PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988



# The Language FCP(:,?)

Shmuel Kliger[1] Eyal Yardeni[1] Kenneth Kahn[2] Ehud Shapiro[1]

(Extended Abstract)

[1]The Weizmann Institute of Science [2]Xerox Palo Alto Research Center

## Abstract

The language FCP(:,?) is the outcome of attempts to integrate the best of several flat concurrent logic programming languages, including Flat GHC, FCP(|,↓) and Flat Concurrent Prolog, in a single consistent framework. The result is a language which is strong enough to accomodate all useful concurrent logic programming techniques, including those which rely on atomic test unification and read-only variables, yet incorporates the weaker languages mentioned as subsets. This allows the programmer to remain within a simple subset of the language such as Flat GHC when the full power of atomic unification or read-only variables is not needed.

## 1 Introduction

Several concurrent logic programming languages were defined and implemented in the past seven years, since the Relational Language was proposed by Clark and Gregory [Cla81]. Recently, most attention has focused on the so called flat concurrent logic programming languages, since their simplicity and ease of implementation, compared with their non-flat ancestors, comes at a relatively low cost in expressive power. On one end of the spectrum of flat concurrent logic programming languages there are Flat GHC [Ued85,Kim87] and Flat PARLOG [Fos87] with non-atomic unification. On the other end of the spectrum is Flat Concurrent Prolog [Mie85](henceforth called FCP(?)), with atomic read-only test unification. An intermediate language is FCP(|,↓) [Sar86], with atomic test unification.

The advantages of the weaker languages are simplicity and ease of implementation. However, their expressive power is deficient in several important respects. In particular, necessary functions of computation control cannot be expressed in languages such as Flat GHC and Flat PARLOG, since these languages cannot reflect on failure. Therefore, to make them practical, these languages were extended with a computation control primitive called a control meta-call [Cla84]. In addition, sophisticated uses of *atomic variables* [Sha88], atomic unification, atomic test unification, and read-only variables, are not available in these weaker languages. Most notable are the short-circuit technique for termination and quiescence detection [Tak83,Sar88b], which, in its full generality, requires atomic variables, the duplex stream protocol and multiple writer streams, which require atomic test unification, and anonymous mutual exclusion and protected data-structures, which require read-only variables. In addition, several meta-level functions, such as live and frozen snapshots [Saf86,Sar88b], and the recording of traces for computation replay and debugging [Lic87] cannot be easily incorporated in the proposed meta-call constructs.

The advantage of the stronger languages is clear: added expressive power. The disadvantages are of two sorts: methodological problems and performance problems. We first consider the methodological problem.

We call a unification which may affect the environment *tell unification*, as opposed to *ask unification* (input unification) which cannot [Sar88a]. Both FCP(|,↓) and FCP(?) have tell unification as the default in head/goal unification, and require special annotations to specify ask unification. In FCP(|,↓) this annotation is attached to the head, while in FCP(?) it is usually attached to the goal. Additionally, FCP(?) can have read-only annotations in the head, which enhances the functionality. This functionality has no correspondence in the other languages.

This default rule seemed useful at first, since it is closer to the original spirit of logic programming and to the practice of Prolog. Experience, however, showed otherwise. Most unifications in concurrent logic programs are ask unifications. If the default is tell unification, and one has to go to special efforts to specify ask unification, then programmers tend to write programs with unintended behavior which, under certain circumstances, produce unexpected bugs. A common question a frustrated FCP(?) programmer finds himself asking is: "who the hell has instantiated this variable?" If, on the other hand, the program mistakenly specifies ask unification instead of tell unification, then bug detection is usually easy. The computation

typically deadlocks, and, by inspecting the deadlocked resolvent, the programmer may easily find the faulty process. Hence we conclude that the default rule in Flat GHC, in which the head specifies ask unification, is preferred over that of FCP(?) and FCP(|,↓).

Although both FCP(?) and FCP(|,↓) share the default of tell unification, they have a different method of specifying ask unification. FCP(|,↓) specifies ask unification statically, by annotating terms in the head as input. FCP(?), on the other hand, specifies ask unification dynamically, by annotating arguments in the call. In comparing the two, it is clear that in most cases static specification of input, as is done in all other languages except FCP(?), is preferred over dynamic specification. The former is clearer, more modular, and easier to implement efficiently. Ask unification can be compiled more efficiently than tell unification using the decision tree compilation technique [Kli88a]. Without global analysis, which infers that the caller always places read-only variables in the appropriate position [Tay88,Gal88], an FCP(?) program would compile less efficiently than a corresponding program in a language with ask unification. On the other hand, there are special applications and programming techniques which enjoy the full power of read-only unification, which is inherently dynamic.

Concerning the performance problem, it has been claimed that a language with atomic unification requires a much more complex runtime execution mechanism compared to a language with non-atomic unification. We think this is not the case. First, to make Flat GHC and PARLOG practical, a control meta-call has been added to these languages. The distributed implementation of this construct requires distributed termination detection, which greatly reduces the freedom of the implementation, and hence reduces the gap between non-atomic and atomic unification. Secondly, performance analysis of the distributed unification algorithm developed for the execution of languages with atomic test unification shows that in the common special cases of unification, such as single-producer single-consumer stream communication and incomplete messages, the overhead of the mechanisms required to ensure the atomicity of unification is negligible compared to the other costs [Tay85,Tay87]. There are no comparable studies of languages without atomic unification. However, the complexity of the algorithms developed at ICOT for the correct distributed implementation of the control meta-call in KL1 (Flat GHC with meta-call and atomic variables) and in Flat PARLOG [Fos88] suggests that the difference in performance between the two approaches would not be significant. More specifically, it seems that a program which does not require atomic unification would exhibit a similar message passing performance whether executed as, say, an FCP(?) program or as a Flat GHC program (assuming atomic variables [Sha88]), using the corresponding distributed unification algorithms.

Therefore we attempt to define a language which will satisfy the following requirements:

- The language should be as expressive as FCP(?), preserving the atomic unification property of the language, tell unification before commit and the read-only protection mechanism. These features enable programming techniques, like the short-circuit, protected data structures and test-and-set, which cannot be naturally implemented in the other flat languages which lack these features.
- The language should encourage a convenient and efficient programming style, as the other flat languages do.
- The language should have a clean and easily implementable operational semantics.

In this paper we introduce two languages, FCP(:) and FCP(:,?), addressing the above requirements. The basic concept of the FCP(:) language was suggested by Saraswat, and incoporated in the language cc(↓,|) [Sar88a]. The main idea in the language cc(↓,|), which is a descendant of the language FCP(|,↓), is to separate the guard part to an *ask* part, which only tests the environment, and a *tell* part, which can also attempt to affect the environment. A clause in this language has the general form:

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

For a clause to commit, both its *ask* and *tell* parts should succeed. Saraswat's motivation in defining cc(↓,|) is quite different from ours, namely to develop a concurrent constraint programming language, whose constraints can be generalized beyond the equality constraints used in concurrent logic programming. Nevertheless, we find the proposal to separate the guard of a guarded clause in this way suitable for our purposes as well, which is to define a concrete concurrent logic programming language that naturally generalizes languages without atomic unification, and can incorporate read-only unification.

The language FCP(:) attempts to solve the first problem, namely, that of the default unification being ask, while maintaining the power of atomic test unification. FCP(:) differs from cc(↓,|) in three main aspects: the guard predicates of the language, the syntactic restriction on FCP(:) clauses and the fairness requirements.

There are some well known paradigms of distributed programming that are realizable in concurrent logic programming by using *read-only variables*. Read-only variables can be used to achieve various forms of dynamic synchronization not achievable in the weaker languages mentioned. The paradigms are: test-and-set, anonymous mutual-exclusion, multiple-writer

stream, distributed queues and protected data structures.

The remainder of this paper is organized as follows. In Section 2 we describe a unified framework for defining the syntax and the abstract operational semantics for the flat concurrent logic programming (FCLP) languages. Using this framework we define the language FCP(:) in Section 3 and the language FCP(:,?) in Section 4. Finally in Section 5 we give a detailed and formal definition of the guard predicates of FCP(:,?). A fully abstract denotational semantics for FCP(?) can be found in [Ger88].

We assume the reader is acquainted with the concepts of logic programming (see [Llo87]). For completeness, definitions of some of these concepts are given in appendix A.

# 2 Flat Concurrent Logic Programming Languages

In this section we define the Flat Concurrent Logic Programming (FCLP) family of languages.

## 2.1 The FCLP Syntax

A flat concurrent logic program is a set of guarded Horn clauses of the form:

$$H \leftarrow G_1, \ldots, G_n \mid B_1, \ldots, B_m \quad m, n \geq 0.$$

where $H$ is called the *head* of the clause, $G_1, \ldots, G_n$ the *guard* and $B_1, \ldots, B_m$ the *body*. The *guard* consists of predicates from a pre-defined set of predicates called the *guard predicates*. In case $n = 0$, the commitment operator ('|') can be omitted and the guard is empty, i.e. the guard is true.

The differences between the syntax of the different flat concurrent programming languages lie in:

- The structure of the guard and the guard predicates themselves.
- The synchronization constructs.
- Syntactic restrictions.

## 2.2 FCLP Operational Semantics

The differences between the operational semantics of each of the FCLP languages are captured in two components:

- The definitions of the guard predicates.
- The effect of trying to reduce a goal with a clause.

**Definition:** Let $V$ be the countable set of the variables in a first order language $L$ and $\Sigma^L$ its signature; $\Sigma^L = \Sigma^L_p \bigcup \Sigma^L_f$, where $\Sigma^L_p$ is the set of predicates and $\Sigma^L_f$ the set of constants and functions symbols. Also let $P$ be a program in $L$ and $\Sigma^P$ its signature. Then

- $\mathcal{GS} = At(V, \Sigma^L)$ is the set of all atoms over $V$ and $\Sigma^L$.
- $CS$ is the set of clauses in $L$.
- $SS$ is the set of substitutions over $V$ and $\Sigma^L_f$.

Given a flat concurrent logic programming language $L$, we define the semantics of a program in $L$ using a clause-try function, or *try function* for short. A try function generalizes the notion of goal-head unification of logic programs. We define:

### The Try Function

**Definition:** Let $\mathcal{TRY} = 2^{SS} \bigcup \{\{fail\}, \{suspend\}\}$. A try function, $try_L$, for a language $L$ has the type:

$$try_L : \mathcal{GS} \times CS \longrightarrow \mathcal{TRY}.$$

Let $G \in \mathcal{GS}$, $C \in CS$ then $try_L(G, C)$ is a set of substitutions, the set $\{fail\}$ or the set $\{suspend\}$.

The try function must satisfy four properties:

(a) Try substitutions are equal up to renaming:

$$\forall \theta, \theta' \in try_L(G, C).\theta \approx \theta'.$$

(b) Suspension is not stable:

$$suspend \in try_L(G, C) \Longrightarrow \exists \theta.suspend \notin try_L(G\theta, C).$$

(c) Failure is stable:

$$fail \in try_L(G, C) \Longrightarrow \forall \theta.fail \in try_L(G\theta, C).$$

(d) Try substitution is monotonic:

$(\theta \in try_L(G, C) \wedge \theta' \in try_L(G', C) \wedge G'$ is an instance of $G) \Longrightarrow G'\theta'$ is an instance of $G\theta$

**Notes:**

1. Condition (b) implies that a suspended clause try may be resumed in the future, if appropriate additional bindings are provided by the environment.
2. Condition (c) implies that a failed clause need not be tried again.

**Definition:** If the try function of a language $L$ satisfies the following property (e), then $L$ is called *monotonic*:

(e) Success is monotonic:

$$\theta \in try_L(G, C) \Longrightarrow \forall \theta' \exists \theta''.\theta'' \in try_L(G\theta\theta', C)$$

### Transition System

Given a try function $try_L$ for a flat concurrent logic programming language $L$, we associate with every program $P$ of $L$ a *transition system* $\Pi^P_L = \langle S, T \rangle$ that consists of a set of *states* $S$ and a set of *transitions* $T$.

**The set of states $S$:**

Let $\mathcal{R} = At(V, \Sigma^P)^*$ be the set of sequences of goals (resolvents) over $V$ and $\Sigma^P$, and $\Theta \subseteq SS$ the set of substitutions over $V$ and $\Sigma^P_f$. Then

$$S = \{\langle R;\theta\rangle \mid R \in \mathcal{R} \cup \{tt, ff, dl\} \text{ and } \theta \in \Theta\}.$$

**The set of transitions $T$:**
There are four types of transitions; which are functions from $S$ to $2^S$. We use $s \xrightarrow{\tau} s'$ to denote $s' \in \tau(s)$.

1. **Reduce$_L$.** $\langle A_1,\ldots,A_i,\ldots,A_n;\theta\rangle \xrightarrow{Reduce_L} \langle (A_1,\ldots,B_1,\ldots,B_k,\ldots,A_n)\theta';\theta\circ\theta'\rangle$
   if $\theta' \in try_L(A_i, C)$ for some renamed clause $C$ of P with body $B_1,\ldots,B_k$ that does not share variables with $\langle A_1,\ldots,A_i,\ldots,A_n;\theta\rangle$.

2. **Fail$_L$.** $\langle A_1,\ldots,A_i,\ldots,A_n;\theta\rangle \xrightarrow{Fail_L} \langle ff;\theta\rangle$
   if $fail \in try_L(A_i, C)$ for every renaming of every clause $C$ in $P$.

3. **Halt$_L$.** $\langle true,\ldots,true;\theta\rangle \xrightarrow{Halt_L} \langle tt;\theta\rangle$

4. **Deadlock$_L$.** $\langle A_1,\ldots,A_n;\theta\rangle \xrightarrow{Deadlock_L} \langle dl;\theta\rangle$
   if $Reduce_L$, $Fail_L$, and $Halt_L$ do not apply.

**Definition:** Given a state $s \in S$, a state $s' \in S$ *follows* $s$ if $\exists\tau \in T$ such that $s \xrightarrow{\tau} s'$.

**Computation**

**Definition:** A transition $\tau$ is *enabled* on a state $s$, if $\{s' \mid s \xrightarrow{\tau} s'\} \neq \emptyset$.

**Definition:** A state $s$ is *terminal* if no transition is enabled on $s$.

By definition a terminal state of the form $\langle tt;\theta\rangle$ is called a *success state*, $\langle ff;\theta\rangle$ is called a *failure state* and $\langle dl;\theta\rangle$ is called a *deadlock state*.

**Definition:** A *computation* $c$ is a (finite or infinite) sequence of states: $s_0, s_1, s_2, \ldots \quad s_i \in S$ satisfying:

**Initiation** – Let $\epsilon$ denote the empty substitution. Then $s_0 \in \{\langle R, \epsilon\rangle \mid R \in \mathcal{R}\}$.

**Consecution** – $\forall i = 0,1,2,\ldots \ \exists\tau \in T$, such that $s_i \xrightarrow{\tau} s_{i+1}$.

**Termination** – $c$ is finite and of length $k$ if and only if $s_k$ is a terminal state.

Any prefix of a computation is called a *partial computation*.

# 3 The Language FCP(:)

We introduce here two new languages, FCP(:) and FCP(:,?), using the above framework. A program in these languages can perform tell unification as part of the test for the clause selection. If the test unification fails, it should leave no trace of its attempted execution; in other words, test unification should be atomic.

## 3.1 Syntax

An *ask-tell clause* is a guarded Horn clause of the form:
$H :- A_1,\ldots,A_n : T_1,\ldots,T_m \mid B_1,\ldots,B_k$
$m, n, k \geq 0$, where $H$ is called the *head* of the clause, $A_1,\ldots,A_n : T_1,\ldots,T_m$ the *guard* and $B_1,\ldots,B_k$ the *body*. Each $A_i$ is an *ask guard* and $A_1,\ldots,A_n$ is called the *ask part*, while each $T_i$ is a *tell guard* and $T_1,\ldots,T_m$ is called the *tell part*.

In case $m = 0$, the ':' is omitted and the guard has only an ask part. In case $n = m = 0$, the commitment operator is also omitted and the guard is empty, i.e. the guard is true.

The *guard* consists of predicates from a pre-defined set of predicates called the *guard predicates*. The guard predicates are divided into *ask predicates* and *tell predicates*. The ask and tell predicates are defined in Section 5 of this paper.

The only function symbols in the language are tuples $\langle \_,\_,\ldots,\_\rangle$ with arity greater than or equal to one. The constants are strings, integers, reals and nil. Sometimes we denote the term $\langle f, t_1, t_2, \ldots, t_n\rangle$ by the term $f(t_1, t_2, \ldots, t_n)$, and the term $\langle X, Y\rangle$ by the term $[X|Y]$.

**Definition:** A *guard goal* is a goal whose predicate is taken from the pre-defined set of predicates (the guard predicates), and an *ask (tell) goal* is a goal whose predicate is taken from the pre-defined set of predicates (the ask (tell) predicates). The set of ask goals over $V$ and $\Sigma_f$ is denoted by $\mathcal{ASK}$ and the set of tell goals over $V$ and $\Sigma_f$ is denoted by $\mathcal{TELL}$.

The set of legal clauses of FCP(:) is defined in appendix B. The syntax restrictions on this language ensures that no new term is generated during the execution of the head unification and the ask part. It also enables a static analysis to determine the order of execution of the ask part's guards in an optimized compilation. By applying the static analysis on a clause, a transformation can be done in which the unification goals are folded into the head. However, this transformation can be done only on unification goals, and not on other guard goals.

## 3.2 Operational Semantics

The operational semantics of FCP(:) is defined by its try function using the framework described above for the FCLP operational semantics.

A clause try in FCP(:) can be decomposed into two steps: the *validation step*, in which the arguments of the goal are tested without changing the goal, and the *satisfaction step*, in which the tell part is satisfied with possible effects on the goal [Sar88a]. If the validation step needs more information in order to succeed then the clause try suspends.

**The Try Function**

First, we have to define the semantics of the guard predicates (the ask and tell predicates). The semantics of the guard predicates is given via an evaluation function.

**Definition:** The *evaluation function*,

$$\pi : ASK \longrightarrow \{true, false, suspend\}$$

is a function, which for every guard goal $G \in ASK$, $\pi(G)$ can return *true*, *false* or *suspend*. The function can be extended to operate on a set of guard goals. The *evaluation function* of a set of ask goals $Gs$ is defined by

$$\pi(Gs) = \begin{cases} true & \text{if } \forall G \in Gs,\ \pi(G) = true \\ false & \text{if } \exists G \in Gs \text{ s.t. } \pi(G) = false \\ suspend & \text{otherwise} \end{cases}$$

The definition of the evaluation function for each ask guard of FCP(:) is given in the full paper. Some special examples are given Section 5.

**Notations:** Given a clause $C$ and a substitution $\theta$, $Ask\theta$ ($Tell\theta$) denotes the set of ask (tell) goals resulting from applying the substitution $\theta$ on the set of goals in $Ask$ ($Tell$).

The arguments of each non-unification tell guard are statically divided into input and output arguments. The guard computes a partial function from the input arguments. The value of the function is unified with the corresponding output arguments. If the function is defined on some input, the result of the tell guard is defined to be the result of the unification implied by the guard. In order to make the functions of the tell part total we restrict their domain by adding preconditions to the ask part. Each tell guard may have preconditions that appear in the ask part to ensure that it is well defined. For example, the tell guard $make\_tuple(1+2, X)$ creates the tuple $\langle \_, \_, \_ \rangle$ and unifies it with $X$. The precondition is that 1+2 is a positive integer valued arithmetic expression.

**Definition:** Let $G$ be a tell guard of the form $G = f(X_1, \ldots, X_k, Y_1, \ldots, Y_l)$, where $X_i$'s are the input arguments, $Y_i$'s are the output arguments and $f : Terms^k \rightarrow Terms^l$ is the function computed by $G$. The *unification implied* by $G$ is the unification

$$\langle Y_1, \ldots, Y_l \rangle = f(X_1, \ldots, X_k)$$

For most of the tell guards $l = 1$. In the example above $X$ is the output argument and the implied unification is $X = \langle \_, \_, \_ \rangle$.

**Definition:** Let:

1. $Tell = U_1, \ldots, U_n, O_1, \ldots, O_m$ where:
   - $\forall i, 1 \leq i \leq n, U_i = (T_{i1} = T_{i2})$
   - $\forall i, 1 \leq i \leq m, O_i \neq (T_1 = T_2)$ with implied unification

     $$\langle Y_{i1}, \ldots, Y_{il_i} \rangle = f_i(X_{i1}, \ldots, X_{ik_i})$$

2. $T = \langle T_{11}, \ldots, T_{n1}, Y_1, \ldots, Y_m \rangle$
3. $T' = \langle T_{12}, \ldots, T_{n2}, f_1, \ldots, f_m \rangle$
   where $Y_i$ is a shorthand of $\langle Y_{i1}, \ldots, Y_{il_i} \rangle$ and $f_i$ is a shorthand of $f_i(X_{i1}, \ldots, X_{ik_i})$.

then

$$mgu(Tell) \stackrel{\text{def}}{=} mgu(T, T').$$

**Definitions:** Given a goal $G$ and an FCP(:) clause $Head \leftarrow Ask : Tell \mid Body$.

- Let $\Psi = (\exists\theta.(Head\theta = G) \wedge \pi(Ask\theta) = true)$.
  $\Psi$ is called the *validation condition* of the goal $G$ and the clause $C$.
- Let $\Phi = (\exists\theta.(Head\theta = G\theta) \wedge \pi(Ask\theta) = true)$.

**Definition:** The *try function* for FCP(:) is a function

$$try_{FCP(:)} : \mathcal{GS} \times CS \longrightarrow TRY.$$

such that:

$try_{FCP(:)}(G, C)$

$$\ni \begin{cases} \theta \circ \sigma & \text{if } \Psi \wedge \sigma \in mgu(Tell\theta) \\ fail & \text{if } \neg\Phi \vee (\Psi \wedge fail \in mgu(Tell\theta)) \\ suspend & \text{otherwise} \end{cases}$$

where $\theta$ is a most general substitution that satisfies $\Psi$. Note that $suspend \in try_{FCP(:)}(G, C)$ if $\Phi \wedge \neg\Psi$.

The try function is well defined. It can be shown that the conditions are pairwise disjoint and their union covers all the possible goal and clause relations.

### 3.3 Examples

In this section we present examples of techniques in FCP(:) that require non-empty tell parts and therefore cannot be written directly in a language like Flat GHC.

**Mutual exclusion**

Let $p_1, \ldots, p_n$ be 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 bound to the variable *ME*. Each process $p_k$ competing for a lock attempts nondeterministically to bind *ME* to its identifier $I_k$, or to check that *ME* is already bound to some $I \neq I_k$.

The skeleton of each process is as follows, assuming its first argument is bound to *ME* and second to its unique identifier.

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

Single-round mutual exclusion can be achieved in Flat GHC using a stream merger [Sha88].

**The duplex stream protocol [Sar87]**

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 consumed. This can be achieved using the following duplex stream protocol. The producer places a message $M$ on the stream wrapped as *write(M)*. The consumer, when reaching the uninstantiated tail 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 consumed. 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 already been 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 will not place a *read* message on an initially empty stream.

# 4 The Language FCP(:,?)

FCP(:,?) extends FCP(:) with the notion of read-only variables. We assume, in addition to the set of standard (henceforth called *writable*) variables, an additional isomorphic set of *read-only* variables. The read-only operator ? is an isomorphic mapping from writable variables to read-only variables. For any writable variable $X$, $X?$ is called the read-only variable corresponding to $X$. The read-only operator is extended to terms, where it is the identify function. A read-only variable $X?$ is a variable that can be asked but cannot be told a value. It receives a value $T?$ if and only if its corresponding writable variable $X$ receives the value $T$ (or $T?$).

Trying to tell a value to a read-only variable causes a suspension. The read-only annotation is only a top level protection, i.e. if $X?$ is instantiated to a term $T$ then the sub-terms of $T$ are not protected unless they are explicitly read-only annotated.

**Definition:** A substitution, $\theta$, is *admissible* if $X?\theta = X?$ for every variable $X$. Let *ADMS* denote all admissible substitutions.

**Definition:** The *read-only extension*, $\theta_?$, of an admissible substitution $\theta$, is the most general substitution satisfying:

1. $\theta_?$ is an instance of $\theta$, i.e. $X\theta_?$ is an instance of $X\theta$ for every $X$.
2. $\theta_?$ is idempotent, i.e. $\theta_? \circ \theta_? = \theta_?$.
3. $X?\theta_? = (X\theta_?)?$ for every writable variable $X$.

**Definition:** The *read-only mgu*, $mgu_?$, of two terms $T_1$ and $T_2$ is defined by:

$$mgu_?(T_1,T_2) \ni \begin{cases} \theta_? & \text{if } \theta \in mgu(T_1,T_2) \bigcap ADMS \\ fail & \text{if } T_1 \text{ and } T_2 \text{ are not unifiable} \\ suspend & \text{otherwise} \end{cases}$$

## 4.1 Syntax

The syntax of FCP(:,?) is the same as that of FCP(:) with the additional '?' (*read-only*) symbol. A read-only symbol may be appended to variables. A variable annotated with the read-only symbol is called a *read-only variable*. Only the tell part and the body of a clause in FCP(:,?) may contain read-only variables.

## 4.2 Operational Semantics

As for FCP(:), we define the operational semantics of FCP(:,?) using the above scheme for defining FCLP languages. The definitions are similar to those of FCP(:) with the modifications needed to incorporate read-only unification. The definition of the evaluation function in FCP(:,?) is the same as for FCP(:) Hence, we redefine here only the try function for FCP(:,?).

**Definitions:**

Given a goal $G$ and an FCP(:,?) clause $Head \leftarrow Ask : Tell \mid Body$.

- Let $\Psi = (\exists\theta \in ADMS.(Head\theta = G) \wedge \pi(Ask\theta) = true)$.
- Let $\Phi = (\exists\theta.(Head\theta = G\theta) \wedge \pi(Ask\theta) = true)$.

**Definition:** The *try function* for FCP(:,?) is a function

$$try_{FCP(:,?)} : \mathcal{GS} \times CS \longrightarrow TRY$$

such that:

$try_{FCP(:,?)}(G, C)$

$$\ni \begin{cases} \theta \circ \sigma & \text{if } \Psi \wedge \ \sigma \in mgu_?(Tell\theta) \\ fail & \text{if } \neg\Phi \vee (\Psi \wedge \ fail \in mgu_?(Tell\theta)) \\ suspend & \text{otherwise} \end{cases}$$

where $\theta$ is a most general substitution that satisfies $\Psi$.

Note that the definition of the try function of FCP(:,?) is different from the definition of the try function of FCP(:) only in the case of success.

### 4.3 Examples

In this section we present examples of the programming techniques achievable in FCP(:,?), due to the read-only variable.

**Test-and-set**

A variable X is tested to actually be a variable and then set to a non-variable term. This is done by a clause of the form:

```
test_and_set(X,T) ← true : X=Y?, Y=T | true.
```

This clause try will succeed only if X is a variable, in which case it will set X to the term T.
Note that the order in which the unification goals occur in the guard is immaterial, according to the definition of $mgu_?(Tell)$.

**Anonymous mutual-exclusion**

A multiple-writer stream, which preserves message multiplicity even in the presence of unifiable messages can be defined, using the above test-and-set technique, as follows:

```
write(M,Ms,Ms') ←
      true : Ms=[X?|,Ms'], M=X | true.
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.
A stream imposes total order on messages. The channel data-structure [Tri87] generalizes this idea for a partially ordered set of messages.

**Protected data-structures**

Another important application of read-only variables is to protect processes communication 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 bind some variable it binds this varibale itself to some erroneous value, it may cause the operating system process to fail.

Read-only variables allow a simple solution. 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 consumers, and keeping the writable access to it to oneself. This is achieved by placing a read-only variable *X?* in every 'hole' in the data structure, and keeping *X* to oneself. For example, protected-stream producer can be defined as follows:

```
p(Xs,...) ← Xs=[Message|Xs'?], p(Xs',...).
```

## 5 Ask and Tell Guards

The set of guard predicates is the same in FCP(:,?) and in FCP(:). The guard predicates are divided into two sets of predicates: ask predicates and tell predicates. The ask predicates appear only in the ask part of an FCP(:,?) clause and are used for testing arguments of the global environment. The ask predicates do not write on the global environment, i.e. they do not bind variables of the goal, and do not allocate any new local variables. The tell predicates appear only in the tell part of an FCP(:,?) clause and can assign values to goal variable or generate new variables.

We specify here the two sets of predicates. For each predicate we define its syntax and operational semantics. The list of the guard predicates is derived from the Logix user manual [Sil88].

An FCP(:,?) *expression* consists of variables, constants, arithmetic or boolean operators applied to expressions or one of a fixed set of functions applied to expressions. The set of functions is arbitrarily chosen and can be extended by functions like *sine*, *cosine* or boolean operations on bit strings, etc. The question of how to allow user programs to extend the set of arithmetic functions remains open. $value(E)$ denotes the *arithmetic value* of an expression $E$. The *computation* of the expression, denoted by $comp(E)$, succeeds, given the arithemetic value of the expression, if all the operands of the expression are valid, fails if one of the expression's operand is invalid, fails causing an arithemetic exception if such an exception occurs and suspends otherwise. The full definition of these notions appears in the full version of this paper [Kli88b] due to space limitations.

### 5.1 Ask Guards

In this subsection we list the ask predicates of FCP(:,?). For some predicates we define their operational semantics, i.e. when the predicate succeeds, fails or suspends. The operational semantics is defined using the evaluation function. First, we define the notion of monotonicity of an ask goal.

**Definition:** Let $G$ be an ask goal and $\pi$ an evaluation function. $G$ is *monotonic* with respect to $\pi$ if the following holds:

1. Success is stable : $\pi(G) \rightarrow \forall\theta.\pi(G\theta)$.
2. Failure is stable : $\pi(G) = false \rightarrow \forall\theta.\pi(G\theta) = false$.
3. Suspension is not stable : $\pi(G) = suspend \rightarrow \exists\theta.\pi(G\theta) \neq suspend$.

Hence, for a monotonic guard $G$, it is enough to define the condition for its success; once defined the rest is deducible. $G$ fails if there is no instance of $G$ for which it succeeds, and suspends otherwise.

All the ask predicates of FCP(:,?), except the predicate **unknown**, are monotonic with respect to their evaluation function. Hence, for each ask goal $G$ we give the condition for its success, i.e. the condition that implies $\pi(G) = true$.

Table 1 contains the list of the ask predicates of FCP(:,?), divided into five categories: **unification predicates** for checking if two terms are equal, **type checking predicates** for checking the type of an argument, **arithmetic predicates** for arithmetic comparisons of arithmetic expressions, **term comparison predicates** for comparing different types of terms and **term inspection predicates** for checking subarguments of terms. The full definition of the predicates' syntax and semantics is presented in [Kli88b].

### 5.1.1 The *unknown* Guard

The guard **unknown** is not a monotonic predicate. The guard never suspends, it either succeeds or fails, and success is not stable, i.e.

$$\pi(unknown(X)) = true \not\rightarrow \forall\theta.\pi(unknown(X\theta)) = true.$$

**unknown(X)** –
**succeeds** if $X$ is a variable.
**succeeds** or **fails** if $X$ is non-variable[1].

### 5.1.2 The *otherwise* Guard

**otherwise** –
**succeeds** when all textually previous clauses fail.
**suspends** otherwise.

### 5.1.3 The *var* Guard

The guard **var** is not a monotonic guard. The guard never suspends, it either succeeds or fails and success is not stable.

[1]The fairness condition guarantees that if $X$ is bound to non-variable term then eventually $\pi(unknown(X)) = false$.

**var(X)** –
**succeeds** if $X$ is a variable.
**fails** if $X$ is non-variable.

The guard **var** is <u>not</u> defined as part of FCP(:) or FCP(:,?) since most of the programing techniques achieved by this guard are achievable by using the read-only variable and/or the **unknown** guard. Implementing the guard requires locking the variable being checked, thus causing an unnecessary overhead. However, the guard **var** can be used to achieve some of the programming techniques that are otherwise achievable only by the read-only variable. These techniques include test-and-set, anonymous mutual exclusion, multiple-writer stream and distributed queues. An alternative construct called *inform*, which also achieves these capabilities, was proposed by Saraswat [Sar88c].

| Category | The Guards | |
|---|---|---|
| unification | equal | $=$ |
| | inequality | $\neq$ |
| type checking | integer(·) | |
| | real(·) | |
| | number(·) | |
| | string(·) | |
| | constant(·) | |
| | tuple(·) | |
| | known(·) | |
| arithmetics | arithmetic equal | $=:=$ |
| | arithmetic not equal | $\backslash=:=$ |
| | less than | $<$ |
| | less than or equal | $\leq$ |
| | greater than | $>$ |
| | greater than or equal | $\geq$ |
| term comparison | term less than | $@<$ |
| | term less than or equal | $@\leq$ |
| term inspection | arg(·,·,·) | |

**Table 1: Ask Guards**

## 5.2 Tell Guards

In this subsection we list the tell guards of FCP(:,?). All the tell guards, except for the **unify** guard, have preconditions. The pre-conditions, which are added to the ask part, have to succeed in order for the tell guard to succeed. For each tell guard we specify its preconditions. A tell guard that has preconditions implicitly adds them to the ask part. In fact, by adding those preconditions we implicitly associate with each tell guard an input relation, i.e. the input mode of the guard's arguments is statically defined.

**Example:** The tell guard $X := Y + Z$ has the precondition: $number(Y)$, $number(Z)$. Hence, the clause

$p(Y, Z) \leftarrow$ true : $X := Y + Z \mid \ldots$

is equivalent to the clause

$p(Y, Z) \leftarrow \text{number}(Y), \text{number}(Z) : X := Y + Z \mid \ldots$

If $(number(Y), number(Z))$ succeeds then the implied unification is

$$\{X = value(Y) + value(Z)\}.$$

The notion of monotonicity is the same for the ask guards and the tell guards, i.e, both ask guards and tell guards are monotonic with respect to their evaluation function if success and failure are stable and suspension is not stable. Like the ask guards, it is sufficient for monotonic tell guards to define when they succeed and what is the set of unifications implied by the tell guard.

Table 2 contains the list of tell guards of FCP(:,?) divided into four categories: **unification guards** for unifying two terms, **arithmetic guards** for unifying the arithmetic value of an expression with a term, **term creation guards** for creating terms and **type conversion guards** for converting terms of one type to another. The full definition of the guard's syntax and semantics is presented in [Kli88b].

| Category | The Guards |
|---|---|
| unification | unify = |
| arithmetic | assignment := |
| term creation | make_tuple(·,·) |
| type conversion | string_to_dlist(·,·,·)<br>list_to_string(·,·)<br>tuple_to_dlist(·,·,·)<br>list_to_tuple(·,·)<br>convert_to_number(·,·)<br>convert_to_string(·,·) |

**Table 2: Tell Guards**

## 6 Discussion and Conclusion

In this paper we have described a formal framework for defining flat concurrent logic programming languages. We use this framework for defining the languages FCP(:) and FCP(:,?). The language FCP(:,?) extends FCP(:) with read-only unification. Thus, in addition to the programming techniques of Flat GHC and FCP(|,↓), all the techniques of FCP(?) that rely on read-only variables are available. The added expressiveness comes at a cost: while Flat GHC and FCP(:) are all monotonic languages [Sar87] (except for the unknown/1 predicate), FCP(:,?) is not. As we have shown, this language attempts to integrate the best of several existing flat concurrent logic programming languages. The novelty of FCP(:,?) lies in its ability to accomodate all useful concurrent logic programming techniques, including those which rely on atomic test unification and read-only variables, while at the same time providing the flexibility to remain within a more simple subset of the language when the full power of atomic unification or read-only variables is not needed. In addition, the authors believe that FCP(:,?) can be implemented efficiently using the decision tree compilation technique, which has already been proven to be an efficient compilation method for flat concurrent logic programming languages [Kli88a]. Our future work will attempt to verify this assumption.

## Acknowledgments

We would like to thank Michael Codish, Peter Gerstenhaber and Bill Silverman for their comments on previous drafts.

## References

[Cla81] K.L. Clark and S. Gregory, "A relational language for practical programming", *Proc. Conf. on Functional Programming Languages and Computer Architectures*, ACM , pp. 171-178, October 1981. Also Chapter 1 in [Sha87]

[Cla84] K.L. Clark and S. Gregory, "Notes on Systems Programming in PARLOG", *Proc. International Conference on Fifth Genaration Computer Systems*, pp. 299-306, ICOT, 1984

[Fos87] I. Foster and S. Taylor, "Flat PARLOG: A basis for comparison", Technical Report CS87-13, Department of Computer Science, The Weizmann Institute of Science, 1987.

[Fos88] I. Foster, "Parallel Implementation of PARLOG", To appear in *Proc. International Conference of Parallel Processing*, 1988.

[Gal88] J. Gallagher and E. Shapiro, "Using safe approximations of fixed points for analysis of logic programs", *META88, Proc. of the workshop on Meta-Programming in Logic Programming*, pp. 185-198, June 1988.

[Ger88] R. Gerth, M. Codish, Y. Lichtenstein and E. Shapiro, "Fully abstract denotational semantics for flat concurrent prolog", *Proc. Third Annual Symposium on Logic in Computer Science*, pp. 320-333, IEEE, 1988.

[Hou86] A. Houri,E. Shapiro, "A sequential abstract machine for Flat Concurrent Prolog", Chapter 38 in [Sha87].

[Kim87] Y. Kimura and T. Chikayama, "An abstract KL1 machine and its instruction set", *Proc. Symposium on Logic Programming*, pp. 468-477, IEEE, 1987.

[Kli88a] S. Kliger and E. Shapiro, "A Decision Tree Compilation Algorithm for Flat Concurrent Prolog", *Procceedings of the Fifth International Con-*

*ference and symposium on logic programming*, K. Bower and R.A. Kowalski (eds.), MIT Press, pp. 1315-1336, 1988.

[Kli88b] S. Kliger, E. Yardeni, K. Kahn and E. Shapiro, "The Language FCP(:,?)", Technical Report CS88-07, Department of Computer Science, The Weizmann Institute of Science, 1987.

[Lic87] Y. Lichtenstein and E. Shapiro, "Concurent algorithmic debugging", Technical Report CS87-20, Department of Computer Science, The Weizmann Institute of Science, 1987. Also *Proc. of the ACM Workshop on Parallel Debugging*, 1988.

[Llo87] J. Lloyd, *Foundations of Logic Programming*, $2^{nd}$ ed. Springer-Verlag, 1987.

[Mie85] C. Mierowsky, S. Taylor, E. Shapiro, J. Levy and M. Safra, "The design and implementation of Flat Concurrent Prolog", Technical Report CS85-09, Department of Computer Science, The Weizmann Institute of Science, 1985.

[Saf86] S. Safra and E. Shapiro, "Meta Interpreters for real", *Information Processing 86*, pp.271-278, North-Holland, 1986. Also Chapter 25 in [Sha87].

[Sar86] V. A. Saraswat, "Problems with Concurent Prolog", Technical Report CMU-CS-86-100, Computer Science Department, Carnegie-Mellon University, 1986.

[Sar87] V. A. Saraswat, "Merging Many Streams Efficiently: The Importance of Atomic Commitment", Chapter 16 in [Sha87].

[Sar88a] V. A. Saraswat, "A somewhat logical formulation of CLP synchronization primitives", *Proceedings of the Fifth International Conference and symposium on logic programming*, K. Bower and R.A. Kowalski (eds.), MIT Press, pp. 1298-1314, 1988.

[Sar88b] V. A. Saraswat, D. Weinbaum, K. Kahn and E. Shapiro, "Detecting stable properties of networks in concurrent logic progammimg langauges", *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pp. 210-222, 1988.

[Sar88c] V. A. Saraswat, *Concurrent Constraint Programming Languages*, Ph.D. Thesis, Carnegie-Mellon University, 1988.

[Sha83] E. Shapiro, "A Subset of Concurrent Prolog and its Interpreter", Chapter 2 in [Sha87].

[Sha87] E. Shapiro, ed., *Concurrent Prolog, Collected Papers*, The MIT Press, 1987.

[Sha88] E. Shapiro, "The Family of Concurrent Logic Programming Languages", Department of Computer Science, The Weizmann Institute of Science, 1988.

[Sil88] W. Siverman, M. Hirsch, A. Houri, E. Shapiro, "The Logix System User Manual, Version 1.3", Chapter 21 in [Sha87].

[Tak83] A. Takeuchi, "How to solve it in Concurrent Prolog", *unpublished note*, 1983.

[Tay85] S. Taylor, S. Safra and E. Shapiro, "Parallel implementation of Flat Concurrent Prolog", International Journal of Parallel Programming, Vol. 15, No. 3, pp 245-275. Also Chapter 39 in [Sha87].

[Tay87] S. Taylor, R. Shapiro and E. Shapiro, "FCP: Initial Studies of Parallel Performance", Technical Report CS87-19, Department of Computer Science, The Weizmann Institute of Science, 1987.

[Tay88] S. Taylor, *Parallel Logic Programming Techniques*, Ph.D. Thesis, Department of Computer Science, Weizmann Institute of Science (submitted).

[Tri87] E.D. Trible, M.S. Miller, K. Kahn, D.G. Bobrow, C. Abbott and E. Shapiro, " Channels: A Generalization of Streams", Chapter 17 in [Sha87].

[Ued85] K. Ueda, "Guarded Horn Clauses", in E. Wada (ed.) *Logic Programming, LNCS* **221**, pp. 168-179, Springer-Verlag, 1986. Also Chapter 4 in [Sha87].

# A Definitions and Notations

## A.1 Substitution, Instance, Composition and Most General Unifier

**Notations:** Let

- $\mathcal{W}\mathcal{V}$ denote a set of *writable* variables.
- $\mathcal{R}\mathcal{V}$ denote a set of *read-only* variables.
- $\mathcal{V}$ denote $\mathcal{W}\mathcal{V} \cup \mathcal{R}\mathcal{V}$.
- $Terms(\mathcal{V})$ denote all terms over the variables of $\mathcal{V}$ (within some fixed Herbrand universe).
- $vars : Terms(\mathcal{V}) \rightarrow 2^{\mathcal{V}}$ denote the function giving the set of variables appearing in term.

**Definition:** A *substitution* is a function $\theta : \mathcal{V} \rightarrow Terms(\mathcal{V})$, which is the identity function almost everywhere.

**Representation:** A substitution is usually represented by its finite set of non-identical pairs, i.e. let

$$dom(\theta) = \{X \in \mathcal{V} \mid \theta(X) \neq X\}$$

then $\theta$ is usually represented as the set:

$$\{X \leftarrow \theta(X) \mid X \in dom(\theta)\}.$$

**Notation:** For a term $T$ and a substitution $\theta$, $\theta(T)$ is denoted by $T\theta$.

**Definition:** Let $T$ and $T'$ be terms. $T'$ is an *instance* of $T$ if there is a substitution $\theta$ such that $T' = T\theta$.

**Definition:** Let $\theta$ and $\theta'$ be substitutions. The *composition operation* on substitutions $\theta \circ \theta'$ is a substitution such that for every term $T$,

$$T(\theta \circ \theta') = (T\theta)\theta'.$$

**Definition:** A *renaming of variables* in a term $T$ is a substitution:

$\{U_1 \leftarrow V_1, \ldots, U_n \leftarrow V_n\}, \forall i, 1 \leq i \leq n, U_i, V_i \in \mathcal{V}$
where $(\forall i \neq j) V_i \neq V_j$ and
$(vars(T) \backslash \{U_1, \ldots, U_n\}) \bigcap \{V_1, \ldots, V_n\} = \emptyset$.
Two terms $T_1$ and $T_2$ are *equal up to renaming*, denoted by $T_1 \approx T_2$, if there exists a renaming substitution $\theta$ such that $T_1\theta = T_2$.

**Definition:** A substitution $\theta$ is a *unifier* of $T_1$ and $T_2$ if $T_1\theta = T_2\theta$.

**Definition:** A substitution $\theta$ is a *most general unifier* of $T_1$ and $T_2$ if $\theta$ is a unifier of $T_1$ and $T_2$ and, for every other unifier $\theta'$ of $T_1$ and $T_2$, there is a substitution $\theta''$ such that $\theta' = \theta \circ \theta''$.

Let $S$ denote a set of substitutions. We define the function $mgu$,

$$mgu : Terms(\mathcal{V}) \times Terms(\mathcal{V}) \longrightarrow 2^S \bigcup \{fail\}$$

i.e. $mgu(T_1, T_2)$ returns the set of the most general unifiers of $T_1$ and $T_2$ if they are unifiable, and $\{fail\}$ otherwise.

## B FCP(:) syntax

**Definition:** The set C of legal clauses of FCP(:) is the minimal set $\mathcal{X}$ satisfying:

1. $H \leftarrow true : T \mid B \in \mathcal{X}$ where $T \in \mathcal{TELL}^*$.
2. $H \leftarrow A : T \mid B \in \mathcal{X} \Longrightarrow H \leftarrow A, G : T \mid B \in \mathcal{X}$
   if
   $G \in ASK \wedge vars(G) \subseteq vars(\{H, A\})$.
3. $H \leftarrow A : T \mid B \in \mathcal{X} \Longrightarrow H \leftarrow A, G : T \mid B \in \mathcal{X}$
   if
   $G = (t1 = t2) \wedge dom(mgu(t1, t2)) \subseteq vars(\{H, A\})$.
4. $H \leftarrow A : T \mid B \in \mathcal{X} \Longrightarrow H \leftarrow A, G : T \mid B \in \mathcal{X}$
   if
   $G = arg(T, I, S) \wedge vars(\{T, I\}) \subseteq vars(\{H, A\})$.
5. $H \leftarrow A : T \mid B \in \mathcal{X} \Longrightarrow H \leftarrow A, G : T \mid B \in \mathcal{X}$
   if
   $G = (t1 = \backslash = t2) \wedge$
   $(vars(t1) \bigcup vars(t2)) \subseteq (vars(\{H, A\}) \bigcup (\text{`\_'}))$.
6. $H \leftarrow A_1, \ldots, A_j, \ldots, A_n : T \mid B \in \mathcal{X} \Longrightarrow$
   $H \leftarrow A_j, \ldots, A_1, \ldots, A_n : T \mid B \in \mathcal{X}$
7. $H \leftarrow true, A_2, \ldots, A_n : T \mid B \in \mathcal{X} \Longrightarrow$
   $H \leftarrow A_2, \ldots, A_n : T \mid B \in \mathcal{X}$