ) ←
    queue(ToQ),
    processors'(N,ToQ)@next.

processors'(0,_).
processors'(N,ToQ) ←
    N>0 | N':=N-1,
    processor(ToQ),
    processors'(N',ToQ)@next.

processor(ToQ) ←
    reduce(true,ToQ).

reduce(true,ToQ) ←
    write(dequeue(A),ToQ,ToQ'),
    reduce(A,ToQ').
reduce((A,B),ToQ) ←
    write(enqueue(B),ToQ,ToQ'), reduce(A,ToQ').
reduce(A,ToQ) ←
    clause(A,B), reduce(B,ToQ).

queue(In) ←
    See Section 7.3.
```

Communication can be reduced, at the expense of slightly slower distribution of work, by placing a buffer in each processor. The buffer forwards requests to the global queue only if it overflows (has too many *enqueue* requests) or underflows (cannot satisfy a *dequeue* request). For example, in experiments made on a 16 processor computer on a particular application a buffer size of about 10 was found optimal [165].

Code management
General solutions to the code management problem are also applicable to concurrent logic programming languages. One approach to the problem is described in [187].

20. Applications of Concurrent Logic Programming Languages

Since their beginning, the design of concurrent logic languages was closely coupled with the development of prototype applications, which were used as feedback to the design process. The application programs were those which testified to the little difference between flat and non-flat languages from an expressiveness point of view. The systems programs were those which stretched the synchronization capabilities of logic languages to their limits, and provided examples where the power of atomic test unification and read-only unification shows through.

A description of numerous applications, as well as further references, can be found in the Concurrent Prolog book [164]. The book reports on the implementation of parallel and distributed algorithms, systems programming, and the implementation of embedded languages, among others. Other applications of concurrent logic languages include [35,67,98,99,118,137,144,145,148,157,158,172,181,183]. Combined, these applications witness to the generality and versatility of the concurrent logic programming approach.

PART V. CONCLUSIONS

21. Relation to Other Languages and Computational Models

21.1 Prolog, parallel logic languages and concurrent constraint languages

Prolog

Concurrent logic languages, as presently defined, are not an alternative to Prolog. They are, in a sense, lower level languages, which exhibit their strength mainly in the implementation of parallel algorithms, distributed systems, reactive systems, and in systems programming. Hence the question of the integration of these languages with higher-level languages in general, and with Prolog in particular, has received considerable attention.

One of the initial goals in the design of Concurrent Prolog [160] was the definition of a language which includes Prolog as a subset. It seemed that this goal was not realized in the initial design of the language, and hence this design was termed "a subset of Concurrent Prolog". Later, it was found out that an Or-parallel Prolog interpreter can be specified easily in that subset (the interpreter is shown in Section 18) and, as a consequence, that the original design did achieve this goal. However, the move to flat languages opened up again the question of the integration of Prolog and concurrent logic languages.

Two solutions were discussed in Section 14. One is to provide some interface between two separate languages: some form of Prolog, and some concurrent logic language [24,25]. Another is to embed some form of Prolog into a concurrent logic language [29,165,198,200]. A third solution is to provide some of the mechanism of concurrent logic language via extensions to Prolog, such as freeze [12] and wait declarations [132]. The problem with the first solution is that it really does not address the essence of the problem, namely to find an integrated solution in which the various strengths of logic programming can brought to bear. It is applicable to any two programming language, not necessarily logic programming ones. The problem with the second solution is performance. Techniques for efficient implementation of concurrent logic languages lag *one step* behind those of sequential Prolog, and there are claims that the algorithms employed in the embedding of Prolog in concurrent logic programming are not feasible. The third solution is largely limited to transformational applications, since it cannot change the basic fact that Prolog is not a reactive programming language.

CP(↓,|,&), Andorra, and Pandora

The synchronization and commitment mechanisms of concurrent logic languages are useful also in non-reactive applications. This motivated a different line of research — the design of non-reactive languages that attempt to supersede Prolog in expressiveness and performance, without being rooted in its sequential execution model.

Saraswat [150,153,157] investigated a parallel logic language, called CP(↓,|,&), that incorporates both don't-care and don't-know nondeterminism, and synchronization by input matching. Although an efficient implementation on top of sequential Prolog is described [Sad], the language seems even more difficult to implement "for real" than the non-flat languages discussed in Section 18.

Yang and Aiso [209,210] also propose a language with don't-care and don't-know nondeterminism, called P-Prolog, but use a different synchronization mechanism — the determinacy conditions described in Section 12 on P-Prolog$_x$.

Recently, an elegant integration of the ideas of P-Prolog and of Or-parallel Prolog, called the Andorra model, was proposed by D.H.D. Warren (personal communication), and integrated in the Andorra language [72]. The idea is as follows: reduce in parallel determinate goal atoms as long as possible (And-parallelism). When no determinate atoms remain, choose one atom for an *Or-split*.

Create two or more subgoals, one for each clause unifiable with the chosen atom, and continue in parallel reducing the resulting independent goals (Or-parallelism). Under the Andorra model pure logic programs may exhibit the synchronization behavior of concurrent logic programs, yet enjoy a complete proof procedure. If in an Or-split the leftmost atom is chosen, Andorra is more efficient (in terms of the number of reductions required) than ordinary Or-parallel Prolog, since it prunes the search space better.

The ideas in the Andorra model can also be employed in an implementation of the flat subset of CP($\downarrow$,|,&). Another recent proposal along these lines is Pandora [8] — a parallel logic language incorporating PARLOG-like synchronization, and a mechanism for specifying which goal atom to choose for an Or-split.

Concurrent constraint logic programming

The framework of constraint logic programming [92,31] proved in recent years to be a powerful generalization of logic programming, both from a theoretical and from a practical point of view. Maher [124] suggested using concepts of constraint logic programming to provide a logical characterization of synchronization in concurrent logic programming. The conditions for the success of input matching and guard checking of a goal atom A with a clause $A' \leftarrow G \mid B$ are customarily defined operationally, as in this paper. Maher showed how this condition can be specified logically, as the requirement that the accumulated constraint (corresponding to the accumulated substitution in our model) entails the existential constraint $(\exists)A{=}A' \wedge G$, where the existential quantifier ranges over the variables local to the clause. Saraswat [156,157] developed these ideas further. He developed a framework of concurrent constraint logic programming in which a computation progresses by agents that communicate by placing constraints in a global store and synchronize by checking that constraints are entailed by the store. Agents correspond to goal atoms, placing constraints correspond to unification, and checking constraints correspond to matching and guard checking in concurrent logic programming. Employing the concepts of consistency and entailment between partial information (i.e. constraints), Saraswat was able to provide a logical characterization of constraint-based constructs that correspond to non-atomic unification, atomic test unification, read-only unification, test-and-set, and others. Constraint logic programming offer a logical framework for dealing with domains other than Herbrand terms, such as boolean, integer, and real arithmetic. Saraswat showed how such domains and others can be incorporated in concurrent logic languages using this framework.

The initial work on concurrent constraint logic programming is very promising, and one may expect that it will have as much theoretical and practical impact on concurrent logic programming as constraint logic programming had on logic programming.

21.2 Distant relatives — Delta Prolog and Fleng

Delta Prolog

Delta Prolog [140] is Prolog augmented with CSP-like communication primitives. Delta Prolog is different from the other languages surveyed in two respects. First, it is not a logic programming language in the sense that a successful computation corresponds to a proof of a goal statement, and a partial computation corresponds to proofs of a conditional statement. Specifically, the role of the communication primitives of Delta Prolog in the declarative reading of programs is unclear. In concurrent logic languages the synchronisation primitives can be ignored in the declarative reading, since they affect only which answer substitution is found, but not the substitution itself. This is not the case in Delta Prolog. Although Delta Prolog can be given axiomatic semantics, this can be done for any programming language, not only for a logic programming one. The second difference between Delta Prolog and the other languages surveyed is that Delta Prolog is not reactive, since it may backtrack on communication.

It is not clear yet in which application area the particular features of Delta Prolog show their advantage.

Fleng

Fleng [134,135,136] is a simple concurrent programming language inspired by GHC and Kernel PARLOG [23]. Its syntax uses (guardless) Horn clauses. Like GHC, it uses goal/clause matching for synchronization, and its unification is non-atomic. Unlike GHC, unification, as well as any other primitive, reports termination.

Fleng has no notion of failure. Every primitive operation terminates and reports its termination status. For example, the unification primitive *unify(X, Y,Result)*, attempts to unify X and Y. If it succeeds it assigns *Result* the value *true*, if it fails it assigns the value *false*.

In spite of its appearance, Fleng is not a logic programming language, since not every successful computation corresponds to a proof of the goal statement. In particular, the goal *unify(a a,true)*, terminates successfully, but apparently so does *unify(a,b,true)*.

The insistence that a successful computation should correspond to a proof is not a mere nicety, and Fleng cannot simply drop the title of being a logic programming language and live happily ever after. The concept of failure serves the fundamental role of an exception mechanism in logic programming. In its absence, some other mechanism must be developed. As is evident from other languages [73], a sound exception handling mechanism is not a trivial component of a language, and its incorporation in Fleng would certainly complicate its semantics. Specifically, if Fleng's present exception handling mechanism (namely the *Result* variable of each primitive) cannot be used to report the exception, as in the call *unify(a,b,true)*, what exception should be raised? The most natural one is to fail the computation, which brings us back to square one...

If failure is reinstated in Fleng, then it becomes similar in expressiveness to KL1, since it can be naturally embedded in KL1 and vice versa.

21.3 Dataflow languages

Concurrent logic languages share with dataflow languages [1] single-assignment (or write-once) variables and dataflow synchronization. However, this is mainly a similarity in spirit, not in implementation. The basic operation that is synchronized by dataflow in concurrent logic languages is the process try. It corresponds typically to several tens, up to several hundreds, of conventional machine instructions. In contrast, the synchronized operation in dataflow models corresponds typically to one conventional machine instruction. This difference explains why realizations of concurrent logic languages on conventional hardware have acceptable synchronization overhead, whereas dataflow language seem to necessitate a specialized architecture.

Other differences between the two models is that dataflow languages are typically deterministic, whereas concurrent logic languages are not, and that dataflow languages and architectures are typically geared for scalar operations, whereas logic languages operate mainly on compound data-structures, which may contain logical variables.

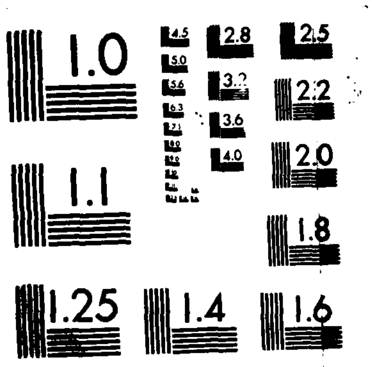
21.4 Functional languages

Much has been said on the relation between functional and logic languages [36]. In the context of concurrent programming, the major observation is that functional languages are, by design and ideology, transformational, rather than reactive. Functional programs denote time-independent functions from inputs to output, and notions of state, synchronization, communication, and nondeterminism are alien to them.

Functional programs can be parallelized, and often yield efficient parallel algorithms. However, without major extensions [7,52,53,54,69,77], which seem to undermine their original motivation and 'semantic elegance', functional programming languages cannot be used for the specification and implementation of reactive systems.

Concurrent logic languages, on the other hand, have explicit notions of processes, process state, communication, synchronization, and nondeterminism. Furthermore, processes can have several

AD-A213 958 THE FAMILY OF CONCURRENT LOGIC PROGRAMMING LANGUAGES 2/2
(U) WEIZMANN INST OF SCIENCE REHOVOT (ISRAEL) DEPT OF
APPLIED MATHEMATICS AND COMPUTER SCIENCE E Y SHAPIRO
UNCLASSIFIED MAY 89 CS89-08 F/G 12/5 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0

outputs, and inputs and outputs of processes can be combined into arbitrary process networks. These, combined with properties of the logical variable, seem to be the source of their power as concurrent languages; all are absent from the base model of functional languages.

In addition, it seems that there are usually simple translations from concurrent functional languages to concurrent logic languages [117]. Thus, a possible architecture of a parallel computer system, which provides both styles of programming, each for the application it suits best, is a system in which the base language is a concurrent logic programming language, which implements the underlying operating system and programming environment, and higher-level functional languages are implemented by translation to it. Such an architecture is proposed by [117].

21.5 Message-passing models of concurrency

The origins of concurrent logic programming languages can be traced back to the work of Kahn and MacQueen [94], which offered a model of concurrency based on deterministic asynchronous processes computing relations over data streams. van Emden and de Lucena [42] were intrigued by this notion, and showed how one can use logic programs to specify such processes. Clark and Gregory [20] took these ideas a crucial step further and, influenced by the notions of CSP [86,87], introduced synchronization and committed-choice nondeterminism into logic programs.

Concurrent logic languages are similar to CSP and Occam [91] in their notion of processes, nondeterminism, and synchronization via communication. They are similar to Occam, and different from CSP and Actors [78], in that processes communicate via 'ports' (realized by logical variables) rather than by naming the destination process or object.

One difference between CSP and Occam on the one hand and concurrent logic languages on the other hand are the type of communication and synchronization they employ. In the former communication is synchronous; in the latter asynchronous. In the former a communication channel is necessarily point-to-point. In the latter it is, in the general case, many-to-many. We find the added flexibility of the communication protocols available in concurrent logic languages over those of CSP and Occam quite apparent. The additional overhead entailed by this added flexibility is yet to be determined. Presently, it is not clear for which tasks Occam-like point-to-point synchronous protocols are inherently more efficient than the general asynchronous protocols employed in concurrent logic language, and vice versa.

Another fundamental difference is that CSP and Occam can operate on and communicate only "ground" data, whereas the ability to communicate and share incomplete data structures, i.e. data structures containing logical variables, is fundamental to concurrent logic languages, and is their main source of expressive power.

Being concrete programming languages, concurrent logic languages are not directly comparable to abstract computation models such as CCS. However, it seems that if one abstracts away the details of the data domain (i.e. terms and unification), and concentrates on the synchronization aspect of concurrent logic languages, then models which can be thought of as the asynchronous counterparts of CCS [129] emerge [65].

Although the syntax of CSP and CCS seems superficially different from that of concurrent logic languages, there is a close analogy between the basic operators of the two families, shown in Figure 8.

21.6 Concurrent object-oriented programming

The underlying operational model of concurrent logic languages resembles that of concurrent object-oriented models, such as Actors [78], in that both consist of a dynamic collection of light-weight processes, computing by performing local computations and exchanging messages. There are, however, several apparent differences.

First, Actor objects, like CSP processes, address each other by name, and not via channels.

CCS/CSP operators	Guarded Horn clauses
action prefix	guard
parallel composition	conjunction
restriction	clause head
choice	alternative clauses
relabeling	(implicit) clause renaming
recursion	recursion

Figure 8: Analogy between CCS/CSP operators and guarded Horn clauses

The advantage of channels over object names is modularity and abstraction; this had led Occam's designers to depart from CSP in this respect. It is easier to connect one process network to another by assigning output channels of one to input channels of the other, than by informing one the names or mail addresses of the appropriate processes in the other. Channels are also more abstract, since knowing a channel does not imply knowing who receives or sends messages on that channel. A process can have several input channels, which provide different access modes to its local data; this feature can be the basis of a capability system. Several processes may listen on the same channel, each handling a different set of messages, or handling a different aspect of a message.

If one is able to pass channels in messages, as in logic languages, than channels have another, perhaps more fundamental, advantage over name-based addressing. Process-names in messages, like incomplete messages, can be used for network reconfiguration. However, this is only one particular application of incomplete messages. The use of incomplete messages in the back-communication protocol, in dialogues, in the bounded-buffer protocol, in the duplex-stream protocol, and in others is based on the ability to allocate communication channels on the fly, and on the fact that the channel implicitly embeds some context information, which is used in the protocol. There is no natural way to achieve these effects in name-based addressing.

The drawback of concurrent logic languages, compared to Actor-like languages, is not their underlying operational model, but rather the verbose syntax required for expressing object-oriented programs. The description of an object with one input channel and some state variables in a concurrent logic language has the typical form:

```
p([Message|In],...state variables...) ←
    ... handle Message, update state variables...,
    p(In,...new state variables...).
```

Furthermore, when several processes share the same output channel ("talk to the same object"), then some protocol, such as the spawning of a merge network, need to be followed. This is in contrast to Actor-like languages, in which state variables are assumed not to change unless a change is stated explicitly, and explicit mergers need not be created in front of receiving objects, since they are assumed implicitly.

Another bookkeeping service provided automatically by object-oriented languages is object deallocation; when there are no more references to an object, it is deallocated, and its storage is reclaimed. In concurrent logic languages, unreferenced data-structures are reclaimed by garbage collection, but the conditions for process termination must be specified explicitly, by one or more unit clauses. Sometimes the burden of doing so manually should better be avoided.

A mechanism for detecting that a variable is referenced only by one process [17] can be used for garbage collecting processes: A process that detects that it is the only one referencing its input stream may perform some cleanup operations (*e.g.* close its output streams or unify its segment of a short-circuit) and terminate (K. Kahn, personal communication). Although the pragmatics of this mechanism is quite well understood, its logical semantics still needs to be worked out.

The question of the proper integration of inheritance in a concurrent object-oriented framework is still open. Delegation was suggested as a mechanism which is more suitable to a concurrent framework. As discussed in Section 7.6, objects which delegate incomprehensible messages can be specified in concurrent logic languages by augmenting the process with additional output stream, and adding a delegating clause which uses the *otherwise* construct. This mechanism, however, is also quite verbose.

These observations have lead to the design of new object-oriented languages, such as Vulcan [95], POLKA, and POOL [35]. These languages attempt to enjoy the best of both worlds. They adopt the channel concept of concurrent logic languages, but do not require explicit repetition of state variables, explicit mergers, or explicit delegation mechanism. Another important design consideration for these language was that their implementation be in terms of natural and efficient translations to concurrent logic languages. This would allow the exploitation of implementations of such languages, as well as support integration between applications that are best described by an object-oriented language, and applications that enjoy the full power of concurrent logic languages.

Consider the standard bank account example. In the Vulcan language, a process with the desired behavior is specified by the following program:

```
class(account, [Balance=0, Name="No Name Given", Errors,...]).

account :: deposit(Amount) →
    new Balance := Balance + Amount.

account :: balance(Balance).

account :: withdraw(Amount) →
    Balance ≥ Amount
    ifTrue new Balance := Balance - Amount
    ifFalse Errors : Overdrawn(Name, Balance, Amount, ...).
```

A more conservative solution in the same direction was to devise a new "surface syntax" for concurrent logic programs, rather than a completely new languages. The surface syntax, called ***logic programs with implicit variables*** [96], allows specifying only what has changed in the process's state during a transition, rather than the entire old and new states explicitly, as required by plain logic programs. In addition, it has a special notation to support stream communication, and array operations. For example, the bank account in FCP(|) with implicit variables would be specified as follows:

```
procedure account(In)+(Balance=0,Name="No Name Given",Errors,...).

account ←
    In ? deposit(Amount) |
    Balance' := Balance + Amount,
    account.

account ←
    In ? balance(Balance) |
    account.

account ←
    In ? withdraw(Amount),
    Balance ≥ Amount |
    Balance' := Balance - Amount,
    account.

account ←
    In ? withdraw(Amount),
Balance < Amount |
Errors ! Overdrawn(Name,Balance,Amount,...),
account.
```

The variable X' specifies the new value of the process argument X. The stream notation M ? Xs is a shorthand for the input matching Xs=[M|Xs'], and M ! Xs is a shorthand for the same unification.

Unlike in Vulcan this notation is employed only for stream, rather then channel [194] communication. An extension of this approach to incorporate channels as an abstract data type is being investigated.

21.7 Linda

Linda [13,2] is a set of primitives that operate concurrently on a multiset of tuples, called a Tuple Space. Tuples in a Tuple Space are accessed associatively using a degenerate form of unification between tuples and tuple templates. The basic operations are *out*(T) (insert a tuple T to the Tuple Space) *in*(T) (delete a tuple matching T, instantiating variables in T; block if a matching tuple is not available) and *rd*(T) (find a tuple matching T, instantiate variables in T). A fourth primitive , *eval*, support process forking. Augmenting a conventional sequential programming language with these Linda primitives results in a concurrent programming language in which processes communicate and synchronise via the Tuple Space.

A comparison of Linda and concurrent logic program is given in [13]. A critique of this comparison, which demonstrates an embedding of Linda's primitives in a variant of FCP(:) is given in [166].

21.8 Nondeterministic transition systems and UNITY

Nondeterministic transition systems are a natural method for specifying concurrent systems. Indeed, we have given the semantics of concurrent logic programming languages using nondeterministic transition systems. Recently, a notation was proposed for specifying concurrency called UNITY [16]. UNITY is based on unbounded iterative nondeterministic transitions.

Concurrent logic languages share with UNITY the goal of being a foundation for a general purpose concurrent programming language, the belief that the execution model of such a language should be abstract, rather then being tied with a concrete architecture, and the conviction that nondeterminism is an essential component in such a model. Another point in common between UNITY and the stronger concurrent logic languages is the size of the atomic operation: both the simultaneous assignment of UNITY and atomic unification in languages such as FCP(:) involve atomic transactions which read from and write to several variables.

One difference between UNITY and concurrent logic languages is the notion of a process. A UNITY program has one global state, and transitions operating on it, possibly concurrently; it does not have an explicit notion of a process. Concurrent logic programs have a natural notion of a process. However, this difference is only apparent. The notion of a process in concurrent logic programs is in the eyes of the beholder — it is not an inherent part of transition system of concurrent logic programs. Similarly, one can often identify "processes" in UNITY programs, if one so desires.

Another difference between UNITY and concurrent logic languages is the notion of termination. Concurrent logic programs terminate by explicit instructions. UNITY programs terminate implicitly, by reaching a fixpoint. One implication of this decision is that there is no distinction between successful termination and deadlock. We feel that this difference is mostly a matter of definition: one can define a different model of concurrent logic programs in which termination is by fixpoint; similarly, one can define "NITY", which is like UNITY except that there are explicit termination conditions. To our opinion, explicit termination is preferable both from the programmer's and from the implementor's point of view in both models.

We find the fundamental difference between UNITY and concurrent logic languages in the notion of a variable. In UNITY, variables are mutable; therefore a transition must exclude other transitions from writing on variables it reads from, and from accessing variables it writes to. In concurrent logic languages, variables are single-assignment, therefore no mutual exclusion mechanisms are required when reading a variable. The effect of mutable unshared variables can be achieved nonetheless in concurrent logic languages, as explained in Section 7, using iterative processes.

It seems that this fundamental difference is the source of another difference between UNITY and concurrent logic languages, namely their attitude to architectures. Although both are architecture independent, the gap between the general UNITY model and concrete architectures, such as a non-shared memory parallel computer, is sufficiently large that the authors of UNITY suggest that special sublanguages should be tailored for particular parallel architectures. In contrast, authors of concurrent logic languages believe their languages are suitable for all architectures. The burden of matching the application to the architecture resides solely with the algorithm designer and programmer. The belief, which is backed by the implementation efforts, is that concurrent logic languages are suitable for a wide range of architectures, including synchronous and asynchronous shared-memory computers, and tightly and loosely coupled non-shared memory computers. The difference between these architectures is not necessarily in the concurrent logic language suitable for them, but rather in the tradeoffs in communication and computation they offer, which determine which algorithms will better match a particular architecture.

This difference is not a *coincidence*. The single assignment property of logic variables means that even in a language with atomic test unification, locking of variables is very rarely necessary. Specifically, it is necessary almost only when the atomicity of unification is actually exploited to achieve some *synchronization task*. *For example*, in simple benchmarks of the parallel implementation of FCP(?) on the iPSC hypercube, more than 95% of the message trafic was associated with reading remote values (which does not require locking because of the single assignment property), and less than 5% with locking remote variables [191]. This is achieved without any special compilation or program analysis techniques. In UNITY, on the other hand, in the absence of additional information, every transition which accesses more than one variable requires locking all variables accessed. Therefore special sublanguages, which are structured to mimic the underlying architecture, have to be employed to make the model realistic.

On a methodological level, there are other differences between the approach of UNITY and that of concurrent logic languages. UNITY does not attempt to address questions of meta-programming and systems programming, or, more generally, how would a parallel computer system, whose base language is UNITY, be constructed. This question has been fundamental to concurrent logic programming from its beginning.

22. Conclusion

This survey attempted to convey the soundness, breadth, and potential of the logic programming approach to concurrency. Progress in the following can foster fully realizing this potential:

- Provide competitive implementations of concurrent logic languages for sequential, parallel and distributed computers.
- Develop simpler semantic foundations for concurrent logic languages.
- Exploit the simplicity of these languages to provide advanced program development environments and tools.
- Exploit the simplicity of these languages to provide advanced program analysis, transformation, and optimisation techniques, to aid in their efficient implementation.
- Further develop programming methodologies and techniques for these languages.
- Enhance concurrent logic programming by incorporating ideas and methods from constraint

logic programming.

- Further explore techniques for embedding higher-level languages, and design higher-level languages (such as parallel constraint programming languages) especially suitable for embedding in concurrent logic languages.

Acknowledgements

Comments by Andrew Davidson, Moreno Falaschi, Yishai Feldman, Ian Foster, Haim Gaifman, John Gallagher, Steve Gregory, David Harel, Ken Kahn, Michael Maher, Martin Nilsson, Vijay Saraswat, Jiro Tanaka, Kazunori Ueda, Peter Wegner, Eyal Yardeni, and the anonymous referees are gratefully acknowledged.

References

[1] Ackerman, W.B., Data flow languages, *IEEE Computer* 15(2), pp.15-25, 1982.

[2] Ahuja, S., Carriero, N., and Gelernter, D., Linda and friends, *IEEE Computer* 19(8), 26–34, 1986.

[3] Ali, K.A.M., Or-parallel execution of Prolog on a multi-sequential machine, SICS Technical Report, 1986.

[4] Ali, K.A.M., Or-parallel execution of Horn clause programs based on the WAM and shared control information, SICS Technical Report, 1986.

[5] Alkalaj, L., and Shapiro, E., An architectural model for a Flat Concurrent Prolog processor, in Bowen, K. and Kowalski, R.A. (eds.), *Proc. 5th International Conference Symposium on Logic Programming*, pp. 1277–1297, MIT Press, 1988.

[6] Apt, K.R., and van Emden, M.H., Contributions to the theory of logic programming, *J. ACM* 29(3), pp. 841–863, 1982.

[7] Bage, G., and Lindstrom, G., Committed choice functional programming, *Proc. International Conference on Fifth Generation Computer Systems*, pp. 666–674, ICOT, Tokyo, 1988.

[8] Bahgat, R., and Gregory, S., Pandora: Non-deterministic parallel logic programming, To appear in *Proc. 6th International Conference on Logic Programming*, Lisbon, MIT Press, 1989.

[9] de Bakker, J.W., and Kok, J.N., Uniform abstraction, atomicity and contractions in the comparative semantics of Concurrent Prolog, *Proc. International Conference on Fifth Generation Computer Systems*, pp. 347–355, ICOT, Tokyo, 1988.

[10] Baron, U., Chassin de Kergommeaux, J., Hailperin, M., Ratcliffe, M., Robert, P., Syre, J.-C., and Westphal, H., The parallel ECRC Prolog system PEPSys: An overview and evaluation results, *Proc. International Conference on Fifth Generation Computer Systems*, pp. 841–850, ICOT, Tokyo, 1988.

[11] Bowen, D.L., Byrd, L., Pereira, L.M., Pereira, F.C.N., and Warren, D.H.D., PROLOG on the DECSystem-10 User's Manual, Technical Report, Department of Artificial Intelligence, University of Edinburgh, 1981.

[12] Carlsson, M., Freeze, indexing, and other implementation issues in the WAM, in Lassez, J.-L. (ed.), *Proc. 4th International Conference on Logic Programming*, pp. 40–58, MIT Press, 1987.

[13] Carriero, N., and Gelernter, D., Linda in context, *Comm. ACM* 32(4), 444–458, 1988.

[14] Chandy, M., and Lamport, L., Distributed snapshots: determining global states of distributed systems, *Transactions on Computer Systems* 3(1), pp. 63–75, 1985.

[15] Chandy, K.M., and Misra, J., A paradigm for detecting quiescent properties in distributed computations, in Apt, K.R. (ed.), *Logics and Models of Concurrent Systems*, pp. 325–342, Springer-Verlag, 1985.

[16] Chandy, K.M., and Misra, J., *Parallel Program Design*, Addison-Wesley, 1988.

[17] Chikayama, T., and Kimura, Y., Multiple reference management in Flat GHC, in Lassez, J.-L. (ed.), *Proc. 4th International Conference on Logic Programming*, pp. 276–293, MIT Press, 1987.

[18] Chikayama, T., Sato, H., and Miyazaki, T., Overview of the parallel inference machine operating system (PIMOS), *Proc. International Conference on Fifth Generation Computer Systems*, pp. 230–251, ICOT, Tokyo, 1988.

[19] Clark, K.L., Predicate logic as a computational formalism, Research Report DOC 79/59, Department of Computing, Imperial College, London, 1979.

[20] Clark, K.L., and Gregory, S., A relational language for parallel programming, *Proc. ACM Conference on Functional Languages and Computer Architecture*, pp. 171–178, 1981. Also Chapter 1 in [164].

[21] Clark, K.L., and Gregory, S., PARLOG: A parallel logic programming language, Research

Report DOC 83/5, Department of Computing, Imperial College, London, 1983.

[22] Clark, K.L., and Gregory, S., Notes on systems programming in PARLOG, *Proc. International Conference on Fifth Generation Computer Systems*, pp. 299–306, ICOT, Tokyo, 1984.

[23] Clark, K.L., and Gregory, S., Notes on the implementation of PARLOG, Research Report DOC84/16, 1984. Also in *J. Logic Programming* 2(1), pp. 17–42, 1985.

[24] Clark, K.L., and Gregory, S., PARLOG: Parallel programming in logic, *ACM TOPLAS* 8(1), pp. 1–49, 1986. Revised as Chapter 3 in [164].

[25] Clark, K.L., and Gregory, S., PARLOG and PROLOG united, in Lassez, J.-L. (ed.), *Proc. 4th International Conference on Logic Programming*, pp. 927–961, MIT Press, 1987.

[26] Clark, K.L., McCabe, F.G., and Gregory, S., IC-PROLOG — language features, in Clark, K.L., and Tärnlund, S.-A. (eds.), *Logic Programming*, pp. 253–266, Academic Press, London, 1982.

[27] Clark, K.L., and Tärnlund, S.-A., A first-order theory of data and programs, in Gilchrist, B. (ed.), *Information Processing* 77, pp. 939–944, North-Holland, 1977.

[28] Clocksin, W.R., and Alshawi, H., A method for efficiently executing Horn clause programs using multiple processors, Technical Report, Department of Computer Science, Cambridge University, Cambridge, 1986.

[29] Codish, M., and Shapiro, E., Compiling Or-parallelism into And-parallelism, *New Generation Computing* 5(1), pp. 45–61, 1987. Also Chapter 32 in [164].

[30] Codish, M., Gallagher, J., and Shapiro, E., Using safe approximations of fixed points for analysis of logic programs, *Proc. META88, Workshop on Meta-Programming in Logic Programming*, Bristol, 1988.

[31] Colmerauer, A., Opening the Prolog-III universe, *BYTE Magazine* 12(9), August 1987.

[32] Costa, G., and Stirling, C., Weak and strong fairness in CCS, *Information and Computation* 73, pp.207–244, 1987.

[33] Crammond, J., A comparative study of unification algorithms for Or-parallel execution of logic languages, *Proc. IEEE International Conference on Parallel Processing*, pp. 131–138, 1985.

[34] Crammond, J.A., Implementation of committed choice logic languages on shared memory multiprocessors, Ph.D. Thesis, Department of Computer Science, Heriot-Watt University, May 1988. Also Technical Report PAR 88/4, Department of Computing, Imperial College, London, 1988.

[35] Davison, A., POOL: A PARLOG object oriented language, Department of Computing, Imperial College, London.

[36] DeGroot, D., and Lindstrom, G. (eds.), *Logic Programming — Functions, Relations and Equations*, Prentice-Hall, New Jersey, 1986.

[37] Dijkstra, E.W, Hierarchical ordering of sequential processes, *Acta Informatica* 1, 115–138, 1971.

[38] Dijkstra, E.W., Guarded commands, nondeterminacy, and formal derivation of programs, *CACM* 18(8), pp. 453–457, 1975.

[39] Dijkstra, E.W., *A Discipline of Programming*, Prentice-Hall, New Jersey, 1976.

[40] Dijkstra, E.W., and Scholten, C.S., Termination detection for diffusing computations, *Information Processing Letters* 11(1), pp. 1–4, 1980.

[41] van Emden, M.H., and Kowalski, R.A., The semantics of predicate logic as a programming language, *J. ACM* 23(4), pp. 733–742, 1976.

[42] van Emden, M.H., and de Lucena, G.J., Predicate logic as a language for parallel programming, in Clark, K.L., and Tärnlund, S.-A. (eds.), *Logic Programming*, pp. 189–198, Academic Press, London, 1982.

[43] Ershov, A.P., et al. (eds.), Special Issue: Selected papers from the Workshop on Partial Evaluation and Mixed Computation, 1987, *New Generation Computing* 6(2,3), 1988.

[44] Falaschi, M., and Levi, G., Finite failures and partial computations in concurrent logic

languages, *Proc. International Conference on Fifth Generation Computer Systems*, pp. 364–373, ICOT, Tokyo, 1988.

[45] Falaschi, M., Levi, G., Martelli, M., and Palamidessi, G., A new declarative semantics for logic languages, in K. Bowen and R.A. Kowalski (eds.), *Proc. 5th International Conference Symposium on Logic Programming*, pp. 993–1005, MIT Pr ess, 1988.

[46] Foster, I., Efficient metacontrol in parallel logic programming, Research Report PAR 87/18, Department of Computing, Imperial College, London, 1987.

[47] Foster, I., Parallel implementation of PARLOG, *Proc. International Conference of Parallel Processing*, 1988.

[48] Foster, I., *PARLOG as a Systems Programming Language*, Ph.D. Thesis, Department of Computing, Imperial College, London, 1988.

[49] Foster, I., and Taylor, S., Flat Parlog: A basis for comparison, *International J. of Parallel Programming* 16(2), 1987.

[50] Frances, N., *Fairness*, Springer-Verlag, 1987.

[51] Frances, N., and Rodeh, M., Achieving distributed termination without freezing, *IEEE Transactions on Software Engineering* SE-8(3), pp. 359–385, 1982.

[52] Friedman, D.P., and Wise, D.S., Aspects of applicative programming for parallel processing, *IEEE Trans. on Computers* C-27(4), pp. 289–296, 1978.

[53] Friedman, D.P., and Wise, D.S., An approach to fair applicative multiprogramming, in Kahn, G. (ed.), *Semantics of Concurrent Computation*, *LNCS* 70, pp. 203–226, Springer-Verlag, 1979.

[54] Friedman, D.P., and Wise, D.S., An indeterminate constructor for applicative programming, *Conference Record 7th ACM Symposium on Principles of Programming Languages*, pp. 245–250, 1980.

[55] Fuchi, K., and Furukawa, K., The role of logic programming in the Fifth Generation Computer Project, *New Generation Computing* 5(1), pp. 3–28, 1987.

[56] Furukawa, K., and Ueda, K., GHC process fusion by program transformation, *Proc. 2nd Conference on Japan Society of Software Science and Technology*, pp. 89–92, 1985.

[57] Furukawa, K., Okumura, A., and Murakami, M., Unfolding rules for GHC programs, in Bjorner, D. et al. (eds.), *Proc. Workshop on Partial and Mixed Computation*, Gl. Avernaes, 1987.

[58] Futamura, Y., Partial evaluation of computation process — an approach to a compiler-compiler, *Systems, Computers, Controls* 2(5), pp. 721–728, 1971.

[59] Gaifman, H., Maher, M.J., and Shapiro, E., Existential constraints, reactive behaviors, and fully abstract compositional semantics of concurrent logic programs, Technical Report, Department of Computer Science, The Weizmann Institute of Science, Rehovot, 1 989.

[60] Gallagher, J., *An Approach to the Control of Logic Programs*, Ph.D. Thesis, Department of Computer Science, Trinity College, Dublin, 1983.

[61] Gallagher. J., Transforming logic programs by specialising interpreters, *Proc. 7th European Conference on Artificial Intelligence*, pp. 109–122, Brighton, 1986.

[62] Gallagher. J., Codish, M., and Shapiro, E., Specialisation of Prolog and FCP programs using abstract interpretation, *New Generation Computing* 6, pp. 159–186, 1988.

[63] Gaifman, H., and Shapiro, E., Fully abstract compositional semantics for logic programs, *Proc. ACM Symposium on Principles of Programming Languages*, pp. 134–142, 1989.

[64] Gaifman, H., and Shapiro, E., Proof Theory and Semantics of Logic Programs, To appear in *Proc. IEEE Symposium on Logic in Computer Science*, 1989.

[65] Gerth R., Codish M., Lichtenstein Y., and Shapiro E., Fully abstract denotational semantics for Flat Concurrent Prolog, *Proc. IEEE Symposium on Logic in Computer Science*, pp. 320–333, 1988.

[66] Gregory, S., *Parallel Logic Programming in PARLOG*, Addison-Wesley, 1987.

[67] Gregory, S., Neely, R., and Ringwood, G.A., PARLOG for specification, verification and

simulation, in Koomen, C.J., and Moto-Oka, T. (eds.), *Proc. 7th International Symposium on Computer Hardware Description Languages and their Application s*, pp. 139–148, Elsevier/North-Holland, Amsterdam, 1985.

[68] Gregory, S., Foster, I.T., Burt, A.D., and Ringwood, G.A., An abstract machine for the implementation of PARLOG on uniprocessors, *New Generation Computing* 6(4), 389–420, 1986.

[69] Halstead, R.H., MultiLisp — A language for concurrent symbolic computation, *ACM Trans. on Programming Languages and Systems* 7(4), pp. 501–538, 1985.

[70] Harel, D. (ed.), *Algorithmics: The Spirit of Computing*, Addison-Wesley, 1987.

[71] Harel, D., and Pnueli, A., On the development of reactive systems, in Apt, K.R. (ed.), *Logics and Models of Concurrent Systems*, Springer Verlag, 1985.

[72] Haridi, S., and Brand, P., ANDORRA Prolog — an integration if Prolog and committed choice languages, *Proc. International Conference on Fifth Generation Computer Systems*, pp. 745–754, ICOT, Tokyo, 1988.

[73] Harper, R., MacQueen, D., and Milner, R., Standard ML, Technical Report ECS-LFCS-86-2, University of Edinburgh, 1986.

[74] Harsat, A., and Ginosar, R., CARMEL – a VLSI architecture for Flat Concurrent Prolog, EE PUB. Technical Report, Department of Computer Science, Technion, Haifa, 1987.

[75] Harsat, A., and Ginosar, R., CARMEL-2: a second generation VLSI architecture for Flat Concurrent Prolog, *Proc. International Conference on Fifth Generation Computer Systems*, pp. 962–969, ICOT, Tokyo, 1988.

[76] Hellerstein, L., and Shapiro, E., Implementing parallel algorithms in Concurrent Prolog: The MAXFLOW experience, *J. Logic Programming* 3(2), pp. 157–184, 1984. Also Chapter 9 in [164].

[77] Henderson, P., Purely functional operating systems, in Darlington, J., Henderson, P., and Turner, D. (eds.), *Functional Programming and Its Applications*, Cambridge University Press, 1982.

[78] Hewitt, C., A universal, modular actor formalism for artificial intelligence, *Proc. International Joint Conference on Artificial Intelligence*, 1973.

[79] Hewitt, C., The challenge of open systems, *Byte Magazin*, pp. 223–242, 1985.

[80] Hill, R., LUSH-resolution and its completeness, DCL Memo 78, Department of Artificial Intelligence, University of Edinburgh, 1974.

[81] Hirata, M., Letter to the editor, *SIGPLAN Notices*, pp. 16–17, May 1986.

[82] Hirata, M., Programming language Doc and its self-description, or, X=X is considered harmful, *Proc. 3rd Conference of Japan Society of Software Science and Technology*, pp. 69–72, 1986.

[83] Hirata, M., Parallel list processing language Oc and its self-description, *Computer Software* 4(3), pp. 41–64, 1987 (in Japanese).

[84] Hirsch, M., Silverman, W., and Shapiro, E., Computation control and protection in the Logix system, Chapter 20 in [164].

[85] Hoare, C.A.R., Monitors: an operating systems structuring concept, *Comm. ACM* 17(10), pp. 549-557, 1974.

[86] Hoare, C.A.R., Communicating sequential processes, *Comm. ACM* 21(8), pp. 666-677, 1978.

[87] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall, New Jersey, 1985.

[88] Hopcroft, J.E., and Ullman, J.D., *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[89] Houri, A., and Shapiro, E., A sequential abstract machine for Flat Concurrent Prolog, Chapter 38 in [164].

[90] Ichiyoshi, N., Miyasaki, T., and Taki, K., A distributed implementation of Flat GHC on the Multi-PSI, in Lassez, J.-L. (ed.), *Proc. 4th International Conference of Logic Programming*, pp. 257–276, MIT Press, 1987.

[91] INMOS Ltd., *OCCAM Programming Manual*, Prentice-Hall, New Jersey, 1984.

[92] Jaffar, J., and Lassez, J-L., Constraint logic programming, *ACM Symposium on Principles of Programming Languages*, Munich, 1987.

[93] Johnson, S.D., Circuits and systems: Implementing communications with streams, Technical Report 116, Computer Science Department, Indiana University, 1981.

[94] Kahn, G., and MacQueen, D., Coroutines and networks of parallel processes, in Gilchrist, B. (ed.), *Information Processing 77, Proc. IFIP Congress*, pp. 993–998, North-Holland, 1977.

[95] Kahn, K., Tribble, E.D., Miller, M., and Bobrow, D.G., Vulcan: Logical concurrent objects, in Shriver, B., and Wegner, P. (eds.), *Research Directions in Object-Oriented Programming*, MIT Press, 1987. Also Chapter 30 in [164].

[96] Kahn, K., Silverman, W., and Shapiro, E., Logic programs with implicit variables, unpublished, 1988.

[97] Kimura, Y., and Chikayama, T., An abstract KL1 machine and its instruction set, *Proc. IEEE Symposium on Logic Programming*, pp. 468–477,San Francisco, 1987.

[98] Kishimoto, M., et al., An evaluation of the FGHC via practical application programs, ICOT Technical Report TR-232, Institute for New Generation Computer Technology, Tokyo, 1987.

[99] Kliger, S. and Shapiro, E., A decision tree compilation algorithm for FCP(|,:,?), in Bowen, K. and Kowalski, R.A. (eds.), *Proc. 5th International Conference Symposium on Logic Programming*, pp. 1315–1336, MIT Press, 1988.

[100] Kliger, S., Yardeni, E., Kahn, K., and Shapiro, E., The language FCP(:,?), *Proc. International Conference on Fifth Generation Computer Systems*, pp. 763–773, ICOT, Tokyo, 1988.

[101] Kohda, Y., and Tanaka, J., Deriving a compilation method for parallel logic languages, *Logic Programming, LNCS* **315**, pp. 80–94, Springer-Verlag, 1988.

[102] Kowalski, R.A., *Logic for Problem Solving*, Elsevier, North-Holland, 1979.

[103] Kung, H.T., Why systolic architectures?, *IEEE Computer* **15**(1), pp. 37–46, 1982.

[104] Kusalik, A.J., Bounded-wait merge in Shapiro's Concurrent Prolog, *New Generation Computing* **1**(**12**), pp. 157–169, 1984.

[105] Lai, T.H., Termination detection for dynamically distributed systems with non-first-in-first-out communication, *J. Parallel and Distributed Computing* **3**, pp. 577–599, 1986.

[106] Lam, M., and Gregory, S., PARLOG and ALICE: a marriage of convenience, in Lassez, J.-L. (ed.), *Proc. 4th International Conference on Logic Programming*, pp. 294–310, MIT Press 1987.

[107] Lamport, L., Time, Clocks, and the ordering of events in a distributed system, *Communications of the ACM*, July 1978.

[108] Lassez, J.-L., Maher, M.J., and Marriot, K., Unification revisited, in J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, pp. 587–626, Morgan Kaufmann, 1987.

[109] Levi, G., Models, unfolding rules and fixpoint semantics, in Bowen, K. and Kowalski, R.A. (eds.), *Proc. 5th International Conference Symposium on Logic Programming*, pp. 1649-1665, MIT Press, 1988.

[110] Levi, G., and Palamidessi, C., An approach to the declarative semantics of synchronization in logic languages, in Lassez, J.-L. (ed.), *Proc. 4th International Conference on Logic Programming*, pp. 877–893, MIT Press, 1987.

[111] Levi, G., and Sardu, G., Partial evaluation of meta-programs in a multiple world's logic language, in Bjorner, D., Ershov, A.P. and Jones, N.D. (eds.), *Workshop on Partial Evaluation and Mixed Computation*, Gl. Avernaes, 1987.

[112] Levy, J., A unification algorithm for Concurrent Prolog, Tärnlund, S.-A. (ed.), *Proc 2nd International Conference on Logic Programming*, pp. 333–341, Uppsala, 1984.

[113] Levy, J., A GHC abstract machine and instruction set, in Shapiro, E. (ed.), *Proc. 3rd International Conference on Logic Programming, LNCS* **225**, pp. 157–171, Springer-Verlag,

1986.

[114] Levy, J., *Concurrent Prolog and Related Languages*, Ph.D. Thesis, Department of Computer Science, The Weizmann Institute of Science, Rehovot, 1988.

[115] Levy, J., and Friedman, N., Concurrent Prolog implementations — two new schemes, Technical Report CS86-13, Department of Computer Science, The Weizmann Institute of Science, Rehovot, 1986.

[116] Levy, J., and Shapiro, E., Translation of Safe GHC and Safe Concurrent Prolog to FCP, Chapter 33 in [164].

[117] Levy, J., and Shapiro, E., CFL — A concurrent functional language embedded in a concurrent logic programming environment, Chapter 35 in [164].

[118] Lichtenstein, Y., and Shapiro, E., Concurrent algorithmic debugging, *Proc. ACM Workshop on Parallel Debugging*. Also Technical Report CS87-20, Department of Computer Science, The Weizmann Institute of Science, Rehovot, 1987.

[119] Lichtenstein, Y., and Shapiro, E., Representation and enumeration of Flat Concurrent Prolog computations, Chapter 27 in [164].

[120] Lichtenstein, Y. and Shapiro, E., Abstract algorithmic debugging, in Bowen, K. and Kowalski, R.A. (eds.), *Proc. 5th International Conference and Symposium on Logic Programming*, pp. 1315–1336, MIT Press, 1988.

[121] Lloyd, J.W., *Foundations of Logic Programming*, Second Edition, Springer-Verlag, 1987.

[122] Lusk, E., Butler, R., Disz, T., Olson, R., Overbeek, R., Stevens, R., Warren, D.H.D, Calderwood, A., Szeredi, P., Haridi, S., Brand, P., Carlsson, M., Ciepielewski, A., and Hausman, B., The Aurora Or-Parallel Prolog system, *Proc. International Con ference on Fifth Generation Computer Systems*, pp. 819–830, ICOT, Tokyo. 1988.

[123] Maher, M.J., Equivalences of logic programs, in J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, pp. 627–658, Morgan Kaufmann Publishers, Los Altos, 1987.

[124] Maher, M.J., Logic semantics for a class of committed-choice programs, in Lassez, J.-L. (ed.), *Proc. 4th International Conference on Logic Programming*, pp. 858–876, MIT Press, 1987.

[125] Manna, Z., and Pnueli, A., Specification and verification of concurrent programs by ∀-automata, Report STAN-CS-88-1230, Department of Computer Science, Stanford University, Stanford, 1988.

[126] Mattern, F., Algorithms for distributed termination detection, *Distributed Computing* **2**, pp. 161–175, 1987.

[127] Miyazaki, T., Takeuchi, A., and Chikayama, T., A sequential implementation of Concurrent Prolog based on the shallow binding scheme, *IEEE Symposium on Logic Programming* pp. 110-118, 1985. Also Chapter 37 in [164].

[128] Mierowsky, C., Taylor, S., Shapiro, E., Levy, J., and Safra, S., The design and implementation of Flat Concurrent Prolog, Technical Report CS85-09, Department of Computer Science, The Weizmann Institute of Science, Rehovot, 1985.

[129] Milner, R., *A Calculus of Communicating Systems*, *LNCS* **92**, Springer-Verlag, 1980.

[130] Misra, J., Distributed discrete-event simulation, *Computing Surveys* **18**(1), pp. 39–65, 1986.

[131] Murakami, M., A declarative semantics of parallel logic programs with perpetual processes, *Proc. International Conference on Fifth Generation Computer Systems*, pp. 374–381, ICOT, Tokyo, 1988.

[132] Naish, L., *MU-Prolog 3.1db Reference Manual*, Internal Memorandum, Department of Computer Science, University of Melbourne, 1984.

[133] Nilsson, M., and Tanaka, H., Fleng Prolog — The language which turns supercomputers into Prolog machines, in Wada, E. (ed.), *Proc. Japanese Logic Programming Conference*, pp. 209–216, ICOT, Tokyo, 1986. Proceedings also published as Springe r *LNCS* **264**.

[134] Nilsson, M., and Tanaka, H., The art of building a parallel logic programming system or from zero to full GHC in ten pages, in Wada, E. (ed.), *Proc. Japanese Logic Programming*

Conference, pp. 155–163, ICOT, Tokyo, 1987. Proceedings also to a ppear as Springer *LNCS*.

[135] Nilsson, M., and Tanaka, H., Massively parallel implementation of Flat GHC on the connection machine, *Proc. International Conference on Fifth Generation Computer Systems*, pp. 1031–1040, ICOT, Tokyo, 1988.

[136] Nilsson, M., and Tanaka, H., A Flat GHC implementation for supercomputers, in Bowen, K. and Kowalski, R.A. (eds.), *Proc. 5th International Conference Symposium on Logic Programming*, pp. 1337–1350, MIT Press, 1988.

[137] Ohki, M., et al., An object-oriented programming language based on a parallel logic programming language KL1, ICOT Technical Report TR-222, Institute for New Generation Computer Technology, Tokyo, 1987.

[138] Okabe, Y., and Yajima, S., Parallel computational complexity of logic programs and alternating turing machines, *Proc. International Conference on Fifth Generation Computer Systems*, pp. 356–363, ICOT, Tokyo, 1988.

[139] Okumura, A., and Matsumoto, Y., Parallel programming with layered streams, *Proc. IEEE Symposium on Logic Programming*, pp. 224–231, San Francisco, 1987.

[140] Pereira, L.M., and Nasr, R., Delta-Prolog: a distributed logic programming language, *Proc. International Conference on Fifth Generation Computer Systems*, pp. 283–291, ICOT, Tokyo, 1984.

[141] Picca, R., Bellone, J., and Levy, J., Or-parallel And-interleaving execution of Concurrent Prolog, Technical Report CS87-07, Department of Computer Science, The Weizmann Institute of Science, Rehovot, 1987.

[142] Pnueli, A., Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends, in de Bakker, J.W., de Roever, W.P., and Rozenberg, G. (eds.), *Current Trends in Concurrency, Overviews and Tutorials* , *LNCS* **224**, pp. 510–584, Springer-Verlag, 1986.

[143] Ramakrishnan, R., and Silberschatz, A., Annotations for distributed programming in logic, *Conference Record 13th ACM Symposium on Principles of Programming Languages*, pp. 255–262, 1986.

[144] Reches, E., Gudes, E., and Shapiro, E., Parallel access to a distributed database and its implementation in Flat Concurrent Prolog, Technical Report CS88-11, Department of Computer Science, The Weizmann Institute of Science, Rehovot, 1988.

[145] Ringwood, G.A., PARLOG86 and the dining logicians, *CACM* 31(1), pp. 10–25, 1988.

[146] Ringwood, G.A., Pattern-directed, Markovian, linear, guarded definite clause resolution, Department of Computing, Imperial College, London.

[147] Robinson, J.A., A machine oriented logic based on the resolution principle, *J. ACM* **12**(1), pp. 23–41, 1965.

[148] Safra, S., *Partial Evaluation of Concurrent Prolog and Its Implications*, M.Sc. Thesis, Technical Report CS86-24, Department of Computer Science, The Weizmann Institute of Science, Rehovot, 1986.

[149] Safra, S., and Shapiro, E., Meta-interpreters for real, *Information Processing 86*, pp. 271–278, North-Holland, 1986. Also Chapter 25 in [164].

[150] Saraswat, V.A., Partial correctness semantics for CP[↓,|,&], *Proc. 5th Conference on Foundations of Software Technology and Theoretical Computer Science*, *LNCS* **206**, pp. 347–368, New Delhi, 1985.

[151] Saraswat, V.A., Problems with Concurrent Prolog, Technical Report 86-100, Carnegie-Mellon University, 1986.

[152] Saraswat, V.A., Merging many streams efficiently: The importance of atomic commitment, Chapter 16 in [164].

[153] Saraswat, V.A., A compiler of CP(↓,|,&) on top of Prolog, Technical Report CS-87-174, Carnegie-Mellon University, 1987.

[154] Saraswat, V.A., The concurrent logic programming language CP: Definition and operational

semantics, *Proc. ACM Symposium on Principles of Programming Languages*, pp. 49–63, 1987.

[155] Saraswat, V.A., The language GHC: Operational semantics, problems and relationship with CP[↓,|], *Proc. IEEE Symposium on Logic Programming*, pp. 347–358,San Francisco, 1987.

[156] Saraswat, V.A., A somewhat logical formulation of CLP synchronisation primitives, in Bowen, K. and Kowalski, R.A. (eds.), *Proc. 5th International Conference Symposium on Logic Programming*, pp. 1298–1314, MIT Press, 1988.

[157] Saraswat, V.A., *Concurrent Constraint Programming Languages*, Ph.D. Thesis, Carnegie-Mellon University, 1989.

[158] Saraswat, V.A., Weinbaum, D., Kahn, K., and Shapiro, E., Detecting stable properties of networks in concurrent logic programming languages, *Proc. ACM Conference on Principles of Distributed Computing*, 1988.

[159] Shapiro, E., *Algorithmic Program Debugging*, MIT Press, 1983.

[160] Shapiro, E., A subset of Concurrent Prolog and its interpreter, ICOT Technical Report TR-003, Institute for New Generation Computer Technology, Tokyo, 1983. Also chapter 2 in [164].

[161] Shapiro, E., Alternation and the computational complexity of logic programs, *J. Logic Programming* 1(1), pp. 19–33, 1984.

[162] Shapiro, E., Concurrent Prolog: A progress report, *IEEE Computer* 19(8), 44–58, 1986. Also Chapter 5 in [164].

[163] Shapiro, E., Systolic programming: A paradigm of parallel processing, *Proc. International Conference on Fifth Generation Computer Systems*, pp. 458–471, 1984. Also Chapter 7 in [164].

[164] Shapiro, E. (Editor), *Concurrent Prolog: Collected Papers*, Vols. 1 and 2, MIT Press, 1987.

[165] Shapiro, E., Or-parallel Prolog in Flat Concurrent Prolog, Chapter 34 in [164].

[166] Shapiro, E., Embedding Linda and other joys of concurrent logic programming, Technical Report, Department of Computer Science, The Weizmann Institute of Science, Rehovot, 1989.

[167] Shapiro, E., and Mierowsky, C., Fair, biased, and self-balancing merge operators: Their specification and implementation in Concurrent Prolog, *New Generation Computing* 2(3), pp. 221–240, 1984. Also Chapter 14 in [164].

[168] Shapiro, E., and Safra, S., Multiway merge with constant delay in Concurrent Prolog, *New Generation Computing* 4(2), pp. 211–216, 1986. Also Chapter 15 in [164].

[169] Shapiro, E., and Takeuchi, A., Object-oriented programming in Concurrent Prolog, *New Generation Computing* 1(1), pp. 25–49, 1983. Also Chapter 29 in [164].

[170] Silverman, W., Hirsch, M., Houri, A., and Shapiro, E., The Logix system user manual, Version 1.21, Chapter 21 in [164].

[171] Sterling, L.S., and Shapiro, E., *The Art of Prolog*, MIT Press, 1986.

[172] Suzuki, N., Experience with specification and verification of complex computer using Concurrent Prolog, in Warren, D.H.D, and van Caneghem, M. (eds.), *Logic Programming and Its Applications*, pp. 188–209, Ablex Pub. Co., New Jersey, 1986.

[173] Szöke, D., *Distributed Flat Concurrent Prolog on a Network Architecture*, M.Sc. Thesis, Department of Computer Science, The Weizmann Institute of Science, Rehovot, 1987.

[174] Takeda, Y., Nakashima, H., Masuda, K., Chikayama, T., and Taki, K., A load balancing mechanism for large scale multiprocessor systems and its implementation, *Proc. International Conference on Fifth Generation Computer Systems*, pp. 978–986, IC OT, Tokyo, 1988.

[175] Takeuchi, A., How to solve it in Concurrent Prolog, 1983 (unpublished note).

[176] Takeuchi, A., Algorithmic debugging of GHC programs and its implementation in GHC, Chapter 26 in [164].

[177] Takeuchi, A., and Furukawa, K., Bounded-buffer communication in Concurrent Prolog, *New Generation Computing* 3(2), pp. 145–155, 1985. Also Chapter 18 in [164].

[178] Takeuchi, A., and Furukawa, K., Partial evaluation of Prolog programs and its application to meta-programming, *Information Processing 86*, pp. 415–420, North-Holland, 1986.

[179] Takeuchi, A., et al., A description language with AND/OR parallelism for concurrent systems and its stream-based realization, ICOT Technical Report TR-229, Institute for New Generation Computer Technology, Tokyo, 1987.

[180] Tamaki, H., A distributed unification scheme for systolic programs, *Proc. International Conference on Parallel Processing*, pp. 552–559, 1985.

[181] Tanaka, J., A simple programming system written in GHC and its reflective operations, *Proc. Japanese Logic Programming Conference*, pp. 143–149, ICOT, Tokyo, 1988.

[182] Tanaka, J., Meta-interpreters and reflective operations in GHC, *Proc. International Conference on Fifth Generation Computer Systems*, pp. 774–783, ICOT, Tokyo, 1988.

[183] Tanaka, J., Ueda, K., Miyazaki, T., Takeuchi, A., Matsumoto, Y., and Furukawa, K., Guarded Horn clauses and experiences with parallel logic programming, *Proc. FJCC ACM*, pp. 948–954, Dallas, 1986.

[184] Tanaka, H., et al., First annual report of the research on a large scale knowledge information processing system, ICOT Technical Report, 1988 (Partly in Japanese).

[185] Taylor, H., Localizing the GHC suspension test, in Bowen, K. and Kowalski, R.A. (eds.), *Proc. 5th International Conference Symposium on Logic Programming*, pp. 1257–1271, MIT Press, 1988.

[186] Taylor, S., *Parallel Logic Programming Techniques*, Ph.D. Thesis, Department of Computer Science, The Weizmann Institute of Science, Rehovot, 1988.

[187] Taylor, S., Av-Ron, E., and Shapiro, E., A layered method for process and code mapping, *J. New Generation Computing* 5(2), 1987. Also Chapter 22 in [164].

[188] Taylor, S., and Foster, I., *Strand Language Reference Manual*, Technical Report PAR 88/10, Department of Computing, Imperial College, London, 1988.

[189] Taylor, S., Hellerstein, L., Safra, S., and Shapiro, E., Notes on the complexity of systolic programs, *J. Parallel and Distributed Computing* 4(3), 1987. Also Chapter 8 in [164].

[190] Taylor, S., Safra, S., and Shapiro E., A parallel implementation of Flat Concurrent Prolog, *J. Parallel Programming* 15(3), pp. 245–275, 1987. Also Chapter 39 in [164].

[191] Taylor, S., Shapiro, R., and Shapiro, E., FCP: A summary of performance results, in Fox, G. (ed.), *Proc. 3rd Conference on Hypercube Concurrent Computers and Applications*, pp. 1364–1373, ACM Press, 1988.

[192] Taylor, S., and Shapiro, E., An improved parallel algorithm for Flat Concurrent Prolog, Technical Report CS88-09, Department of Computer Science, The Weizmann Institute of Science, Rehovot, 1988.

[193] Tick, E., A performance comparison of And- and Or-parallel logic programming architectures, ICOT Technical Report TR-421, Institute for New Generation Computer Technology, Tokyo, 1988.

[194] Tribble, E.D., Miller, M.S., Kahn, K., Bobrow, D.G. and Abbott, C., Channels: A generalization of streams, in Lassez, J.-L. (ed.), *Proc. 4th International Conference of Logic Programming*, pp. 839–857, MIT Press, 1987. Also Chapter 17 in [164].

[195] Uchida, S., Taki, K., Nakashima, K., Goto, A., and Chikayama, T., Research and development of the parallel inference system in the intermediate stage of the FGCS project, *Proc. International Conference on Fifth Generation Computer Systems*, pp. 16–36, ICOT, Tokyo, 1988.

[196] Ueda, K., Concurrent Prolog re-examined, ICOT Technical Report TR-102, Institute for New Generation Computer Technology, Tokyo, 1985.

[197] Ueda, K., *Guarded Horn Clauses*, Ph.D. Thesis, Information Engineering Course, University of Tokyo, Tokyo, 1986.

[198] Ueda, K., Guarded Horn Clauses, in Wada, E. (ed.), *Logic Programming*, *LNCS* 221, pp. 168–179, Springer-Verlag, 1986. Also Chapter 4 in [164].

[199] Ueda, K., Guarded Horn Clauses: A parallel logic programming language with the concept of a guard, ICOT Technical Report TR-208, Institute for New Generation Computer Technology, Tokyo, 1986 (revised in 1987). Also in Nivat, M., and Fuchi, K. (eds.), Prog ramming of Future Generation Computers, pp. 441–456, North-Holland, 1988.

[200] Ueda, K., Making exhaustive search programs deterministic, *New Generation Computing* 5(1), pp. 29–44, 1987.

[201] Ueda, K., Making exhaustive search programs deterministic, Part II, ICOT Technical Report TR-249, Institute for New Generation Computer Technology, Tokyo, 1987.

[202] Ueda, K., Parallelism in logic programming, *Proc. IFIP Congress*, 1989.

[203] Ueda, K., and Chikayama, T., Concurrent Prolog compiler on top of Prolog, *Proc. IEEE Symposium on Logic Programming*, pp. 119–126, 1985.

[204] Ueda, K., and Furukawa, K., Transformation rules for GHC programs, *Proc. International Conference on Fifth Generation Computer Systems*, pp. 582–591, ICOT, Tokyo, 1988.

[205] Vasey, P., Qualified answers and their application to transformation, in Shapiro, E. (ed.), *Proc. 3^{rd} International Conference on Logic Programming, LNCS* **225**, pp. 425–432, Springer-Verlag, 1986.

[206] Warren, D.H.D., An abstract Prolog instruction set, Technical Report 309, Artificial Intelligence Center, SRI International, 1983.

[207] Warren, D.H.D., The SRI model for Or-parallel execution of Prolog — abstract design and implementation, *Proc. IEEE Symposium on Logic Programming*, pp. 92–102, San Francisco, 1987.

[208] Weinbaum, D., and Shapiro, E., Hardware description and simulation using Concurrent Prolog, *Proc. CHDL '87*, pp. 9–27, Elsevier Science Publishing, 1987. Also Chapter 36 in [164].

[209] Yang, R., *A Parallel Logic Programming Language and Its Implementation*, Ph.D. Thesis, Keio University, 1986.

[210] Yang, R., and Aiso, H., P-Prolog: a parallel logic language based on exclusive relation, in Shapiro, E. (ed.), *Proc. 3^{rd} International Conference on Logic Programming, LNCS* **225**, pp. 255–269, Springer-Verlag, 1986.

[211] Yardeni, E., and Shapiro, E., A type system for logic programs, Chapter 28 in [164].