# Programming in Janus

Vijay A. Saraswat, Xerox PARC    Ken Kahn, Xerox PARC
Jacob Levy, Technion, Haifa

(Extended Abstract, December 1989)

**Abstract**

In an accompanying paper we present the design and definition of Janus, a performance-efficient distributed constraint programming language [SKL90]. Janus is based on an algebra of infinite trees and hyper-multisets and enjoys the *failure-free property*: Janus computations cannot abort because the store is inconsistent. This is achieved by imposing syntactic restrictions which ensure that a variable may occur at most twice in a run-time configurations, and by implementing an annotation scheme which ensures that only one of these occurrences can be used to generate constraints on the variable.

Prima facie, Janus syntactic restrictions appear quite severe. In this paper we show that a large variety of programming idioms supported by eventual publication Herbrand languages such as Parlog and FGHC can be directly and naturally supported in Janus. Furthermore, because of Janus properties, it is possible for a Janus compiler to generate code to recycle memory when it becomes garbage. Because of these properties, it is possible for a Janus compiler to generate very efficient code. Indeed, in some cases, we argue that a Janus compiler for a sequential target machine can generate code comparable in efficiency to code generated from similar programs written in conventional imperative languages.

## 1 Introduction

Concurrent logic programming languages, have been thoroughly studied in the last few years. Many programming idioms for these languages have been discovered, which deal, for example, with data flow computations, producer-consumer synchronization, short circuits (used for the detection of stable properties of networks [SWKS88]), recursive doubling (technique for path shortening, using the "transparent forwarding" facility provided by the ability to transmit askers and tellers as messages), incomplete messages (messages whose responses can automatically—and efficiently—be routed back to the receiver), many-to-one communication using bags, techniques for modelling agents with mutable local state, etc. Many of these idioms are discussed in the literature in such places as [Sha83], [Kah89], [FT89], [Sar89], [Gre87], [Ued86], etc. Surprisingly, *all* of these idioms can be directly used in Janus. Some others, such as "producer protection" which were hitherto unavailable in the cc languages are also, quite surprisingly, obtained in Janus.[1]

In the rest of this section we summarize the main features of the constraint system $\mathcal{J}$ underlying Janus, and the syntactic restrictions enforced by the Janus system. [SKL90] discusses the underlying motivations in more detail; the discussion here is adapted from that paper. In the rest of this paper we assume a familiarity with the programming techniques discussed in the concurrent constraint (logic) programming literature. Here we focus on differences which result from the syntactic restrictions which Janus imposes, as well as the additional functionality which arrays and bags provide. We also point out where an implementation could take special advantage of Janus restrictions to produce particularly efficient code.

## 1.1 A review of Janus

The important ways in which Janus programs differ from those in other eventual publication Herbrand languages (e.g. FGHC, Strand) are

- *Syntactic restrictions on variables.* These restrictions are designed to be strong enough to ensure *at compile time* that no conflicting constraints can occur at runtime.[2]

- *Introduction of bags and arrays.* Rather than the usual domain of Herbrand terms, the domain of Janus objects are bags and arrays. Because of our desire to maintain the failure-freeness property of Janus programs we need to permit cycles in these bags and arrays. Hence, we define bags as infinitary "hyper-multisets" and arrays as infinite, finite-width tree. Bags are introduced to properly handle many-to-1 communication. Arrays are a generalization of terms (or tuples) which enable Janus programs to be competitive with convential imperative programs.

$\mathcal{J}$ is defined over a domain of objects that, roughly, corresponds to infinite, finite-width trees (called *arrays* since their sons can be accessed via arithmetic indexing) and infinitary "hyper-multisets". Two multisets are considered the same if there exists a bijection between the index sets of their immediate sons such that corresponding sons are recursively "bisimilar". This complication is necessary in order to be able to solve equations such as $X = \{X\}$. However, we have shown in [SKL90] that even if such "cyclic" bags or arrays are created, they cannot be accessed at run-time by any agent so that in practice the user does not

[1] Producer protection is also unavoidable—that is the essence of the "constraint programming without constraint-solving" slogan.

[2] In logic programming terminology we ensure that unification cannot fail.

| Constraint | Description | Status | Restrictions |
|---|---|---|---|
| Arithmetic | | | |
| $Y * Z$ | multiplication | Tell-only | number(Y), number(Z) |
| $Y + Z$ | addition | Tell-only | number(Y), number(Z) |
| Arrays | | | |
| array(X) | predicate | Ask-only | |
| $\|X\|$ | cardinality | Ask-only | array(X) |
| $\langle X_1, \ldots, X_n \rangle$ | array constructor | | |
| array(N,B) | constructor | Tell-only | $N \geq 0 \wedge$ fixed(B) |
| $\langle X_1, \ldots, X_n \mid X \rangle$ | array concatenation | | array(X) |
| array(N,A,T) | matched array of askers & tellers | Tell-only | $N \geq 0$ |
| $X \circ Y$ | array concatenation (constructor) | Tell-only | array(X), array(Y) |
| A.I | element selection | Tell-only | $0 \leq I, I < \|A\|$ |
| A[I..J] | subsequence selection (constructor) | Tell-only | $0 \leq I, I \leq J, J < \|A\|$ |
| $A[I_1 \rightarrow t_1, \ldots, I_n \rightarrow t_n]$ | Updating (constructor) | Tell-only | $0 \leq I_1, I_1 < \|A\|$ <br> $\ldots,$ <br> $0 \leq I_n, I_n < \|A\|$ |
| Bags | | | |
| bag(X) | predicate | Ask-only | |
| bag(X,Y) | constructor | Tell-only | |
| element(X,Y) | selector | Ask-only | $\neg bag(X) \wedge$ bag(Y) |

Table 1: Some basic constraints in $\mathcal{J}$

have to be concerned with writing programs that manipulate such objects. Nor need a Janus implementation be concerned with deciding whether two multisets are the same, since the syntactic restrictions of Janus ensure that this cannot be asked. As summarized in Table 1, $\mathcal{J}$ allows constraints for creating and accessing arrays and bags. It also allows "naive arithmetic", the sort of arithmetic allowed in the overwhelming majority of programming languages, namely, calculations.

Briefly, arrays of size N initialized to contain B (which is forced) may be created by array(N,B). This is is equivalent to $< B, B, \ldots, B >$ with N occurrences of B. An array of askers and tellers of size N is created by array(N, ^A, ^T) which is equivalent to the conjunction of $A =< X_1, \ldots, X_n >$ and $T =< \hat{~}X_1, \ldots, \hat{~}X_n >$. A.I selects the Ith element of A (using zero indexing). A[I..J] returns the portion of A between I and J inclusive. $A[I_1 \rightarrow t_1, \ldots, I_n \rightarrow t_n]$ returns an array of the same dimension as A and whose elements are the same except for the indices $I_i$ which are associated with the corresponding $t_i$.

Bags are constructed by bag(X,Y) which makes the bag of X (or its elements if X is a bag) and Y (or its elements if Y is a bag). element(X,Y) matches a bag which has at least one element which is either a value or a teller. The match binds X to one such element and Y to the remainder of the bag.

A Janus program is an unordered collection of clauses of the form

$\mathtt{p(t_1, \ldots, t_n) \leftarrow C \mid C_1, B.}$

where the $\mathtt{t_1}, \ldots, \mathtt{t_n}$ are $\mathcal{J}$-terms, $\mathtt{B}$ is a possibly empty conjunction of goals, $\mathtt{C_1}$ and $\mathtt{C}$ are $\mathcal{J}$-permissible Tell- and Ask-constraints respectively such that $\mathtt{C} \wedge \mathtt{C_1}$ is $\mathcal{J}$-satisfiable. In such a rule $\mathtt{p(t_1, \ldots, t_n)}$ is said to be the *head*, $\mathtt{C}$ the *guard*, and $\mathtt{C_1}, \mathtt{B}$ the *body*.[3]

Rules in Janus also satisfy certain syntactic conditions, involving the number of occurrences of variables, and the function "^". Let $K$ be the clause $\mathtt{h} \leftarrow \mathtt{C} \mid \mathtt{G}$. Say that $\mathtt{C}$ *forces* $\mathtt{X}$ if for some parameter $\alpha$, $\mathcal{J} \models (\mathtt{C} \Rightarrow \mathtt{X} = \alpha)$. Say that $\mathtt{X}$ is *present* in a term $\mathtt{s}$ if it occurs in $\mathtt{s}$ even if occurrences of $\hat{}\mathtt{X}$ are not considered. Say that a variable $\mathtt{X}$ *occurs actively* in $\mathtt{G}$ if the occurrence is in the first argument of an "=" goal or a "^" term, and *occurs passively* in $\mathtt{G}$ if it has a non-active occurrence in $\mathtt{G}$.

The intuitive idea of the following conditions is to ensure that at run-time a variable $\mathtt{X}$ "occurs" in at most two goals. Further exactly one of these occurrences (the *teller* for $\mathtt{X}$) is as the first argument of a "^" term. (No other restrictions are placed on the second occurrence, which is called the *asker* for $\mathtt{X}$.) The conditions will ensure that only the goal with access to the teller for a variable can impose constraints on this variable. The two variable occurence limit is relaxed when the variable is forced (since its ground). When the variable is an array, the conditions permit multiple occurences which refer to different portions of the array.

The conditions are as follows.

1. If any of $\hat{}\mathtt{t}$ or $\mathtt{t=s}$ occur in $\mathtt{G}$, then $\mathtt{t}$ must be a variable. For technical convenience, we also require that if any of $\mathtt{t.i}, |\mathtt{t}|, \mathtt{t[i_1 \rightarrow t_1, \ldots, i_n \rightarrow t_n]}, \mathtt{t[i..j]}$ occur in $K$, then $\mathtt{t}$ must be a variable. If $\mathtt{array(N, s,t)}$ occurs in $K$, then there must be variables $\mathtt{A}$ and $\mathtt{B}$ such that $\mathtt{s} \equiv \hat{}\mathtt{A}$ and $\mathtt{t} \equiv \hat{}\mathtt{B}$. Finally, for efficiency reasons, we require that if $\mathtt{element(t, u)}$ occurs in $\mathtt{h}$, then $\mathtt{t}$ and $\mathtt{u}$ must be variables unconstrained by $\mathtt{C}$.

2. For every variable $\mathtt{X} \in \mathbf{var}(K)$, unless $\mathtt{C}$ forces $\mathtt{X}$ or $\mathcal{J} \models \mathtt{C} \Rightarrow \mathtt{array(X)}$, $\mathtt{X}$ must occur at most twice in $\mathtt{p} \leftarrow \mathtt{G}$, and the following conditions must be satisfied:

   (a) $\mathtt{X}$ occurs actively in $\mathtt{G}$ only if it occurs passively in $\mathtt{G}$, or $\hat{}\mathtt{X}$ occurs in $\mathtt{h}$;

   The intuition is: the active occurrence will "produce" information which must "go" somewhere—either out to the environment through a teller in the head, or to an asker in the body.

[3] As usual, we allow the notation $\mathtt{p} \leftarrow \mathtt{C_1}, \mathtt{B}.$ to abbreviate a rule with an empty guard, and the notation $\mathtt{p.}$ to abbreviate a rule with an empty guard and body.

(b) `X` occurs passively in `G` only if it occurs actively in `G` or is present in `h`, and,

The intuition is: information "coming in" must come from somewhere, either from a producer in the body, in the case of a "local variable", or from the asker in the head.

(c) `X` does not occur twice in `h`.

This just for efficiency reasons.

3. If $\mathcal{J} \models$ `C` $\Rightarrow$ `array(A)`, for some variable $A$, then `^A` must not occur in `h`, and `A` may not occur actively in `G`. `A` may occur passively in `G` any number of times, provided that $K$ satisfies the *reference partition condition* for $A$.

With each variable $A$ such that `C` $\Rightarrow$ `array(A)`, we associate a constraint $\phi$, the reference partition condition, over $\mathbf{var}(K)$ and a new variable $X$. For the clause to be acceptable it must be the case that $\mathcal{J} \models$ `C` $\Rightarrow \exists$`X`$.\phi$. (Intuitively, the condition being enforced is that no two occurrences of $A$ "reference" a common element in the array, so that at run-time the two reference property is maintained—note that one occurrence is "taken up" by the occurrence in the head which is necessary if `C` is to be satisfied.) Let $A$ occur $m$ times in the body of the clause. Say that an occurrence of $A$ is *outer* if it is not in the $i$th argument of an `update` $(\_[\_ \rightarrow \_, \ldots, \_ \rightarrow \_])$ term, for $i > 1$. Say that an occurrence is *stand-alone* if at that occurrence $A$ is not the first argument of a $|\_|, \_[\_..\_]$ or `update` term. Then $\phi$ is the conjunction of the constraints $\phi_q$, obtained from the following table, for each outer occurrence $q$:

| Expression | Constraint |
|---|---|
| $A.I$ | $X.I = q$ |
| $A[I..J]$ | $\forall K.(I \leq K, K \leq J \Rightarrow X.K = q)$ |
| $\|A\|$ | *true* |
| $A[I_1 \rightarrow t_1, \ldots, I_n \rightarrow t_n]$ | $\forall K(0 \leq K, K < \|A\|, \wedge_{j=1}^{n}(K \neq I_j) \Rightarrow X.K = q)$ $\wedge \wedge_{j=1}^{n} ((\wedge_{l=j+1}^{n} I_j \neq I_l) \Rightarrow \phi_j)$ where $\phi_i$ is the conjunction of the conditions associated with each occurrence of $A$ in $t_i$. |
| $A$ standalone | $\forall K.(0 \leq K, K < \|A\| \Rightarrow X.K = q)$ |

Note again that variables present in the clause may occur in $\phi$. For example, if the only passive occurrences of `A` in the body of a clause are in the term `A[I` $\rightarrow$ `A.J, J` $\rightarrow$ `A.I]` (this is the *swapping* idiom for arrays in Janus), then

$$\begin{aligned}\phi \equiv\ & \forall K.(0 \leq K, K < |A|, (K \neq I), K \neq J) \Rightarrow X.K = 1 \\ & \wedge((I \neq J) \Rightarrow X.J = 2) \wedge X.I = 3\end{aligned}$$

Note that $\mathcal{J} \models \exists$`X`$.\phi$, hence the reference partition condition is satisfied for this clause, regardless of the ask condition for the clause.[4]

[4]The array reference partition condition can be relaxed if it can be inferred that certain

The dynamic semantics of Janus programs is that of cc programs, which is discussed in detail in [Sar89,SR90]. The only noteworthy point is that the Ask-and-Fix operator [Sar89, Chapter 8.4] is used instead of the Ask operator. This implies that at run-time *one* (of possibly many) most general matchers is chosen for the goal and head of a clause. It is the use of `element(X,B)` which can introduce multiple general matches.

More precisely, the operational semantics of a Janus program can be described in terms of a transition system over configurations of the form $\langle \mathtt{G}, \sigma : \mathtt{V} \rangle$, for $\mathtt{G}$ a conjunction of goals and $\sigma$ a constraint such that $\mathbf{var}(\mathtt{G}) \cup \mathbf{var}(\sigma) \subseteq \mathtt{V}$. The single transition rule is:

$$\frac{\begin{array}{l}(1)\ K \equiv (p(t_1, \ldots, t_n) \leftarrow C \mid C_1, B) \in P,\\ \qquad \mathbf{var}(K) \cap V = \emptyset, \mathbf{var}(p(t_1, \ldots, t_n)) = \{Y_1, \ldots, Y_k\}\\ (2)\ \mathcal{J} \models \sigma \Rightarrow \theta(C) \wedge (\wedge_{i=1}^{n} s_i = \theta(t_i))\end{array}}{\begin{array}{l}P \vdash \langle G_1 \wedge p(s_1, \ldots, s_n) \wedge G_2, \sigma : V \rangle \longrightarrow\\ \langle G_1 \wedge B \wedge G_2, (\sigma \wedge \theta(C_1)) : (V \cup \mathbf{var}(K)) \rangle\end{array}}$$

where $\theta$ is some substitution on $\mathbf{var}(K)$. (Here we presume that the program $P$ is obtained by closing the set of input clauses $P_0$ under all possible renamings of clauses. This is only for technical convenience.)

As a result of the above definitions we can show that Janus computations satisfy a number of run-time properties. Perhaps the most crucial is that no Janus computation can result in a store which is inconsistent. Note that this property is achieved even though constraints are being added to the store *without* checking for consistency at run-time. This follows from the interesting property that in any configuration obtained from the execution of a Janus program, if a goal $g$ has as subterm a term `^t`, then $t$ is a variable, say $X$, and no other goal contains `^X` as a subterm, and $X$ occurs in at most one goal in the configuration, ignoring occurrences of `^X`. Thus, at run-time, variables work as point-to-point communication channels, and only the agent which possesses the "teller end" can produce constraints on the variable (can send messages down the channel).

# 2 Standard data-flow programs

It should be apparent to the reader how conventional data-flow programs in the style of [Kah74] can be written in this style. Indeed, Janus can be thought of as being an indeterministic "data-flow" language in which receive and send rights for channels can be passed as (parts of) messages. Most systolic computations [Kun79] are just particular kinds of dataflow programs and can hence be expressed directly in Janus.

references are to forced values. An array consisting completely of ground values can be referred to without any of these restrictions.

**Producer-consumer interactions** We show how to set up a bounded buffer producer-consumer interaction between two agents. When the two agents are created, a new channel is generated, and the producer is given the teller for the channel, and the consumer, the asker. Subsequently, the producer chooses to generate messages on the channel through the teller in the usual stream-oriented way, and the consumer "receives" these messages through the asker.

```
goal ← producer(^B, [demand, demand, ..., demand | Ds]), consumer(B, ^Ds).

producer(^B, [demand | Ds]) ← B = [produced | Bs], producer(^Bs, Ds).
producer(^B, []) ← B = [].

consumer([produced | Bs], ^Ds) ← Ds = [demand | MoreDs], consumer(Bs, ^MoreDs).
consumer([], ^Ds) ← Ds = [].
```

*In a* Janus *implementation that reuses structures, if both goals reside in the same address space, the above program would run* in constant space*, very much in the same way as a concurrent shared memory program with atomic reads and writes. When the goal is executed, a buffer of length n is created. The* same *buffer gets reused over and over again, to represent two lists, the demand list, and the list of values produced that have not yet been consumed. This sophisticated behavior follows merely from the normal per-clause compilation process which recycles a cons that has been read (since it is the only reader and the writer after writing no longer has a reference) to be the cons that is being told. When, in this paper, we claim that a program runs in constant space we mean that for each clause of the program the same number and size data structures are constructed in the body as are read in the head.*

*In general, wherever possible, it is usually valuable to write programs whose clauses are "memory balanced", that is, they construct data structures using as many words of memory as used by the data structures read. In a structure-reusing implementation, the implementation of such a program proceeds by side-effecting the memory locations read. For example, the usual program for* `append/3` *can be compiled in such an implementation essentially to code that does a destructive* `nconc` *of the input lists.*

### 2.0.1 Short circuits

The standard short-circuit programming technique can be expressed in Janus by imposing a consistent directionality of information flow. Consider the following program sketch where information flows from left to right:

```
p(..., L, R) ← p(..., L, ^M), p(..., M, R).
p(..., L, ^R) ← R = L.
```

An initial query may be `p(done,^D),q(D)`. "q" can wait until `D=done` succeeds, before proceeding. A good implementation of Janus can exploit the directionality of these short-circuits so that it is never the case that `D` is bound to `done` via a long dereference chain.

In [SWKS88], streams of switches, a programming technique for repeated use of a short-circuit, was presented in the context of Herbrand programming.

```
p(..., L, R) ← p(..., L, M), p(..., M, R).
p(..., L, R) ← L = [M | L′], R = [M | R′], p(L′, R′).
p(..., L, R) ← R = L.
```

When an agent desires to close a switch *for the current phase*, it merely equates its left channel to a cons and its right channel to a cons, and equates their cars, recurring with the cdrs. This appears to be an instance where run-time unification is required, though it is guaranteed that the unification will succeed. Unification in this case provides a degree of decoupling between successive `p` agents: a `p` agent can decide to close a switch in the current phase without having to synchronize with any other agent. It records its decision in the store through the constraints it has imposed on its left and right channels. When all agents have imposed similar constraints, the end of the current phase can be detected by the agent holding both ends of the chain. Thus the bidirectionality of unification is put to good use. Note that the third rule in one step closes a switch for all future rounds.

This technique can be adapted to Janus in two interesting ways. The first adaptation serializes the technique but is very efficient and requires no consing:

```
p(..., L, R) ← p(..., L, ^M), p(..., M, R).
p(..., [M | L], ^R) ← R = [M | R′], p(L, ^R′).
p(..., L, ^R) ← R = L.
```

The second rule makes the agent wait for its leftmost neighbor to close its switch for this phase before closing its own and thereby informing its rightmost neighbor. The incoming cons `[M|L]` can be immediately recycled to construct `[M|R']`: effectively the *same* cons is passed around from the first agent to the last (and in fact may even be cycled for further rounds).

The version which does not serialize introduces `s` agents between every pair of agents on a short circuit. These agents act like buffers enabling the `p` agents to close their switches independently:

```
p(..., L, R) ← p(..., L, ^M1), s(M1, M2), p(..., ^M2, R).
```

```
p(..., ^L, ^R) ← L = [M | L′], R = [^M | R′], p(..., ^L′, ^R′).
p(..., L, ^R) ← R = L.

s([^M1 | L], [M2 | R]) ← M1 = M2, s(L, R).
s(^L, R) ← L = R.
```

Note that when a `p` agent terminates, the `s` agent to its right discovers a teller on its left input channel. It uses this teller to communicate the channel connecting it to the `p` agent to its right, and then terminates. As a result, termination of a `p` agent causes termination of the `s` agent to its right, but leaves the chain intact.

A common use of streams of switches is for phased computations where agents can proceed to the next phase only if the other agents have finished this phase. In Herbrand languages, each agent can read the rightmost end of the stream before proceeding. In Janus other means of one to many communication are necessary as discussed later.

### 2.0.2 Incomplete messages

A major source of simplicity and elegance in Janus-like languages is that servers can respond directly to service requests by communicating on ports embedded in the service request. This technique is illustrated by the following "queue-server" program:

```
queue(Input) ← split(Input, ^Enq, ^Deq), match(Enq, Deq).
split([enqueue(X) | Rest], ^En, De) ← En = [X | En1], split(Rest, ^En1, De).
split([dequeue(^X) | Rest], En, ^De) ← De = [^X | De1], split(Rest, En, ^De1).
match([X | Enq], [^Y | Deq]) ← Y = X, match(Enq, Deq).
```

These rules purport to define a predicate `queue/1`, which will receive a sequence of `enqueue` and `dequeue` messages over time and will be able to respond as if these operations were being performed on a queue.[5] Operationally, a `queue` agent splits into two agents. One monitors the input queue and distributes it into two queues, one consisting of the enqueue requests received (preserving order of arrival) and the other consisting of dequeue requests received. Each enqueue request contains the value to be enqueued. Each dequeue request contains a *teller* for a variable, the asker for which is retained by the agent sending the request. When an enqueue request arrives, it is communicated via the tell capability directly to the agent which requested that the dequeue operation be performed. (Note that the length of the dequeue list can be greater than the

[5]Typically the input stream is obtained by merging one stream from each client—the handling of many-to-one communication schemes is discussed in more detail below.

length of the enqueue list, so that the queue may be "negative" for a while.) If we wish to give clients separate "enqueue" and "dequeue" capabilities then only the `match/2` program is needed.

*Implementation-wise, an input list to* `queue/1` *is split up into two lists using half of the memory of the input list and freeing the other half. As soon as corresponding elements are matched up, pairs of conses are deallocated. No consing needs to be done internally since the messages from the clients provide more than enough memory to implement the queue.*

## 2.1 Recursive doubling

The parallel prefix problem is of substantial interest in the area of concurrent algorithms (see e.g. [KRM85]). We use it here to demonstrate how to achieve recursive doubling by sending askers and tellers as messages, thus achieving "transparent forwarding".

Given $n$ numbers $a_1, \ldots, a_n$, the problem is to compute the $n$ numbers $b_1, \ldots, b_n$ such that $b_j = a_1 \oplus \ldots \oplus a_j$ where $\oplus$ is a binary, associative operation. To be concrete, we shall assume that "$\oplus$" is "+" in the following.

To solve the problem, a chain of $n$ agents is generated at run-time. That is, each agent $\mathtt{pp}_i$ shares a variable with the $(i-1)$st agent (the *left* link for the $i$th agent and the *right* link for the $(i-1)$st agent) and with the $(i+1)$st agent. Each agent $\mathtt{pp}_i$ also has access to $a_i$.

In what follows it is useful to think of computation as proceeding in cycles—in each "cycle" an agent receives one message and sends another—even though, in reality, there is no central clock and all agents are executing asynchronously. Each message is of one of two types. First, it may be the constant `<>`. Alternately, it may be a three-element array: the *value*, the *left* connection and the *left2* connection.

An agent $\mathtt{pp}_i$ stores locally the following pieces of information: it stores $a_i$, its left and right connections on the chain, a variable that will be equated to the *final* value of the number at this agent, and another variable, which will represent a connection to the agent "two steps to its right". The role of the last variable—which is critical to the algorithm—is explained below.

When an agent is enabled initially, it sends a message to the agent on its right. In this message, the value component field contains the number at the agent, and the left component is a variable that may be used to receive the messages the agent will send subsequently. The left2 component contains a variable which will be constrained *in the next cycle* to a variable that may be used to receive the messages that the agent *to the left* of the agent sending this message, will send in future.

In each cycle, every agent reads the variable that connects it to the agent to its left. Each such variable will be instantiated to a message of the above type. The agent computes its new local value by performing the given operation ($\oplus$) on the value it receives from the left, and its local value. It also equates the variable in the left field in the message with the variable in the left2 field *in the message it sent out in the previous cycle.* It recurs with its new left connection being the variable in the left2 component field in the message it just read.

If the agent reads a <> message then it sends <> on the other two connections that it has locally, declares that the value it was supposed to compute is its current value and terminates.

```
pp(In, Out) ← spawn(In, <0, <>, <>>, ^Right, Out), sink(Right).

spawn([N | Rest], Left, Right, ^Out) ← number(N) |
    pp(N, Left, ^Right1, ^Right2, ^Final),
    Out = [Final | Out1],
    spawn(Rest, <N, Right1, Right2>, Right, ^Out1).
spawn(<>, Left, ^Right, ^Out) ← Right = Left, Out = <>.

pp(N, <>, ^Right, ^Right2, ^Final) ← Final = N, Right = <>, Right2 = <>.
pp(N, <M, Left2, Left>, ^Right, ^Right2, Final) ←
    pp(M + N, Left, ^Right1, ^Right21, Final),
    Right = <M + N, Right1, Right21>, Right2 = Left2.
```

Note that an agent does not *use* the value of the variables `Left` and `Right2`: it merely serves to equate them so that *some other agents* can directly communicate in the next round. This simple connection pattern achieves recursive doubling: in the $i$th cycle an agent receives messages from another agent $2^i$ steps away in the initial chain. Transparent forwarding is used to obtain the desired effect.

**MetaNote 2.1** *Point out that [HS86] has a number of algorithms which utilize recursive doubling and which can be used in this setting. However, to do so we may need listeners.*

# 3 Computing with arrays

We now demonstrate the flexibility of the richer Janus array constraints.

A major use of arrays is as a random-access data-structure internal to a process, which can be updated sequentially. This is illustrated by the following program for the "Dutch National Flag" problem [Dij76]:[6]

```
dnf(Cols, Sorted) ← dnf(Cols, 0, |Cols| − 1, |Cols| − 1, Sorted).
dnf(In, R, W, B, ^Out) ← R>W | Out = In.
```

[6]The problem is: given an array of colors red, white and blue rearrange it in one pass so that all the red tokens occur first, and are followed by all the white tokens, followed by the blue ones.

```
dnf(In, R, W, B, Out) ← R ≤ W, In.W = red | dnf(In[R → In.W, W → In.R], R + 1, W, B, Out).
dnf(In, R, W, B, Out) ← R ≤ W, In.W = white | dnf(In, R, W − 1, B, Out).
dnf(In, R, W, B, Out) ← R ≤ W, In.W = blue | dnf(In[B → In.W, W → In.B], R, W − 1, B − 1, Out).
```

The above program is almost a direct translation of the sequential program in [Dij76]. Note the idiom for swapping two elements of the array, and recall that assignments can be done in place. The code produced by a Janus compiler for this program should not be very different from code produced for the corresponding program in an imperative, sequential language.

In some cases the implementation can even update disjoint portions of the array in parallel. This is illustrated by quicksort:

```
q(A, ^B) ← |A| ≤ 1 | B = A.
q(A, ^B) ← |A| > 1 | split(A, ^K, ^E), sort(E, K, ^B).

sort(E, K, B) ← % Suspends until E is an array.
    0 < K, K < |E| − 1 | % and it is not already sorted.
    q(E[0..K], ^C), q(E[(K + 1)..(|E| − 1)], ^D), concatenate(C, D, B).
sort(E, 0, ^B) ← B = E.
sort(E, K, ^B) ← |E| − 1 = K | B = E.

concatenate(C, D, ^B) ← % Suspends until C and D are arrays.
    B = C ∘ D.

split(A, K, Result) ← split(A, 1, |A| − 1, K, Result).
split(A, I, Big, K, Result) ← A.0 ≤ A.I, I < Big |
    split(A, I + 1, Big, K, Result).
split(A, I, Big, K, Result) ← A.0 ≥ A.I, I < Big |
    split(A[Big → A.I, I → A.Big], I, Big − 1, K, Result).
split(A, I, Big, ^K, ^Result) ← A.0 ≤ A.I, I = Big |
    Result = A[0 → A.I, I → A.0], K = I.
split(A, I, Big, ^K, ^Result) ← A.0 ≥ A.I, I = Big |
    Result = A[0 → A.(I − 1), (I − 1) → A.0], K = I − 1.
```

Essentially, `split` pivots its input array `A` around `A.0`, in place. The two halves of the array can then be sorted recursively in place, and the results combined.

`sort` and `concatenate` are introduced as agents rather than coding them inline because they read arrays that are produced concurrently with their spawning. If, for example, `sort` were opened in the second rule of `q` then the array subsequence expressions would be correct only if `split` always produces an array of the right size as its third argument. Rather than relying upon a sophisticated compiler analysis to discover that this is indeed the case, we fold the consumer of the array into a `sort` agent which suspends until it receives an array.

*Using the above techniques for representing arrays, a compiler can generate code which will sort the input array in place:* `split` *will take its input array and do its updates in place to produce the pivoted array; this array is split into two smaller pieces (within* `sort`*) in constant time, these two pieces are recursively sorted independently, in place, and then the resulting pieces are joined together (in constant time since the pieces are contiguous) to create the final result.*

# 4 Many to one communication using bags

The communication mechanism discussed in Janus hitherto has focussed on point-to-point directional communication channels. An agent may have a fixed number of such channels. Such communication schemes are too simplistic, however, to deal elegantly with many to one communication. Consider for example a situation in which an a priori unbounded number of clients may seek service from a server.

The server will have read access to a bag. It will wait for requests by asking if its bag is `element(Request,Bag')`. It then responds to the `Request` and recurs with the `Bag'`. A client has a teller to an element of the bag, say `X`. It sends a request by telling `X=bag(Request,X')` and using `^X'` for subsequent requests. It can split its access `X` to the server by telling `X=bag(X1,X2)` thereby creating the two accesses `X1` and `X2`. A client drops bag reference by posting constraint `X={}`.

Consider the simple bank account as an example of how to instantiate this communication scheme:

```
account_receptionist({}, State).
account_receptionist(element(Req, Rest), State) ←
    account(Req, State, ^State1),
    account_receptionist(Rest, State1).

account(balance(^Value), Bal, ^NewBal) ← integer(Bal) |
    NewBal = Bal, Value = Bal.
account(withdraw(Amt, ^Ok), Bal, ^NewBal) ← Amt ≤ Bal |
    Ok = ok, NewBal = Bal − Amt.
account(withdraw(Amt, ^Ok), Bal, ^NewBal) ← Amt>Bal |
    Ok = nope, NewBal = Bal.
account(deposit(Amt), Bal, ^NewBal) ← NewBal = Bal + Amt.
```

The guard `integer(Bal)` was added to the first rule of `account` to indicate that `Bal` is forced and can therefore occur multiple times. We expect a good compiler based upon abstract interpretation can compile away this test.

**Note 4.1** We are considering relaxing our syntactic restrictions to allow Asks of the form $\texttt{element}(\texttt{t}_1, \texttt{Y})$, where $\texttt{t}_1$ can be a term. Some interesting programs can be written clearly and concisely using these more general ask constraints, in a fashion reminescent of *conditional critical regions* in languages based on imperative concurrency [BA82].

For example, `account` can be written to suspend a withdrawal until such time as, if ever, there is enough money in the account to service it:

```
account(element(balance(^Value), Rest), Balance) ← integer(Balance) |
    Value = Balance, account(Rest, Balance).
account(element(withdraw(Amt), Rest), Balance) ← Amt ≤ Balance |
    account(Rest, Balance − Amt).
account(element(deposit(Amt), Rest), Balance) ←
    account(Rest, Balance + Amt).
account({}, Balance).
```

The point to note is that a *withdrawal* message is not even "accepted" (removed from the input bag) until it is possible to service it. However note that the above code does not enforce first come first served policy on debits.

Similarly, one can implement stacks which block excessive pop requests until someo agent does a push:

```
stack({}, Value).
stack(element(pop(^Value), Rest), [V | State]) ← Value = V, stack(Rest, State).
stack(element(push(Value), Rest), State) ← stack(Rest, [Value | State]).
```

While quite convenient, allowing Asks of the form `element(t1,Y)`, where $t_1$ is a non-variable term, introduces complexities in the implementations of bags. The restricted use of bags (where `t1` is a variable whose other occurrence is in the body) can be implemented efficiently as a distributed queue. We are uncertain whether the added clarity and conciseness of some programs due to relaxing this restriction is worth the apparent implementational complexity. This is a topic of ongoing research. □

## 4.1 Merging techniques

**MetaNote 4.1** *May say something about how to implement various sorts of merges using bags.*

## 4.2 Mutual exclusion

**MetaNote 4.2** *Work out various schemes. Many communicating with one, essentially tokens with answer variables. Serializing access. After accepting a message ¡request, Req, Return¿, Req is equated with a structure which contains* *the resource, and a variable for its return. When the value is returned by the agent using the resource, the serializer goes back to accepting messages from others. It may be possible to build in some prioritization schemes which "delay" this process till later.*

*This also illustrates a randezvous style of communication.*

*As an illustration, deal with dining philosophers.*

### 4.3 Single exclusion

**MetaNote 4.3** *First illustrate single round, and then extend it for multiple rounds. Also, first illustrate how it can be done for the atomic languages, and then how it can be done here. Hence, this is part of another general theme which points out how particular atomic Herbrand idioms can be re-expressed in Janus.*

*Remember that this is dynamic single-exclusion. Can use the various merge-tree ideas that we developed for example when Ueda was here.*

*Investigate tradeoffs in expressiveness/implementation efficiency of splay merge-trees vs. bags on a multi-threaded single-node.*

## 5 One to many communication: distribute and copy

Consider the problem of broadcasting information received down one channel to a number of other agents. This can be dealt with quite straightforwardly by defining a predicate `distribute/3`. The first argument is the channel on which communication is received, the second receives the messages received on the first argument, and the third is an array of channels which receive *copies* (without embedded tell capabilities) of the messages. We call these channels which receive copies *listeners* of the channel being copied. Elsewhere we shall discuss the possibility of elevating listeners to become a part of the language.

Intuitively, the idea is clear. For every possible message that an agent can receive on its input channel, it should have a clause which suspends until that message is received and then copies it to the output streams.

```
distribute(X, Y, Z) ← distribute1(X, Y, ^Z1), distribute(Z1, Z).
distribute(X, <^Y>) ← Y = X.
distribute(X, Z) ← |Z| > 1 |
    distribute1(X, ^X1, ^X2), distribute(X1, Z[0..(|Z|/2 − 1)]), distribute(X1, Z[|Z|/2..(|Z| − 1)]).
distribute1(X, ^Y, ^Z) ← scalar(X) | Y = X, Z = X.
% X is a constant, number, or char, etc.
distribute1(X, ^Y, ^Z) ← array(X) | array(|X|, ^Y1, ^Y), array(|X|, ^Z1, ^Z), distribute_array(X, Y1, Z1).
distribute1(element(X1, X), ^Y, ^Z) ←
    Y = bag(Y1, Y2), Z = bag(Z1, Z2), distribute1(X1, ^Y1, ^Z1), distribute1(X, ^Y2, ^Z2).
distribute1(^X, ^Y, Z) ← Y = ^X0, distribute1(X0, ^X, Z).
```

```
distribute_array(<X>, <Y>, <Z>) ← distribute(X, Y, Z).
distribute_array(X, Y, Z) ← |X| > 1 |
    distribute_array(X[0..(|X|/2 − 1)], Y[0..(|X|/2 − 1)], Z[0..(|X|/2 − 1)]),
    distribute_array(X[|X|/2..(|X| − 1)], Y[|X|/2..(|X| − 1)], Z[|X|/2..(|X| − 1)]).
```

Note that if a teller arrives on the first argument it is forwarded to the second and the third argument will get copies of any value the teller later receives. To achieve this, in the last `distribute1/3 rule` we send the teller for a *new* communication channel `^Z` to the selected stream, and then recur with `Z` as the new "input" stream for the distributor. In this way the role of the asker and teller may ping-pong between the first two arguments of the distributor.

Note that this program is syntactically determinate. At run-time, for any call, at most one clause will be applicable. In [SKL90] we discuss how the idiom represented by the above program may often be used in-line in various algorithms.

The `distribute1` rule for arrays is an example of *array mapping*. The result of the mapping is returned immediately as the third argument of an `array/3` constraint. The second argument is then an array of tellers for the corresponding positions in the array of askers. The array of tellers is then "filled in" in parallel.

## 5.1 Techniques to circumvent sharing

At the same time, it is important to be aware that there are some programming idioms in (atomic and eventual) cc languages which cannot directly be achieved in Janus. For example, it is not possible in Janus to share a data-structure (e.g., symbol-table) between multiple agents in such a way that all of them cooperate in filling it out. The basic problem is that there does not seem to be a simple way to guarantee that multiple agents having access to the same variable will produce consistent information about that variable. On the other hand, it is possible to encapsulate the data-structure in an agent and have all other agents communicate with this "guardian". If the data-structure is recursive, a corresponding tree of agents may be generated, thus mitigating some of the serializing aspect of guardians. This is indeed the way computations are organized in conventional programming languages—the point is merely that within cc languages it is sometimes possible to give very elegant presentations which do not require explicit synchronization conditions at the language level. However, by going to Janus this flexibility is lost.

Sometimes the straightforward way of writing a dataflow program does not yield a syntactically legal Janus program; however with a little tweaking it may be possible to get such a program. For example the straightforward program for generating the first `(Count+2)` Fibonacci numbers has three occurrences of `Rest` in the second rule.

```
fib(Seq, Count) ← fibb([0, 1 | Seq], Count).

fibb([X1, X2 | ^Rest], Count) ← Count > 0 | Rest = [X1 + X2 | ^Rest1], fibb([X2 | Rest], Count − 1).
fibb([X1, X2 | ^Rest], 0) ← Rest = [].
```

On the other hand, the following trivial variation is acceptable Janus:

```
fib(Seq, Count) ← fibb([0, 1 | Seq], Count).

fibb([X1, X2 | ^Rest], Count) ← Count>0 | fibb([X2, X1 + X2 | ^Rest1], Count − 1), Rest = [X1 + X2 | Rest1].
fibb([X1, X2 | ^Rest], 0) ← Rest = [].
```

The rule will implicitly suspend until `X1` and `X2` are forced to be numbers; hence they can occur more than twice.

The following is how list intersection can be written as a Herbrand program using difference-lists.

```
intersect(L1, L2, L) ← intersect1(L1, L2, L, []).

intersect1([X | L1], L2, F, T) ← member_add(X, L2, F, M), intersect1(L1, L2, M, T).
intersect1([], L2, F, T) ← F = T.

member_add(X, [], F, T) ← F = T.
member_add(X, [X1 | L], F, T) ← X ≠ X1 | member_add(X, L, F, T).
member_add(X, [X | L], F, T) ← F = [X | T].
```

The difficulty with converting this program to Janus is that `L2` is shared by the `member_add` and recursive `intersect1` agents in the first rule of `intersect1`. One Janus programming technique is to pipeline the computation, so that the shared list is copied as it is being read:

```
intersect(L1, L2, L) ← intersect1(L1, L2, L, []).

intersect1([X | L1], L2, F, T) ← member_add(X, L2, F, M, ^L2′), intersect1(L1, L2′, ^M, T).
intersect1([], L2, ^F, T) ← F = T.

member_add(X, [], ^F, T, ^Out) ← F = T, Out = [].
member_add(X, [X1 | L], F, T, ^Out) ← X ≠ X1 | Out = [X1 | Out′], member_add(X, L, F, T, ^Out′).
member_add(X, [X | L], ^F, T, ^Out) ← F = [X | T], Out = [X | L].
```

*The implementation need not do any copying if the* `member_add` *and* `intersect1` *agents are executing in same address space. Essentially the data structure is being shared over time but only a single agent has access to any component at any one time. This serialization of access to a data structure is quite general, and the structure can contain tellers, unlike solutions which require copying.*

As we have seen, Herbrand programs naturally written with shared data structures can be rewritten in Janus either by explicitly copying the data (e.g. using `distribute/3`), by encapsulating the data in recurrent agents, or by pipelining or serializing the access.

## 5.2 Implementing balanced search trees

Binary search trees (bst) [AHU74] are a fundamental data-structure which can be used to implement many kinds of operations in a simple way. One of the major problems with this data-structure is how to keep the tree balanced. Various more or less complicated schemes, such as $(2:3)$ trees, AVL trees, etc. have been proposed [AHU74]. However, a very interesting result was obtained recently by [ST85] (see [Tar86] for an overview). It was shown that by performing local "splay" operations on a bst, it was possible to guarantee that the amortized cost over a sequence of $N$ operations on the tree would be $O(N\ \log(N))$. These trees are an example of an interesting class of data-structures called "self-adjusting data-structures".

Here we present a program fragment which implements accessing an item in a bst and adjusting the tree in the process. We use the notion of *difference-bsts*. There are two types of difference-bsts, a left one and a right one. The left one is a pair `T-L` where `T` is a bst and `L` is the *right* son of the node with the largest value in `T`. The right one is of the form `T-R` where `T` is a bst and `R` is the *left* son of the node with the smallest value in `T`. An empty bst is represented by a variable. Hence `L-L` denotes the empty left (as well as right) difference bst.

As discussed in [ST85], we start with a notion of a left *fragment* and a right fragment of the new bst to be constructed. Initially, the two fragments are empty.

```
bst(Op, Item, Tree, NewTree) ← bst(Op, Item, L − ^L, Tree, R − ^R, NewTree).
```

We now consider the base cases. We represent the empty tree as a teller; and every other tree as a three-element array . Hence, it is possible for a clause to match a bst with an asker for a variable X, and in the body bind the teller for X with `null` just in case the bst is empty. On the other hand, if the `Item` is found at the root, then the call terminates, with the new tree being set up appropriately.

Thus the base clauses are:

```
bst(access(^Val), Item, L, nil, R, ^New) ← New = nil, Val = null.
bst(access(^Val), Item, Left − ^LT, <Item, A, B>, Right − ^RT, ^New) ←
    Val = true, LT = A, RT = B, New = <Item, Left, Right>.
```

We now consider the zig case, namely that we have reached a node such that the required `Item` is either to the left of the current node and the current node is a leaf, or the required item is the left son of the current node. Depending upon the operation requested, the appropriate action is taken:

```
bst(access(^Val), Item, Left − ^LT, <X, null, B>, Right − ^RT, ^New) ← Item<X |
    LT = null, RT = B, New = <X, Left, Right>, Val = null.
bst(Op, Item, Left, <X, <Item, A1, A2>, B>, R − ^RT, New) ← Item<X |
    RT = <X, NR, B>, bst(Op, Item, Left, <Item, A1, A2>, R − ^NR, New).
```

The recursive cases are straightforward. First the Zig-Zig case:

```
bst(Op, Item, Left, <X, <Y, Z, B>, C>, R − ^RT, New) ← Item<X, Item<Y |
    RT = <Y, NR, <X, B, C>>, bst(Op, Item, Left, Z, R − ^NR, New).
```

Now the Zig-Zag case:

```
bst(Op, Item, L − ^LT, <X, <Y, A, Z>, C>, R − ^RT, New) ← Item<X, Y<Item |
    LT = <Y, A, NL>, RT = <X, NR, C>, bst(Op, Item, L − ^NL, Z, R − NR, New).
```

The symmetric cases for the right sons of the current node are straightforward too. The first two are base cases, the third is the zag-zag case and the last the zag-zig case.

```
bst(access(Value), Item, Left − ^LT, <X, B, nil>, Right − ^RT, ^New) ← X<Item |
    New = <X, Left, Right>, LT = B, RT = nil.
bst(Op, Item, L − ^LT, <X, B, <Item, A1, A2>>, Right, New) ← X<Item |
    LT = <X, B, NL>, bst(Op, Item, L − ^NL, <Item, A1, A2>, Right, New).
bst(Op, Item, L − ^LT, <X, C, <Y, B, Z>>, Right, New) ← X<Item, Y<Item |
    LT = <Y, <X, C, B>, NL>, bst(Op, Item, L − ^NL, Z, Right, New).
bst(Op, Item, L − ^LT, <X, A, <Y, Z, C>>, R − ^RT, New) ← X<Item, Item<Y |
    LT = <X, A, NL>, RT = <Y, NR, C>, bst(Op, Item, L − ^NL, Z, R − ^NR, New).
```

*A very important property of the above program is that all the assignments can be done in place, and no consing needs to be done. In other words, each rule is memory conservative.*

**Historical Note 5.1** A complete version of this program in Prolog first appeared in a note that one of us (Vijay) sent to the Prolog Digest in 1987. It is rather remarkable that that program, even then, was more or less a Janus program. □

# 6 Additional remarks

Note that while it is possible for a group of Janus agents to create cyclic structures at run-time (e.g. `X=<X>`), such structures once created can never be referenced from the outside. This is an advantage that Janus holds over a language such as *Strand* which would allow the creation of such loops, causing the system to enter an infinite loop when it attempts to read such a structure. (Indeed, Strand may even loop if a user happens to write code that executes `X=X` at run-time!)

It turns out that within a single address space Janus already encapsulates memory and allows the communication of memory. For example, if `b` requires a block of 8 words to service a particular request, then that memory can be provided by a client `a` as part of the request.

```
a(^X), b(X).

a(^B) ← B = please_perform_service_x(array(8, 0), ^
    Reply, NextB), a1(Reply), a(^NextB).

b(please_perform_service_x(Memory, Reply, Next)) ← |Memory| > 7 | compute(Memory, Reply), b(Next).
```

We are exploring a class of Janus programs in which no memory is allocated at run time.[7] Rather memory gets passed around from the initial query. These computations start with some hunk of memory and pass it around to get work down. This could be useful inside embedded systems (toys, missiles, watches, whatever) where you want the program to run for years without a risk that it'll "run out of memory".

# 7 Summary

The Janus restriction to two variable occurrences rarely interferes with the common concurrent logic programming techniques. The benefits of failure-freeness and more efficient implementation typically outweigh the relatively minor difficulties in fitting one's programs to the syntactic restrictions of Janus. We present

[7]In general, this may require the ability to reuse the memory of data structures as process/agent records.

various programming techniques for dealing with the inability in Janus to directly share data structures. We have found that many programs are "memory conservative" in Janus, and many more can be re-written to be so.

Janus introduces arrays to the distributed constraint framework. In this paper we have illustrated common uses of these data structures. Many array programs can be compiled to work "in-place" and should be as efficient as those written in conventional imperative languages. In many cases, the array can be modified in place in parallel. Unlike conventional imperative programming such programs run in Janus with an assurance that there are no synchronization bugs in these "simultaneous" accesses.

Janus also introduces bags which provide a principled and efficient means of programming many-to-one communication that avoids much of the clumsiness and ad hocness of previous solutions. A less restricted use of bags provides excellent support for object-oriented programming.

**Acknowledgements.** The research underlying this paper has benefitted tremendously from many, many discussions with Mark Miller and Eric Dean Tribble. We also thank Danny Bobrow, Seif Haridi, Fernando Pereira, Saumya Debray, Udi Shapiro, Bill Kornfeld and Julie Basu for discussions and comments.

# References


[AHU74] Al Aho, John Hopcroft, and J.D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[BA82] M. Ben-Ari. *Principles of concurrent programming.* Prentice Hall International, 1982.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[FT89] Ian Foster and Steve Taylor. *Strand: New concepts in parallel programming.* Prentice Hall, 1989.

[Gre87] S. Gregory. *Parallel Logic Programming in Parlog.* International Series in Logic Programming. Addison-Wesley, 1987.

[HS86] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *CACM*, 29(12):1170–1183, December 1986.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J.L. Rosenfeld, editor, *Proceeedings of IFIP Congress 74*, pages 471–475., August 1974.

[Kah89] Kenneth Kahn. Objects – a fresh look. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 207 –224. Cambridge University Press, July 1989.

[KRM85] C. Kruskal, L. Rudolph, and Snir M. The power of parallel prefix. In *Proceedings of the 1985 Int. Conf. in parallel processing.* IEEE, August 1985.

[Kun79] H.T. Kung. Let's design algorithms for vlsi systems. Technical Report 151, CMU-CS, Jan 1979.

[Sar89] Vijay A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, January 1989. To appear, Doctoral Dissertation Award and Logic Programming Series, MIT Press, 1990.

[Sha83] Ehud Shapiro. A subset of Concurrent Prolog and its interpreter. Technical Report CS83-06, Weizmann Institute, 1983.

[SKL90] Vijay A. Saraswat, Ken Kahn, and Jacob Levy. Janus: A step towards distributed constraint programming. In *Proceedings of the North American Conference on Logic Programming*, October 1990.

[SR90] Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of Seventeenth ACM Symposium on Principles of Programming Languages, San Fransisco*, January 1990.

[ST85] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *JACM*, 32(3):652—686, July 1985.

[SWKS88] Vijay A. Saraswat, David Weinbaum, Ken Kahn, and Ehud Shapiro. Detecting stable properties of networks in concurrent logic programming languages. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC 88)*, pages 210–222, August 1988.

[Tar86] R.E. Tarjan. *CACM*, March 1986. Turing award lecture.

[Ued86] K. Ueda. *Guarded Horn Clauses.* PhD thesis, University of Tokyo, 1986.