A retrospective look at concurrent logic programming

Vijay Saraswat
(saraswat@parc.xerox.com)
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, Ca 94304

Evan Tick
(tick@cs.uoregon.edu)
Department of Computer Science
University of Oregon
Eugene, Or 97403

June 13, 1993

Abstract

We survey the development of the field of concurrent logic programming from the modernist stance of constraint programming. We discuss the basic conceptions of the field, the language designs it has engendered, areas of applicability of these ideas, the underlying semantic notions, and the implementation issues that arise. The paper assumes the reader is familiar with the basic ideas of (constraint) logic programming.

Contents

1 Introduction

Concurrent logic programming arises in a very natural way from considerations of parallel execution of (backward-chaining) logic programs. Unbridled parallel execution very quickly leads to wasteful nondeterministic computation. Goals that are wildly (infinitely) non-deterministic (e.g., `S` $>$ `0`) with `S` unknown) may be scheduled instead of others that may instantly prune search (e.g. `S = 1`). It is not hard to see that substantial gains in performance can be obtained if some operational mechanism were to be provided to *suspend* the execution of certain goals (e.g., `S` $>$ `0`) until certain bindings were obtained — the better to quickly fail some (possibly infinitely many) non-deterministic branches of that goal. Concurrent logic programming languages arose from a desire to explicitly introduce operations to provide this suspension, and to indeterministically select one of one of many competing clauses (commit, i.e., generalized "cut", operations).

From the time in the early 1980's when these ideas were initially recognized by Keith Clark and Steve Gregory [7] to 1987, the group of researchers developing these ideas were primarily concerned with identifying various plausible operational mechanisms for specifying these suspension conditions. The nature of work in this area at that time consisted of examining in detail the most general unification (mgu) algorithm as it was used in executing logic programs, and experimenting with "control annotations" to harness the "eager" execution it engendered. While some derided this activity as "hackers doing logic", others were fascinated by the obviously attractive paradigm for concurrent computation that was thus opened up (particularly by the work of [59]).[1]

The introduction of constraints into logic programming in 1987 [28] provided the first opportunity to establish a more substantial basis for concurrent logic programming. Michael Maher introduced in [41] a suspension condition in the language ALPS that was based on checking that the constraints in a clause were *entailed* ("validated") by the constraints that had been accumulated in the refutation. This opened the door to a *logical* treatment of the suspension condition in stark contrast to the operational hacks that had hitherto occupied the field. As sometimes happens, by stepping back to a more general framework (arbitrary constraint solving), the details of the particular situation (unification — constraint solving over finite trees) fell away and the basic underlying idea emerged in its essential simplicity and generality.

This basic idea, simply put, is *constraint-based computation.* What concurrent logic programming provided the first clear manifestation of was the idea that computation could be conceptualized as occurring through the interaction of simultaneously executing agents posting and checking constraints (on a set of variables) in a shared store [51].

[1] See the recent personal accounts by some of the original participants in the field reported in [1].

But then this style of communication and computation can be isolated from the context in which it was first discovered, and a framework for programming languages built around it from "first principles", as in the work on "process algebras" based on similar lines of development due to Hoare, Milner and their colleagues ([25,42] — see [44] for an overview). This was the origin of the conception of concurrent constraint programming. The move enables the idea to be treated in its own terms and allows all the standard apparatus of modern day programming languages (recursion, block-structure, scoping, modules, types etc.) to be used for developing the paradigm, without being necessarily encumbered by a somewhat ill-suiting parent paradigm. In retrospect, though, it is rather surprising that many of the natural operations one would use to obtain a programming language organized around the ideas of constraint-based communication are already available in logic programming.

In this paper we shall review the field of concurrent logic programming from this modernist stance of constraint programming: constraint logic programming is essentially concurrent constraint programming instantiated over the Herbrand constraint system over finite trees that underlies deduction in first-order logic (see Section 2). We shall emphasize the connection with the work in theoretical computer science on process algebras, rather than the view of concurrent logic programming as a deviant version of (backward-chaining, Horn clause) logic programming. (The latter view is adopted, for example, in another survey of this area, [124].) At the same time, we shall develop and emphasize (see below) a major attraction of the CCP framework — a sound and *complete* view of computation as deduction, a view that unfortunately is not possible if CCP is viewed as a special case of logic programming. In particular, we shall emphasize the three-way connection between the logical interpretation, operational semantics and denotational semantics of these languages. The work on algebraic axiomatizations of various congruences of interest is not sufficiently mature for us to do a proper survey; we shall, however point out results wherever there are some.

A theme that runs through this paper is the separation of the *determinate* CCP languages from the *in*determinate — the determinate languages already offer the power of constraint-based communications and expose the basic new conceptual and semantic ideas in this approach. In some sense, the extension of these semantic ideas to the indeterminate languages, while technically complicated, is not particularly specific to constraint programming. Essentially the same ideas used here have been shown to be applicable to other, different models of computation [15].

We also discuss in some detail the implementation issues that arise in realizing this rather high-level computation model. Over the last decade considerable practical experience has been gained in developing implementations of concurrent logic programming languages in uni-processors, shared-memory multiprocessors and distributed networks. The basic concepts are reviewed and the major results discussed.

A brief word about some themes and issues that are consciously not being developed in this survey. First, we have not adopted the point of view that concurrent logic programming languages are reasonable "machine" or "kernel" languages for advanced parallel and distributed computers. This is the view that underlay work at ICOT and at the Weizmann Institute in this area in the past decade, but its validity is still open to considerable question. For a more sympathetic viewpoint, see [124] and [1].

We have also chosen not to treat here the search-non-deterministic concurrent constraint programming languages, such as Herbrand [51], and languages based on the Andorra principle [62] (e.g., AKL [32]). There are three primary reasons. Space. Time. Substance – the state of our understanding of the possibilities, languages, and implementation tradeoffs for these languages still seems immature.

Logical account of CCP languages. We now describe briefly the basic intuitions behind the logical account of concurrent logic programming underlying this paper. The interpretation inherited from the view of these languages as variants of backward-chaining Horn clauses is not particularly satisfying, primarily because there is no good characterization of deadlocked computations, and of indeterminate computations. We shall review an approach that has recently been developed that provides concurrent constraint programs with an interpretation that is "dual" that of backward-chaining Horn clause logic programs. Programs are viewed as providing *necessary* rather than *sufficient* conditions for a predicate to hold: defintions of relations (such as `append`) specify what must necessarily follow *if* the relation is known to hold between some objects, about which some additional pieces of information may be known. For example, we may state that if `X`, `Y` and `Z` are in the `append` relation, and `X=[]`, then it must be the case that `Y=Z`. Inference progresses forward: from the knowledge that certain pieces of information are true of the objects of interest, some other pieces of information are deduced. Thus the operational interpretation of predicates is that of information transducers: given certain pieces of information, they produce (potentially) more pieces of information. It turns out that this characterization is sound and *complete* for determinate concurrent constraint programming computations in a precise and important sense.

Obtaining a logical characterization of indeterminate computations, is however, a more tricky matter. Only a partial solution — that of the characterization of "may" properties of indeterminate computations — seems possible. Even for that, we have to make the move to a more powerful notion of logic, *linear logic*, recently introdued by [22]. The underlying issue is this: if the programming language is indeterminate, it must be possible to obtain two different answers, say b and c from the same initial computational state a. But then it must also be possible to obtain the answer b *and* c from a, and this may not be acceptable. For example, suppose that the initial computational state describes a situation in which Alice's account contains \$100, and she has given a cheque to Bob for \$100, and a cheque to Charlie for \$100 (and she has no other cheques outstanding). There are basically two evolutions of such a system: one in which Bob's cheque is cashed and the other in which Charlie's cheque is cashed. However, if this indeterminate computation is expressible in logic, then there should be yet a third computation in which the *conjunction* of the two results is possible: Bob's cheque is cashed, and Charlie's cheque is cashed. This situation is clearly incoherent.[2]

The root of the problem lies in that (classical) logic is not able to distinguish the case in which a situation A can give rise to a conclusion B and it can give rise to a conclusion C from the case in which A can give rise to (B and C). For, from an operational perspective, the problem arises because in logic one is freely allowed to *duplicate* formulas: $A \wedge A$ is semantically identical to A. Therefore the system of atoms

```
debit(Account, alice(100, AckA), debit(Account, bob(100, AckB),
account(Account, 100)
```

is semantically indistinguishable from

```
debit(Account, alice(100, AckA), debit(Account, bob(100, AckB),
account(Account, 100), account(Account, 100)
```

The latter system can, however, give rise to a computation in which *both* cheques are cashed.

It is not suprising that indeterminism and concurrency thus bring to the surface the fundamental notions of *object identity* and duplication. It is precisely the commitment of classical logic to deal with "static" notions like truth and falsehood that let us down. What is required here is a system of logic faithful to a dynamic notion of non-duplicable *resources*, a logic that can distinguish between the two situations above. This kind of flexibility has recently been placed on the table by the advent of the so-called sub-structural logics, the most prominent of which is linear logic. Linear logic may be seen as arising from classical logic by dropping certain structural rules that allow formulas to be arbitrarily duplicated and eliminated during the course of a derivation. This causes the familiar conjunction and disjunction operations to split into two: the so-called *additive* versions that copy formulas, and the *multiplicative* versions which do not. This gives formulas the nature of *resources* that must be accounted for carefully during proof (computation). In particular, multiplicative conjunction ($\otimes$) is *not* idempotent: $A \otimes A$ is not logically equivalent to A. The lost power

[2] The first author is indebted to Carl Hewitt for making this argument.

of the structural rules can be recovered locally by means of the "modalities" ! and ?. In particular, the formula $!A$ is allowed to be duplicated any number of times, derelicted to obtain A, and dropped altogether.[3]

Below we shall develop in some more detail the characterization of indeterminate CCP in linear logic. For the time being, we note that linear logic allows one to distinguish the case in which from A we can get *both* B and C (this is represented as $A \multimap B \otimes C$) from the situation in which from A we can get *either* B or C (this is represented as $A \multimap B \,\&\, C$, where $\&$ is the operator for additive conjunction). (And this distinction arises precisely because the identification $A \otimes A = A$ is no longer valid.)

Rest of this paper. We first review the basic ideas behind the CCP paradigm. The determinate CCP languages are then presented — their syntax and operational semantics, the logical readings of programs, denotational semantics, and examples of computations in this style. A similar treatment is then carried out for the indeterminate CCP languages. For historical purposes, we then review the special features of some of the major concurrent logic programming languages that have been developed in the last decade. This also offers an opportunity to discuss some related sub-themes, such as atomic vs. eventual publicatio of constraints, failure-free computations, etc. We then survey the work on implementations of these languages, focusing on different architectural issues that arise. Finally we point to areas of active current work and provide an assessment of the future of the field.

[3] See [56,39] for more motivation and tutorial introductions to linear logic.

2 The Basic Paradigm

The fundamental move underlying concurrent constraint programming is to replace the notion of *store-as-valuation* behind imperative programming languages with the notion of *store-as-constraint*. A constraint may conceptually be taken to be a (possibly infinite) subset of the space of all possible valuations in the variables of interest, e.g. `X = f(a,Y)` denotes the set of all pairs $\langle x, y\rangle$ of finite trees such that the first is a tree with root labelled `f`, and with two sons, the first of which is a tree with root labelled `a` and with no sons, and the second is the tree y. Thus the store may describe only partial pieces of information about the possible values that variables can take. The situation in which the actual denotation of a variable (say, `X` = `a`) is permissible but the exception rather than the rule.

This paradigm shift renders the usual notions of of (imperative) "write" and "read" incoherent. There may be no single, finitely-describable value left to return as the result of a "read" operation on a variable. A "write" operation is only capable of taking a fully-formed concrete value for a variable, and updating the store to have this variable take on this new value. But the new store no longer has any connection with the old store — constraints satisfied by the old store may not be preserved at all by the new store. The underlying principle of *information preservation* and *accumulation* we are trying to build on is irretrievably compromised.

Instead, [50] proposes the replacement of read with the notion of *ask* operations and write with the notion of *tell* operations. An ask operation takes a constraint (say, c) and uses it to probe the structure of the store. It succeeds if the store contains enough information to entail c. Tell takes a constraint and conjoins it to the constraints already in place in the store. That is, the set of valuations describing the resultant store is the intersection of the set of valuations describing the original store and those describing the additional constraint. Thus, as computation progresses, more and more information is accumulated in the store—a basic step does not *change* the value of a variable but rules out certain values that were possible before. Thus, the store is *monotonically refined* — all the constraints that were true of the store before the operation (that is, were entailed by the store), continue to be true of the store *after* the operation.

Central to the notions described above is the idea of constraint systems as systems of partial information. Indeed, it turns out to be natural to adopt Scott's theory of information systems [57] for this purpose. For the time being, suffice to say that all the theory demands is a specification of the pieces of information (in finitely many variables) admissible in the system, and a specification of the entailment relation that holds between them.

The possibilities for concurrency are now quite apparent. Instead of a single agent interacting with the store via ask and tell operations, any number of agents can simultaneously interact with the store in such a fashion. Synchronization is achieved via a blocking ask—an agent blocks if the store is not strong enough to entail the constraint it wishes to check; it remains blocked until such time as (if ever) some other concurrently executing agents add enough information to the store for it to be strong enough to entail the query. Note, in particular, that even though the paradigm is based on the notion of a shared store, ideas such as "read/write locks" do not have to be introduced for synchronization. The basic reason is that only a benign form of "change"—accumulation of information—is allowed in the system.[4] If desired, indeterminacy can be introduced by allowing an agent to block on multiple distinct constraints simultaneously, and specify distinct actions which must be invoked if the corresponding ask condition is satisfied by the store.

Thus, in this view, an agent ("computing station") is thought of as an *information transducer*. An agent can either add a constraint to the store (tell), suspend until there is enough information in the store to entail a given constraint, or decompose into a number of other agents running in parallel, possibly with hidden interconnections, and communicating and synchronizing via the store. Of course, as is standard, computations of unbounded length can be achieved by using recursion.

Compuational significance of the paradigm. While extremely simple, these ideas yield a powerful programming paradigm which derives its strength from the versatility of its communication mechanism. Essentially the entire range of "data structures" in computer science and AI opens up to examination. One can ask of graphs, binary trees, dictionaries, hash tables, association lists, finite sets, temporal intervals, real-valued temporal signals, etc.: what are coherent pieces of partial information about these objects and interrelationshps?[5] Indeed, a tremendous amount of work in computer science and AI is concerned precisely with elaborating such systems of partial information about objects of interest, and investigating their cost vs. expressiveness tradeoffs. Here, we shall just attempt to convey the basic idea, and refer the reader for more details to documents such as [51] and the review of constraint logic programming in this volume.

The essential idea is that variables serve as communication channels between multiple concurrently executing agents. The computational framework does not insist that there be an *a priori* partitioning of the agents into "producers" and "consumers" of information on the variables (as happens, for example, in

[4] Which of course, is not to say that systems with changable state are not describable in the cc paradigm. State change can be represented without compromising the basic paradigm by adapting standard techniques from logic and functional programming. Namely, "assignable variables" are embedded in the local state of a recursive agent—the agent "changes" the value of the variable by merely recurring with a new value for one of its arguments. In some languages in the cc framework (e.g., Janus[119]) there is considerable hope that such mechanisms for representing state change can be competitive in performance with traditional assignment-based techniques.

[5] As indeed these questions have been asked over the centuries for such "classical" data-structures as continuous real-valued functions.

function application, where information flows from an argument to the function body, or in CSP/CCS-style or actor-style languages). In fact, the *same* agent may simultaneously add or check constraints on the same variables.

Also, the computational framework allows for the underlying "network transport protocol" to perform (potentially arbitrarily sophisticated) inferences on the "messages" entrusted to them—these, of course, are the deductions sanctioned by the entailment relation of the constraint system. This allows each agent to state the constraints as it sees them and frees up the programmer from having to put in agents for explicitly collecting these messages and drawing the appropriate inferences ("reformatting the data"). This allows for very "high-level" languages in which essentially computation and communication can be expressed merely by posting and checking constraints.

The Herbrand constraint system. For concreteness, we consider the Herbrand constraint system that underlies logic programming languages. Assume given some underlying domain of denotations, a set I of individuals. The set of interest to us is the set of labelled, finitely branching, finitely deep trees over I, $\mathcal{T}(I)$. (Individuals are identified with the tree with 0 sons labelled with the name of the individual.) We shall be interested in two syntactic categories, Terms and Constraints, specified by the BNF grammar:

$$\begin{array}{llll} \text{(Terms)} & s,t & ::= & X \mid a \mid f(t_1,\ldots,t_n) \\ \text{(Constraints)} & c & ::= & (s=t) \mid \exists X.c \end{array} \tag{1}$$

Here X is a meta-variable ranging over some underlying domain of variables, a ranges over names for the individuals in I, and f over the permissible labels. A constraint $s = t$ is said to be satisfiable if there is an assignment of values to the variables in s and t such that the two terms denote the same element of $\mathcal{T}(I)$. Existential quantification is interpreted in the usual way. A collection of constraints $c_1, \ldots, c_n$ entails c if every valuation that satisfies each of the c_i also satisfies c. It is well known that the satisfiability and entailment algorithms for this constraint system can be solved essentially by the first-order most general unification algorithms. (See, e.g. [37] for a review of the basic ideas.) The fundamental properties of this constraint system are:

- two terms $f(t_1,\ldots,t_n)$ and $g(s_1,\ldots,s_k)$ denote the same tree iff $f = g$, $n = k$, and recursively, t_i denotes the same tree as s_i, for $i = 1,\ldots,n$,
- a constraint $X = t$ is satisfiable iff t does not contain X

As a result, consistency and entailment for this system can be checked in time proportional to the size of the formulas involved. In reality, this theoretical result is not of much use — rather what one is interested in is an assurance that the commonly occurring unification problems are solvable in *constant* time. As is the sad customary practice in logic programming implementations, this is guaranteed by ignoring the "occurrence check". In the spirit of constraint programming, this can be justified by claiming that the implementations are faithful to a constraint solver that works over the domain of rational rather than finite trees[6]. For the purposes of this survey, however, we shall ignore the complications that this compromise brings.

Even a constraint system as mundane as Herbrand already offers powerful possibilities for communication, leading to interesting idioms such as incomplete messages, short-circuits, difference-lists, "messages to the future" etc. [50]—the techniques that have colloquially been referred to as stemming from "the power of the logical variable". (These idioms shall be illustrated later in this paper.) For instance, it is possible to embed a communication channel (variable) in a message as a "first-class" object, in exactly the same way as data. A process p may post the constraint $\mathtt{X} = \mathtt{f(a, Producer)}$. Simultaneously and separately, another process q may post the constraint $\mathtt{X} = \mathtt{f(Item, Consumer)}$. And now q can discover that in fact the store implies `Item = a` (thus, the value `a` has been communicated from p to q — and in fact, to any other process that has access to (contains) the variable X). Further communication can occur on the channel accessed by p as `Producer` and by q as `Consumer` since indeed these have been identified (as a result of the two constraints on `X`).

This kind of communication of a channel in a message is the heart of the power of concurrent constraint programming. It allows for an elegant and simple form of dynamic reconfigurability of processes. This is difficult to achieve in a simple way within frameworks such as those of CCS and CSP.

[6]Rational trees are allowed to contain loops — the requirement is that they must have a finite number of *distinct* sub-trees.

3 Determinate concurrent constraint programming

Kahn's theory of determinate data-flow [34] remains the dominant theory of determinate concurrent computation. While simple, elegant and widely applicable, the theory's emphasis on computing with directional point-to-point communication channels imposes some expressibility restrictions: channels carry only fully formed values from the underlying data domain; networks are not dynamically reconfigurable in that connections cannot themselves be passed as data on communication channels; and each channel has only one reader and one writer. Because of these restrictions, Kahn's theory cannot model such useful programming idioms as incomplete messages, difference lists, recursive doubling, multiple readers/writers of a communication channel, dynamic reconfiguration, etc. Although these techniques were originally developed in the context of indeterminate concurrent logic programming, they are all both inherently determinate and usefully expressible in determinate contexts. s

3.1 Syntax and operational semantics

What should a programming language for the paradigm of store-as-constraints look like? We shall develop here a simple concrete syntax influenced by the syntax of linear logic. For the determinate languages, we could use a syntax from classical logic, since it will turn out that the execution mechanism (operational semantics) for determinate cc programs is complete for classical derivability (and hence also linear derivability). This, however, will not be the case when we move on to indeterminate languages: only completeness with respect to linear derivability will be obtained. So we prefer to use linear syntax up front. This will involve using $\otimes$ for parallel composition instead of $\wedge$, $\multimap$ instead of $\rightarrow$, and explicitly using ! to indicate reusability of formulas.

First, there has to be a mechanism for posting constraints to the store. Constraints are to be regarded as formulas that, once added to the store, are there forever:

$$(Agents) \quad A \quad ::= \quad !c \quad - \text{Tell} \tag{2}$$

Next, there has to be the possibility for checking that certain pieces of information are present in the store, and, if so, reducing to some other agent:

$$A \quad ::= \quad !c \multimap A \quad - \text{Ask Method} \tag{3}$$

In the following we will sometimes use the concrete syntax $c \rightarrow A$ instead of $!c \multimap A$ to emphasise that this usage of $\multimap$ is consistent with its interpretation as classical implication.

There should be a mechanism to execute more than one such agent in parallel:

$$A \quad ::= \quad A \otimes A \quad - \text{Parallel Composition} \tag{4}$$

In concrete syntax we shall write $A \otimes B$ as A, B.

It should be possible to introduce new communication channels private to a scope.

$$A \quad ::= \quad \exists X.A \quad \text{– Local Channels} \tag{5}$$

Finally, there should be the possibility of *replication*, of invoking an Agent an unbounded number of time. For this, we allow atoms with user-defined predicate symbols (called user-defined atoms; let their set be ranged over by the meta-variable H), and replicated definitions:

$$\begin{array}{rcll} A & ::= & H & \text{– Linear Message} \\ & & \mid\, !\forall \bar{X}.H \multimap A & \text{– Linear Method} \end{array} \tag{6}$$

We shall impose the syntactic restriction designed to avoid indeterminacy in process invocation.

Restriction 3.1 We shall require (1) For any given atom G, and linear method $!\forall \bar{X}.H \multimap A$, there should be at most one substitution θ such that $G = \theta(H)$. (This is true if G and H are Herbrand terms, but may not be true if they contain interpreted terms, e.g. $+, \cup$ introduced by an underlying constraint system.) (2) If an agent contains more than one linear method for a predicate letter, then the heads of these two methods shall not be unifiable, once existential variables are replaced with distinct constants. □

The above syntax is very flexible. It does not distinguish between *programs* and *agents*, like logic programming languages have traditionally done. This allows methods to be defined local to the scope of existentially quantified variables (e.q.v.'s) and is a power that we shall exploit subsequently. Furthermore, methods are allowed to have free variables that are quantified outside the method.

Note 3.2 Indeed, it is possible to show that with such dynamic specification of methods, it is not necessary to have any user-defined predicate letters except for one. Essentially, e.q.v. can play the role of constant names for new procedures. However, we shall not take that extra step here in order to remain close to the traditional syntactic presentation of concurrent logic programming languages. □

Note 3.3 Henceforth, when presenting programs in concrete syntax, we shall adopt the following conventions. We shall assume that whether an atom is a constraint atom or a user-definable atom is obvious by inspection. Constraints are assumed to be "!"ed, hence they need not be explicitly "!"ed (both in the left and right hand sides of implications). Also, we adopt the lexical convention that universally quantified variables (u.q.v) begin with a capital letter, and existentially quantified variables (e.q.v) with an underscore. A u.q.v. is considered scoped at the outermost arrow in whose left hand-side it occurs, and an e.q.v. is considered scoped at the largest conjunct containing all occurrences of that variable. This will allow us to almost always dispense with explicit quantifiers in programs. □

The translation from the conventional syntax for concurrent logic programs (e.g., FGHC) to this syntax is straightforward, as the following example shows.

Example 3.1 The `append/3` FGHC program:

```
append([], Y, Z) ← true | Y = Z.
append([A | B], Y, Z) ← true | Z = [A | C], append(B, Y, C).
```

is translated to

```
append(X, Y, Z) ⊸
    X = [] → Y = Z,
    X = [A | B] → (Z = [A | _C], append(B, Y, _C)).
```

□

Operational semantics. To present the operational semantics, it is most convenient to take a configuration to be merely a multiset of agents. (Multiset union is to be treated as conjunction.)[7] Intuitively, a configuration represents the parallel conjunction of agents that are currently present in the system. The collection of constraints in a configuration contitute the store. Below, we shall let Γ range over such a multiset, and C over a multiset of constraints. We shall assume given a relation $\vdash$ between finite multisets of constraints and a constraint, specifying the semantics of the underlying constraint system, in this case Herbrand.

There are four transition rules, corresponding to the four composite combinators in the language. An agent (A, B) decomposes instantly into its constituents:

$$(\Gamma, (A_1, A_2)) \longrightarrow (\Gamma, A_1, A_2) \tag{7}$$

We shall identify formulas that are the same upto renaming of bound variables. To execute an existentially quantified agent, then, we simply find a renamed version in which the bound variable is not free in the configuration:

$$(\Gamma, \exists X.A) \longrightarrow (\Gamma, A) \quad (X \text{ not free in } \Gamma) \tag{8}$$

[7]The approach we use here is a departure from the traditional SOS-style of giving operational semantics rule. However, it has the advantage of being extremely simple and drawing on the presentation of inference rules in a Gentzen-style sequent presentation of the underlying logic.

Computation occurs when a message meets a method. There are two kinds of reductions that can happen, depending on whether the formulas involved are reusable or not.

$$(\Gamma, C, !d \multimap A) \longrightarrow (\Gamma, C, A) \quad (\text{if } C \vdash !d) \tag{9}$$

A message can be consumed, if there is a method that matches, given the constraints in the store:

$$\frac{C, G \vdash H[\bar{t}/\{\bar{X}] \quad M = (\forall \bar{X}.H \multimap A)}{(\Gamma, C, !M, G) \longrightarrow (\Gamma, !M, A[\bar{t}/\bar{X}])} \tag{10}$$

As usual, a configuration Γ is said to be *stuck* if there is no configuration Γ' such that $\Gamma \longrightarrow \Gamma'$. A stuck configuration is said to have *terminated* if it consists of only reusable agents (methods and constraints). The result of a terminated computation is its store, existentially quantified with all variables other than those present in the initial configuration.

The fundamental property of the operational semantics is Church Rosser:

Theorem 3.1 ($\longrightarrow$ is Church Rosser.) *Let $\Gamma, \Gamma_1, \Gamma_2$ be configurations such that $\Gamma \longrightarrow \Gamma_i$, $i = 1, 2$. Then there are configurations Γ_1', Γ_2' such that $\Gamma_i \longrightarrow \Gamma_i'$ and Γ_1' is identical to Γ_2' upto renaming of e.q.v.'s.*

The basic reason, as always, is that no two reduction steps can be in conflict. If a step can be taken at any given stage of the computation, then the same step can be taken at any subsequent stage of the computation. Clearly, this is the case for steps involving Rules (7) and (8) (upto e.q.v. renaming). A reduction by Rule (9) once enabled is always enabled, since constraints added to the configuration are never deleted. There can be only one reduction involving a user-defined atom, by Convention 3.1, and that reduction once enabled will stay enabled. Hence any two basic reduction steps will commute.

The theorem immediately gives us that if an agent has a terminating reduction sequence then all its reduction sequences terminate, and in final stores that are equivalent when projected onto the variables in the initial configuration. Thus, determinate cc programs are, indeed, determinate.

3.2 Logical Semantics

The fragment of logic specified above is in some sense "dual" to the usual definite clause logic underlying logic programming: a clause is of the form $H \rightarrow A$, *not* $H \leftarrow A$; one describes *necessary* rather than sufficient conditions. Intuitively, one thinks of specifying for each predicate some pieces of information that must follow (for some of the arguments to the predicate) when some other pieces of information are know to be true about some arguments to the predicate. Each predicate is thus thought of as a transducer of (partial) information. One thinks of the underlying engine as making explicit pieces of information that are implicit in the original presentation of an agent — as explicit as is needed to answer a (primitive) test.

Lemma 3.2 *Let A be a linear formula, with free variables contained in the free variables of Γ. If $\Gamma \longrightarrow \Gamma'$, then $\Gamma \vdash A$ whenever $\Gamma' \vdash A$.*

This implies the *soundness* of CCP computations for this interpretation in logic:

Theorem 3.3 (Soundness of logical interpration) *Let d be a constraint with its free variables contained in those of Γ. If $\Gamma \longrightarrow^{\star} (\Gamma', C)$, and $C \vdash d$, then $\Gamma \vdash d \otimes \top$.*

In the following $\top$ is a linear logic constant with the following proof rule in a Gentzen style presentation of the logic:

$$\Gamma \vdash \top$$

In other words, any set of formulas can be used to establish $\top$.

Theorem 3.4 (Completeness of logical interpretation) *If $\Gamma \vdash d \otimes \top$, (with free variables of d contained in those of Γ, then there is a configuration Γ' such that $\Gamma \longrightarrow \Gamma'$ and $\Gamma' \vdash d \otimes \top$.*

The basic idea behind the proof is to examine the proof of $\Gamma \vdash d \otimes \top$ and from that proof construct an execution sequence that satisfies the given property.

The soundness and completeness results cited above could have been cast in intuitionistic and classical logic as well as linear logic. No essential features of linear logic are being used in the determinate CCP set-up. However, we are setting the stage for similar results for the indeterminate CCP languages — there we can obtain completeness only with respect to linear derivability.

3.3 Denotational Semantics

Above, we have seen that (terminating) determinate computations give rise to a unique store of constraints. One could then ask about the nature of the mappings from constraints to constraints established by the execution of determinate cc programs. [54] established that the right mathematical objects corresponding to determinate cc programs are *closure operators*. Closure operators are functions f over a partial order L that are extensive $f(x) \geq x$, monotone ($f(x) \geq f(y)$ whenever $x \geq y$) and idempotent ($f(f(x)) = x$). Idempotence says that the range of f is equal to the set of its fixed-points; extensiveness says that each point in the domain is mapped to a fixed-point above it, and monotonicity ensures that this must be the *least* such fixed point. Hence, a closure operator can be uniquely represented by the set of its fixed-points (its range).

Closure operators capture precisely the idea of monotonic accumulation of information inherent in the cc conception. On the one hand this discovery showed us that the mathematical objects underlying (at least the determinate) cc languages were very simple indeed, and had many interesting properties. (For example, the set of resting points of the process $f \wedge g$ is the intersection of the resting points of f and g — from this it is immediate that parallel composition is commutative, associative, idempotent etc.) On the other, by thinking of the cc languages as *the* language framework for computing with closure operators we were lead to a number of simple extensions with interesting computational content. (See Section 3.5.)

3.4 Programming idioms

To make it simpler to write programs over Herbrand, we shall adopt one more convention. Often, we shall want to specify that a parametrized constraint be checked, and allow the check to return values for the paramter. For instance, we may want to check whether `X` has been bound to a cons, and if so to obtain in `Y` the car and in `Z` the cdr. This can be done by checking the constraint `X = [Y | Z]`, where `Y` and `Z` are parameters. To allow for this we introduce:

$$A \quad ::= \quad \forall \bar{X}.c \multimap A \quad \text{– Ask Method} \tag{11}$$

If c is a Herbrand constraint then $\forall \bar{X}.c \multimap A$ is equivelant to $(\exists \bar{X}.c) \multimap \exists \bar{X}.(c, A)$. So the new form can be regarded merely as a convenient shorthand.

3.4.1 Simple data-flow programming.

It should be clear how Kahn data-flow networks can be programmed in cc languages. Essentially, these languages directly provide the ability to compose (recursive) networks concurrently, to establish connections between communication channels, and to hide local communication channels.

```
fib(Seq, Count) -o fibb([0, 1 | Seq], Count).
fib_1([X1, X2 | Rest], Count) -o
    Count > 0 -> (Rest = [X1 + X2 | _Rest1], fib_1([X2 | Rest], Count - 1)),
    Count = 0 -> Rest = [].
```

Shapiro's paper [58] examines many systolic programming idioms from the concurrent logic programming viewpoint.

3.4.2 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 | _D]), consumer(B, _D).
producer(B, X) ⊸
    X = [demand | D] → (B = [produced | _B], producer(_B, D)),
    X = [] → B = [].
consumer(In, D) ⊸
    In = [produced | B] → (D = [demand | _N], consumer(B, _N)),
    In = [] → D = [].
```

3.4.3 Short circuits

The standard short-circuit programming technique can be expressed thus:

```
p(..., L, R) ⊸
    ...
    c1 → p(..., L, _M), p(..., _M, R),
    c2 → R = L.
```

Here, c_1 is the condition under which a new switch must be spliced into the circuit, and c_2 the condition under which the circuit must be closed. An initial query may be `p(done, D),q(D)`. "q" can wait until `D=done` succeeds, before proceeding.

In [55], streams of switches, a programming technique for repeated use of a short-circuit, was presented in the context of Herbrand programming.

```
p(..., L, R) ⊸
    ...,
    c1 → p(..., L, _M), p(..., _M, R),
    c2 → (L = [_M | _L'], R = [_M | _R'], p(L', R')),
    c3 → 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.

3.4.4 Incomplete messages

A major source of simplicity and elegance in cc 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) -o (split(Input, _Enq, _Deq), match(_Enq, _Deq)).
split([Message | Rest], En, De) -o
    Message = enqueue(X) → (En = [X | _En1], split(Rest, _En1, De)),
    Message = dequeue(X) → (De = [X | _De1], split(Rest, En, _De1)).
match([X | Enq], [Y | Deq]) -o Y = X, match(Enq, Deq).
```

These rules 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.[8] 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 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.

3.4.5 Recursive doubling

The parallel prefix problem is of substantial interest in the area of concurrent algorithms (see e.g. [36]). We use it here to demonstrate how to achieve recursive

[8] 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.

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 ith 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) -o Right^spawn(In, [0, [], []], Right, Out).

spawn(In, Left, Right, Out) -o
    In = [] → (Right = Left, Out = []),
    In = [N | Rest], number(N)
     → ((Right_1, Right_2, Out_1)^
```

```
        pp(N, Left, Right1, Right2, Final),
        Out = [Final | Out1],
        spawn(Rest, [N, Right1, Right2], Right, Out1)).

pp(N, Left, Right, Right2, Final) -o
    Left = [] → (Final = N, Right = [], Right2 = []),
    Left = [M, Left2, Left]
     → (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 ith 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.

3.4.6 Translating the λ-calculus

Perhaps the most remarkable illustration of the power of constraint-based communication is the translation of the untyped λ-calculus into determinate CCP. There are a host of related translations; we treat here a translation of the lazy version of the lambda-calculus [2].

Intuitively, an application is treated as a parallel composition, and the operand and the argument are treated as separate processes communicating on a channel hidden from the outside world. The agent corresponding to the argument is sent a message with the identity of the operand, and the result channel. The operand is represented by a process that waits for an evaluation message (that will contain the result channel), and instructs its body to send its result on that channel. A λ-abstraction is seen as a server waiting for a message to come down its specific communication channel (such messages are generated by application). When this message arrives, the formal argument of the abstraction is bound to the (channel representing) the operand, and the body of the abstraction is asked to send its output to the result channel. The term x with result Z is represented by a message on the channel X asking for an evaluation, with the result to be sent to Z.

Let M be an arbitrary lambda-term, and Z an arbitrary variable. We define an agent $\langle M\rangle_Z$ by structural induction on M:

$$\begin{array}{ll} \langle X\rangle_Z & eval(X, Z) \\ \langle \lambda X\ M\rangle_Z & !(\forall X \forall Y\ apply(Z, X, Y) \multimap \langle M\rangle_Y) \\ \langle M N\rangle_Z & \exists X \exists Y.(\langle M\rangle_X, apply(X, Y, Z), !(\forall S. eval(Y, S) \multimap \langle N\rangle_S) \end{array}$$

For example, the agent $\langle \lambda x.x\rangle_Y$ is just:

$$!\forall B\ (apply(Y, X, B) \multimap send(X, B)$$

That is, $\lambda X.X$ is viewed as a server that waits for an argument (X), and return channel (B), and then sends B to X.

3.5 Other topics

3.5.1 Determinate Disjunction.

The simplest example of a new operator introduced by the understanding of determinate `cc` programs as closure operators is the *glb* operation. The set of closure operators forms a lattice, with the least upper bound given by parallel composition. What does the greatest lower bound of f and g give us? On any input c, it gives us the greatest lower bound (in the space of constraints) of $f(c)$ and $g(c)$ — one can think of it is a *determinate or*: it extracts the most *common* information between the two branches. [23] shows how this can be used to obtain considerable improvement in some constraint propagation algorithms.

Other examples of operations on constraints (e.g., indexical constraints, cardinality operators) that can readily be understood in terms of closure operators may be found in [23].

3.5.2 Nondeterminate `cc` language

It is natural to ask what happens when disjunction is added to the picture. This thesis shows one way of doing this operationally — one adds a notion of non-determinate prefixing ($\Rightarrow$), and takes as configurations multisets of pairs of agents and constraints.

However, it is not clear how to connect non-deterministic prefixing to logical notions. Instead [31] explores another route. Let $(A_1 \vee A_2)$ be an agent if A_1 and A_2 are. Operationally, as with nondeterminate prefixing, an agent $(A_1 \vee A_2)$ in store c splits into *two* configurations, $\langle A_1, c\rangle$ and $\langle A_2, c\rangle$. (Conjunctively running agents of $(A_1 \vee A_2)$ are copied into both.) When the store of one branch becomes inconsistent, the branch vanishes. The crucial question now is: what does one regard as *observable* for such a system of disjunctive agents?

One answer is that the observables of such an agent should be the multiset of constraints obtained on each branch. However, this conflicts with the operational requirement that "false" branches are unobservable: an agent which produces false on one branch and `X=1` on another should not be distinguished from the agent that just produces `X=1`.

This leads to the notion that two agents are considered operationally indistinguishable if any contraint c entailed by *each* branch of one is also entailed by each branch of the other, and vice versa. Indeed, this corresponds to the reading of $\vee$ as disjunction: $A \vee B$ entails c just in case A entails c and B entails c.

[31] develops the mathematical foundation of such languages. As in the determinate case, it turns out to be adequate to record for a process its *resting*

points: those stores in which the process is unable to add any more information (on at least *one* of its branches). Again, the underlying mathematical reason is that processes can be taken to denote certain kinds of closure operators, namely *non-deterministic*, *linear* closure operators.[9] Such operators can also be represented by the set of their fixed-points; but now one is not guaranteed that there is a *minimal* fixed point above every input. Rather the result of the computation is taken to be the set of minimal elements above the input.

This line of development is still unfinished however. Clearly, other treatments of disjunction are possible, and should be investigated.

[9] *Non-deterministic* because the operators are functions over a powerdomain (the Smyth powerdomain), since a process is capable of producing more than one result from a single store. *Linear* because the operators do not share any information between stores: the set of (minimal) results that a process can produce from the set of stores $s \cup t$ is the union of the results it can produce on s and t.

4 Indeterminate concurrent constraint programming

4.1 Syntax and Operational semantics

There are several distinct ways of expanding the syntax of agents to introduce indeterminacy in linear concurrent constraint programming. Since our primary task in this paper is to review the state of concurrent logic programming, we shall adopt an extension that most faithfully reflects the practice of concurrent logic programming. Essentially we introduce the notion of guarded choice:

$$(Agents) \quad A \quad ::= \quad \&_{i \in I} c_i \multimap A_i \quad \text{– Guarded choice} \tag{12}$$

In concrete syntax, we will prefer to use the guarded choice box $\Box$ rather than $\&$.

This gives rise to the reduction rule:

$$(\Gamma, C, \&_{i \in I} c_i \multimap A_i) \longrightarrow (\Gamma, C, A_j) \quad C \vdash c_j, j \in I \tag{13}$$

The notions of reduction sequences, and termination remains the same as before.

4.2 Logical semantics

We can now state the basic results connecting the logical interpretation to the operational execution of indeterminate programs. They are identical to the results in Section 3.2.

Lemma 4.1 *Let A be a linear formula, with free variables contained in the free variables of Γ. If $\Gamma \longrightarrow \Gamma'$, then $\Gamma \vdash A$ whenever $\Gamma' \vdash A$.*

Theorem 4.2 (Soundness of logical interpration) *Let d be a constraint with its free variables contained in those of Γ. If $\Gamma \longrightarrow^{\star} (\Gamma', C)$, and $C \vdash d$, then $\Gamma \vdash d \otimes \top$.*

Theorem 4.3 (Completeness of logical interpretation) *If $\Gamma \vdash d \otimes \top$, (with free variables of d contained in those of Γ, then there is a configuration Γ' such that $\Gamma \longrightarrow \Gamma'$ and $\Gamma' \vdash d \otimes \top$.*

4.3 Denotational Semantics

The denotation of an indeterminate process is only somewhat more complex than in the determinate case: rather than storing just the resting points of a process, we must also associate with each resting point the "path" followed by the process in reaching that point. Such a path is also called a *failure*. One of the primary contributions of [54] was to develop a simple treatment of failures via the notion of trace operators (see below).

However, in a fashion well-known from work on the semantics of other indeterminate formalisms, there are still a variety of choices for the appropriate mathematical domain of processes, depending on what one wants to consider observable.

The simplest choice, perhaps, is to observe for a process just the set of all stores that it can lead to, that is, stores it "deadlocks" in (that is, is unable to progress in, unless the environment intervenes to upgrade the store). Such a semantics would be useful, for example, in proving safety (or "partial correctness") properties for the programs: one would be able to show that regardless of the store the process ends up in, some given property holds. One can obtain a fully abstract semantics for this notion of observation by associating with a process just the set of all failures (resting points, together with the paths taken to reach that point) generable by the process. Of importance is the fact that the divergent or undefined process (needed for the base case of the semantic recursion defining the semantics of recursive processes) is just the empty set. This is because there is no store that one can observe such a process quiescing in. Indeed, a consequence of this notion of observation is that a process $\mathtt{c} \rightarrow \mathtt{A}$ is identified with the process $(\mathtt{c} \rightarrow \mathtt{div}) \square (\mathtt{c} \rightarrow \mathtt{A})$, which can non-deterministically choose to diverge. (Here `div` is just some process that can engage in an infinite execution sequence without producing any constraint or demanding any constraint from its environment.)

A minor modification of this treatment is to observe for a process the set of stores it quiesces in, together with a bit indicating whether this store was *terminal*, that is, the process had reached a stage where it would be unable to add any more information even if the environment were to upgrade the information content of the store. For example, `X=1` is a terminal resting point of the agent `X=1`, but not of the agent $(\mathtt{X} = \mathtt{1}, \mathtt{Y} = \mathtt{2}) \rightarrow \mathtt{Z} = \mathtt{3}$. With such a notion of observation we can distinguish between "deadlocked" programs (programs that have quiesced but can continue if the environment were to offer more information), and "terminated" programs. It is straightforward to develop a compositional semantics which is fully-abstract for this notion of observation.

The biggest drawback of the above treatment, however, is that it is not useful for reasoning about *liveness* properties of a program, that is for establishing that a process *must* be able to produce some constraints, regardless of the indeterminate choices that the underlying scheduler may make. For example, we know that the process

$$(\mathtt{X} = \mathtt{1} \rightarrow (\mathtt{Y} = \mathtt{1},\ \mathtt{Z} = \mathtt{2})) \square (\mathtt{R} = \mathtt{1} \rightarrow (\mathtt{Y} = \mathtt{1},\ \mathtt{S} = \mathtt{2})).$$

must produce the constraint `Y=1` on any store stronger than either `X=1` or `R=1`.

The main difficulty is treating non-termination or *divergence*. This is handled in [54] by regarding a divergent process as "catastrophic", in the style of [47,27]. Such a process is modelled as capable of exhibiting any failure whatsoever. Thus, the process $(\mathtt{c} \rightarrow \mathtt{div}) \Box (\mathtt{c} \rightarrow \mathtt{A})$ is identified with $\mathtt{c} \rightarrow \mathtt{div}$: a process that in store `c` can behave completely unpredictably (and hence of which no total correctness assertions can be made).

The set of all processes is identified by presenting some naturally motivated closure conditions on sets of failures, following [33]. (For example, one of the major conditions ensures that processes are finitely nondeterminate.) The resulting notion of a process is intuitive and easy to understand. Combinators corresponding to ask, tell, nondeterminate (dependent) choice, parallel composition and hiding are defined. A simple operational semantics is given, and a full correspondence with the denotational semantics is established.

A very simple relationship was also established between the indeterminate and determinate semantics, showing that the two semantics are essentially identical for determinate programs over finitary constraint systems.

In summary, [54] presents a simple model for the `cc` languages that is fully able to account for nondeterminism and divergence, thus permitting the use of this model in reasoning about liveness properties of programs. The model is also shown to be fully consistent with the intuitive operational semantics of `cc` programs.

Trace operators. A central issue in our semantic treatment is the representation of a failure. A failure could be represented as a sequence of ask/tell annotated constraints. Such a strategy is followed in earlier work on related languages, for example in [48,38,19], [21], [20] etc. However, ask-tell sequences are far too concrete in this setting. They store too much information which must then be abstracted from (usually via complex closure conditions) since they encode the precise path followed by a process to arrive at a resting point. Furthermore, the definition of various combinators becomes rather cumbersome because of the need to "propagate" information down the trace. Instead, we chose to represent observations via a certain kind of closure operators. To capture various operational notions of interest, we introduce the concept of *trace operators*, and *bounded trace operators*.

Intuitively, a trace operator corresponds to the closure operators generated by "straight line" (non-branching) ask-and-tell `cc` programs. Mathematically, their characterizing property is that they are *invertible*: to every trace operator f there corresponds another trace operator f^{-1} such that the only resting point of $f \wedge f^{-1}$ is `false` (the top of the lattice). Intuitively, f^{-1} is obtained from f by taking the characteristic ask-and-tell sequence of f and "inverting" it, making every ask a tell, and every tell an ask. Thus, on any input, the conjunction of the two processes is able to tell exactly what the other asks, and thus proceed hand-over-hand, so to speak, all the way to `false`. Hence `false` is the only resting point for their conjunction. For instance, the inverse for the trace operator

$$(\mathtt{W} = \mathtt{1}) \;\rightarrow\; (\mathtt{X} = \mathtt{2} \wedge (\mathtt{Y} = \mathtt{3} \;\rightarrow\; \mathtt{Z} = \mathtt{4}))$$

is

$$(\mathtt{W} = \mathtt{1}) \wedge (\mathtt{X} = \mathtt{2} \;\rightarrow\; (\mathtt{Y} = \mathtt{3} \wedge (\mathtt{Z} = \mathtt{4} \;\rightarrow\; \mathtt{false})))$$

A *bounded* trace operator is a trace operator that can stop short of **false**; that is, it is a trace operator on a sublattice obtained by considering all the elements below a certain element in the lattice.

"Assume/Tell" sequences. A related line of attack has been independently pursued by Frank De Boer, Catuscia Palamidessi and their colleagues [11,14]. In [16,17], the authors introduced the idea that it was possible to give a semantics to the asynchronous concurrent logic programming languages in a style familiar from the semantics of indeterminate data-flow languages. With a process were associated the set of its *assume-tell* sequences: a sequence $\mathtt{ask(c_1).tell(c_2).ask(c_3)\ldots tell(c_n)}$ is associated with a process if when initiated in store $\mathtt{c_1}$, it can upgrade it to store $\mathtt{c_2}$, and then when initiated in $\mathtt{c_3}$ it can upgrade it to c_4 and so on, until when initiated in $\mathtt{c_{n-1}}$ it can upgrade it to c_n and then quiesce. The denotation of a process is then taken to a set of such sequences closed under some conditions which reflect the asynchronous nature of communication, and the lack of granularity of the Eventual Tell operation.[10]

The resulting model are very similar in nature to the trace-operator based models discussed above — indeed trace operators are just an abstract counterpart of such assume/tell sequences. (A significant technical difference is that the authors do not handle the interaction between divergence and non-determinism, so that their semantics is not adequate for dealing with liveness properties of processes.) The basic point made by the authors — that for asychronous languages the "branching structure" of the refusal set semantics of [5] is not needed — is indeed valid, and is further corroborated by the work of [33]. Indeed, deBoer, Palamidessi and their colleagues have gone on to demonstrate the generality of their approach by applying it to other asynchronous languages in which communication is not "extensive" [14].

[12] discusses an equational axiomatization of the congruence on cc processes induced by the above semantics. This is very valuable for it serves as the foundation on which to build a "work-bench" that can be used to construct and reason about programs. A similar axiomatization for the determinate language is provided in [54].

[10] In a language without Atomic Tell operations, it is not possible to distinguish between the agent $tell(c_1), tell(c_2)$ and the agent $tell(c)$, where c is logically equivalent to the conjunction of c_1 and c_2. Alternatively, each communication $tell(c)$ can be thought of as happening in arbitrarily many, finite "pieces".

Bisimulation semantics. [53] develops a different approach to the semantics of CCP. Following the operationally motivated work of Milner (and Park) [42] (see [44] for an uptodate account), we introduced the notion of *reactive equivalence* of cc programs.

Let the "environment" in which an agent is executed be an abstraction of the rest of the computational system—including concurrently executing agents—with which the agent must interact to complete its computation. In order to interact, an agent must have certain variables—the *visible variables*—in common with its environment. A basic action is said to be a *visible action* if the agent's capability to engage in that action may be influenced by the environment. Thus a visible tell action is an action that imposes "new" constraints (i.e., constraints which are not yet known to have been imposed) on the visible variables, and a visible ask action is one which checks whether some new constraints have been imposed on the visible variables. Note that a visible action is not something that is "directly seen" by another agent—rather it is a step in the agent's own computation which could have been influenced by the action of some other agent. Such circumspection is necessary in understanding this concept because communication in cc languages is very indirect, and hence the concepts developed in the CCS/CSP framework—in which an event is observable if the environment can synchronously engage in it–are not directly relevant.

Unrolling the possible visible actions a process engages yields the *c-tree* (analogous to the "synchronization tree" for CCS) for the process. Two processes are considered *reactively equivalent* if, roughly, any visible action that one can do can be matched by the other in such a way that both processes end up in a state in which they remain reactively equal.

Note that the notion of visible action takes into account that telling (or asking) a constraint c is a silent action if the store is already strong enough to entail c: thus the agent $(X = 1) \rightarrow (X = 1 \wedge Y = 2)$ is reactively equivalent to the agent $(X = 1) \rightarrow Y = 2$. This highlights that the notion of communication in the cc languages is very different from that in CCS or CSP: in those systems each action is taken as atomic and does not interfere with any other actions the system may have taken in the past.

I once believed (in line with traditional dogma in concurrency theory) that reactive equivalence was the "finest" congruence that one could define on indeterminate cc agents. However, the emergence of linear logic has made it possible to treat indeterminacy on a logical footing, and it now seems plausible that there is an even finer congruence than bisimulation, namely linear logic equivalence.

Graph Grammars and True concurrency semantics [46] develops a "true concurrency" semantics for CCP based on the notion of *graph grammars*. The state of the computation is described by a graph, with variables as nodes, and constraint tokens and agents as hyperarcs. *Both* the inference rules in the underlying constraint system, and (the user-specified) agents are expressed as *graph productions* which match subgraphs to other subgraphs, possibly introducing new nodes in the process, and deleting portions of the matched subgraph. A computation is then just a finite or infinite sequence of such graph rewriting steps. From such a computation, it is shown how to construct an *occurrence net*, a particularly simple kind of Petri-net which can express "causality" information. The move from occurrence-nets to partial orders is standard, though it depends on what is desired to be observed of the computation.

An attractive feature of this approach is that the "productions" implementing the underlying constraint system are treated on an equal footing with user-defined agents. Indeed, the constraint system itself can be seen as a closure operator on sets of tokens, ordered by the is-a-subset-of relation.

What remains unclear is the connection between the closure operator semantics (which abstracts from execution sequences), and the partial ordering semantics generated from an occurrence net. More work is needed to resolve this issue. Also unclear is the precise advantage gained by the more general true concurrency approach over the interleaving approach. Perhaps this will become clearer when one considers "refinable" basic actions (e.g., ask operations that themselves involve some computation, i.e., the so-called "deep guards", see [51, Chapter 8]).

Based in part on this work, the authors and their colleagues have gone on to describe in [9] an abstract machine, CHARM (for Concurrency and Hiding in an Abstract Rewriting Machine). Its distinguishing feature is the notion of partitioning the state of a process into a *global* (externally visible) and a local (hidden) part. Transitions preserve the global parts they match on; intuitively the global parts matched represent the structure of the underlying graph that is "asked for" (tested), but not consumed. CHARM is intended as a "unifying framework" for concurrency, since it is able to formalise in the same conceptual framework, the classical approach to graph grammars as well as the CCP paradigm; and its authors expect it will also be useful in formalizing the basic concepts in other concurrent paradigms such as CCS and CSP.

As we shall see below, the connection with graph grammars is particularly appropriate when one considers a visual syntax for (at least) the cc/Herbrand programming languages.

Future work. Among various aspects of concurrent constraint programming, semantic foundations seem to be the most well-developed. Perhaps the most glaring gap is that a comprehensive, systematic, theory of testing, with more powerful tests tied to finer congruences, has not yet been developed. Also, I believe that with the advent of linear logic, it may be possible to tie the notion of testing to the notion of recording as the denotation of a process a well-chosen *subset* of the logical formulas entailed by the process (viewed as a logical formula). If this works out, it will be a very beautiful way of exposing the logical structure of the programming language.

Also, I would hope that the work can serve as the foundation for the development of practical tools to help in partial evaluation, program transformation, and static analysis of programs.

4.3.1 Related work

The basic concepts of concurrent constraint programming were introduced in [49,50]. Subseqently, we developed an operational semantics and a bisimulation semantics in [53]. This line of research extends and subsumes (for all intents and purposes) earlier work on the semantics of concurrent logic programming languages.[11]

The power of the "logical variable" has been amply recognized in recent years, and numerous authors—too numerous for us to survey their work in this extended abstract— have investigated the combination of functional and logic programming languages [40,18,3,30]. However, no account of the basic paradigm comparable with Kahn's original account in simplicity and generality has been forthcoming.

Perhaps the most noteworthy in this context is the work of [30], which discusses a subset of the functional programming language *Id Noveau* with "logical" arrays [3]. Their semantic treatment also identifies a program as a closure operator. However, their treatment is specialized to the particular kinds of arrays they were dealing with; our work is parametrized by a very general notion of constraint system. Further, they do not explicitly discuss the hiding and the Ask operations; though their "Tell" operation does implicit asks. We also present an axiomatization for the model. Their operational model includes an explicit discussion of how the constraints are resolved and represented, we achieve considerable simplicity by abstracting away the details of the constraint resolution process.

The characterization of concurrency via intersection of sets of fixed points has also been elaborated by Hoare in a recent paper [26], in a setting related to Unity [6]. Our paper develops that viewpoint, and presents a characterization of other combinators as well, in the same style. We believe that the concurrent constraint programming languages are the natural setting for the development of a theory of computing with closure operators.

The semantics of nondeterminate concurrent constraint programming languages is becoming an active area [53,20,19,10]. However none of this work explores the very simple characterizations possible for the determinate languages, dealing instead with the representation of processes as various sorts of trees or sets of i/o annotated sequences of constraints. The notion of determinacy was

[11]The semantics presented in this paper does not account for the language Concurrent Prolog studied in [21]; we do not, however, view this as a defect in our approach. Indeed, researchers working with Concurrent Prolog have moved to the Ask-and-Tell cc framework [35,20].

studied in the set-up of logic programming languages in [41], but no characterizations of the kind we provide here were given.

[20] does not consider most of the combinators we treat here, besides having a complicated treatment of ask/tell sequences. Neither does it treat recursion.

In work to appear, deBoer and Palamidessi [10,13] propose a model which is similar to ours in many respects. In particular, they have also recognized that it is sufficient to take sequences of ask/tell actions in order to get a model for the `cc` languages. However, their treatment ignores recursion. Much of the sophistication of the present model lies in the way finite nondeterminate axioms need to be introduced in order to correctly model divergence. Their model in [13] is not compositional with respect to the choice operator.

The treatment described above follows closely Mark Josephs' treatment of receptive processes in [33]. A receptive process can always accept any input from the environment, but may produce different (or no) output depending on the input. Josephs gives a set of axioms for reactive processes that turn out to be more or less what is needed in order to develop a model for the `cc` languages as well. The primary differences lie in the nature of communication, and in the combinators treated. Josephs' theory treats communication in the usual CCS/CSP style as the exchange of uninterpreted tokens with the environment. The constraint-based nature of the `cc` languages imposes additional structure which must be considered. However, as this paper demonstrates, it is possible to adapt his basic model to the `cc` setup without major reworking, which should confirm the essential robustness of his conceptualization of asynchronous systems.

4.4 Logical Semantics

4.5 Programming idioms

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).
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.

5 Programming languages

The basic scheme described above captures the semantics of Flat GHC. Flat Parlog is also essentially identical.

5.1 Atomic publication

5.2 Read-only annotations

New definition in Flat Concurrent Prolog. Based on an attractive but flawed idea.

5.3 Deep guards

5.4 Non-determinism

5.5 Failure-free langauges

In the context of distributed computation, a fundamental problem with cc languages (the *multiple teller* problem) is that a computation may have multiple producers of partial information for the same set of variables. Unless these producers are coordinated in some way with respect to each other, they may produce conflicting constraints on the same variable, thus causing the system to crash. For example, one agent may constrain a variable X to be equated to 3 and another may constrain it to be 4.

Languages such as Herbrand used the notion of atomic publication to circumvent the problem: An agent must augment the constraints in the store atomically, failing if the store is incompatible with the new information. However, this notion is far too heavy-weight to serve as a foundational communication mechanism in a distributed setting Other cc/Herbrand languages such as Parlog, FGHC and Strand deal with the multiple teller problem by ignoring it! They trust that the programmer will organize computations in such a way that the computations will never generate conflicting constraints at run-time. If this trust is violated then the entire computation will abort with a logical error.

These motivations led to the design of Janus [119]. The fundamental intuition is that mechanisms should be devised such that computation is not forced to abort at run-time because the store is rendered inconsistent. Second, the communication mechanism should naturally reflect the notion of asynchronous message passing, to be compatible with basic distributed computation concerns. That is, it should be possible for agents at a site to progress by publishing information without having to coordinate with agents at other sites. (This desideratum is even more important if it is desired that computations in other languages, built using different abstractions, should be able to *inter-operate* smoothly with computations in these languages.)

Finally, if the languages are to be widely used as procedural languages there should at least be the potential for a performance-efficient implementation competitive with that of more conventional distributed assignment-based languages. Even more strongly, we believe that uni-processor implementations of such languages should be comparable in performance with the implementations of sequential, imperative languages.

Janus attacks this problem on two fronts. First, we identified a constraint system, suitably rich for procedural programming, which possesses certain "independence" properties and naturally supports many-to-one communication. Suppose `X` and `Y` are two variables unconstrained in the current store, and an agent decides to post the (consistent) constraint `X=s`, for some term `s` possibly containing `Y`. Then, intuitively, for the constraint system to be implementable well in a distributed setting it should not be possible to construe this imposition as a constraint on `Y`. That is, more generally, if c_1 is a constraint of the form $\wedge_{i=1}^{n} X_i = s_i$ and c_2 of the form $\wedge_{j=1}^{m} Y_j = t_j$, where the X_i and Y_j are distinct, then $c_1 \wedge c_2$ should be consistent iff c_1 and c_2 are consistent separately. The Herbrand constraint system does not satisfy this property: if the constraint `X=f(Y)` is imposed, then this restricts `Y`; for example, the constraint `Y=X` cannot be imposed. Part of our task was to design a constraint system, $\mathcal{J}$, closely related to Herbrand, which satisfies such properties. Essentially, the constraint system is defined over infinite trees [8]; but matters are complicated due to the necessity of introducing "hyper-multisets" in order to cater for many-to-one communication. In designing $\mathcal{J}$ our paramount concern had been that it should be efficiently implementable in a distributed setting, expressive enough to allow straight-forward and elegant formulation of standard algorithms over standard data-structures such as arrays and lists, and should be such that with an understanding of some basic representation decisions made by an implementation, a programmer can fairly accurately predict the cost of executiion of various programs.

Second, we have exploited the idea of *syntactic control of interference*. Compile-time checkable restrictions are introduced which ensure that if the current store is consistent and a goal reduces via a program clause, then the resultant store will be guaranteed to be consistent, *without having to check for consistency*. This is achieved by exploiting a remarkably simple idea that I first stumbled across in 1987 when studying the notion of broadcast-free communication.

Say a clause is binary if any variable occurs free at most twice in the clause. In such a clause, a variable may be considered as a "point-to-point" communication channel because messages generated on it from one occurrence only need to travel to one other occurrence. Now, observe that the resolvent of two binary clauses is also binary! ("Two" is a magic number here — from three one may generate as many occurrences as one wants.) This means that if the source program clauses satisfy this condition, then any configuration generated at run-time will also satisfy this condition. All that remains then is to (a) devise a scheme which ensures that exactly one of these occurrences (the *teller*) can be used to impose a constraint on the variable — the other occurrence (the *asker*) is then used to "receive" information (b) ensure that in this scheme "tell-rights" can be communicated on channels, and (c) determine if this scheme is powerful enough to express the characteristic idioms of concurrent logic programming. All three goals were achieved in [119], adapting ideas from [93].

A major opportunity offered by Janus is that of compile-time garabage-collection. Because any variable has at most two occurrences, once one occurrence of a variable receives a term, it "knows" that there are no other references outstanding to that term (the other occurrence of the variable has been "used up" in binding the variable to the term). Therefore, a compiler can generate code knowing that whenever a data-structure in an atom has been successfully matched, the data-structure is garbage and can be recycled. This is discussed in more detail in the implementation section below.

6 Implementation

Efficient implementation of cc languages, as that of more traditional languages, hinges on low memory usage, compile-time code optimization, and low-overhead runtime management of concurrency. By keeping the program's working set small, locality of the machine's memory hierarchy can be best exploited, reducing expensive faults farther from the CPU. The storage model selected, e.g., stack or heap, is a critical design decision here. Compile-time code optimization also centers around the memory hierarchy: efficient utilization of the available machine registers and cache. This involves avoiding redundant computation (e.g., by strength reduction), which also saves CPU cycles. Finally, the runtime overheads of concurrent task management must be significantly lower than computation within tasks. This is often noted in terms of the communication to computation ratio, assuming that the primary action of task management is transmission of messages between processors.

Implementations to date of cc languages were targeted to the broad categories of uniprocessor, shared-memory multiprocessor, and distributed-memory multiprocessor hosts. Almost all of the implementations use a heap-based storage model wherein tasks are allocated in an *ad hoc* fashion from either global or local storage pools. Stack-based implementations are more compatible with uniprocessor hosts; however, task suspension must still be implemented, perhaps in a manner similar to implementations of the *freeze* primative in certain Prologs. In the long-run, although not seen yet, implementations for multiprocessor hosts must also move to stack-based models, perhaps adopting ideas from partitioning of threads in dataflow languages [**??TAM??**,138]. We discuss these issues further in Section **????** on future work. In this section we review the main themes and efforts in the implementation of cc languages over the past decade. We begin with a brief historical overview.

Historical review. Some interesting comparative work done at the University of Edinburgh by R. Trehan [139] and H. Pinto [118] summarized the experiences of programming and interpreting these languages in the "early days." These concurrent languages could be differentiated primarily by their synchronization mechanisms and how they managed multiple local environments. There are various other attributes, such as granularity control and goal scheduling, unification, etc., that affect implementation complexity and efficiency. We discuss these in depth in the following sections.

Trehan and Pinto's studies focused on interpretation, whereas further evolution of implementation efforts led to compilation and hardware support. The first abstract machine designs for this family of languages were the WAM-like FCP machine (Emu) by A. Houri [94,127], Sequential Parlog machine (SPM) by S. Gregory [90], and the KL1 machine by Y. Kimura [99]. These systems represent the first-generation compiled implementations of concurrent logic languages, evolving into more sophisticated systems. The sequential FCP machine was refined first by S. Taylor into a hypercube system [132] and by S. Kliger into a RISC-based abstract machine and optimizing compiler [102]. The SPM led to J. Crammond's Abstract Machine (JAM), the first parallel Parlog implementation [80], and the PPM [72]. The KL1 machine was implemented on shared-memory machines as Panda [121] and evolved into the abstract machine shared among the "parallel inference machines" (PIMs). A hybridization of a few of these projects was I. Foster and S. Taylor's Flat Parlog machine [85] leading to the Strand Abstract Machine (SAM) [86], ported to several types of multiprocessors. These systems represent the second-generation parallel implementations, the first comparative study of which was by Foster and Taylor [85]. The community is now beginning the construction of third-generation *optimized* compiler-based, portable systems, e.g., the `jc` Janus system [91], Monaco [135], and a portable KL1 system [73].[12]

Specialized hardware efforts were concentrated mainly at ICOT with the decade-long FGCS project and their aim of building PIMs. The personal inference machines (PSI-I,II,III) [131,111] were followed by mockup PIMs (Multi-PSI-V1/2, built of PSI-IIs), and finally PIM/{c,i,k,m,p} [109,122,67,112,105]. The main efforts, PIM/{m,p}, are large multiprocessors (2^8–2^9 processors) based on specialized hardware for direct execution of KL1. Other notable hardware implementation efforts include the Carmel microprocessors [92] and a related microprocessor proposed by Alkalaj [64]. A full analysis of hardware issues in concurrent logic language implementations is beyond the scope of this article (see Tick [134] for instance), although we do correlate the instruction set designs of the software and hardware oriented implementations in Section 6.8.

Originating around 1988, a related family of implementations has been developing around the Andorra model of computation, specifically Andorra I [76,77], Andorra Kernal Language (AKL) [98], Pandora [68,69], and ANDOR-II [130]. We cannot do justice to these implementations in this paper, as they are rapidly evolving.

6.1 Principles and Trends

Efficient implementation of concurrent logic programs requires strong foundations in several areas. As in any parallel system, task[13] switching and task creation are the primitive operations that must be made fast. Furthermore, as in any computational system, task invocation, variable binding, and memory reclamation must also be made fast. For concurrent logic programs, task switching means suspending one task and substituting (resuming) another; task creation means building a body goal task from its parent's arguments and perhaps spawning it on a remote processor. Task invocation is extended here to include the action of executing a goal to its commit point, i.e., performing the clause tries

[12] Interesting comparisons of the execution performance of many of these first, second and third generation systems can be found in Taylor [132] and Tick [135,136].

[13] We use the words *task*, *process*, and *goal* interchangeably.

needed to commit, suspend or fail.[14] Variable binding incurs added overheads to guarantee atomicity (i.e., locking around the update to avoid races among competing writers). Not only is fast memory reclamation critical, but moreover so is efficient use of memory in the first place, since the single-assignment nature of the languages can be quite profligate in touching memory.

By far the most complex implementation aspect of these basic operations is task switching and task invocation because of language synchronization semantics that require implicit synchronization on potentially incoming procedure arguments. This places a burden on the compiler and generally bloats procedure invocations with respect to sequential languages and implementations. The various nuances of language semantics, e.g., deep or flat guards, atomic or nonatomic tell unification, impact implementation efficiency.

Orthogonal to these primitive operations are intelligent task management policies that are desirable: balanced load, balanced granularity, and fair scheduling. These concepts are not unique to concurrent logic programs, and are required independent of how fast the primitive operations can be made. Looking at underlying multiprocessor hosts, we have an additional requirement to achieve full efficiency: latencies must be hidden. Memory latency in distributed multiprocessors is the major problem to dealt with. As Arvind showed [65], hiding latency effectively is directly traded-off against switching tasks quickly. We shall see (Section 6.4) that current concurrent logic programming systems can hide latency, but only within limits, and certainly overly-complex languages features cannot be effectively hidden.

The past ten years have seen a trend towards *devolution* of logic programming languages driven by the practical need to build fast implementations. The most drastic step was the definition of committed-choice languages that did not backtrack, enabling the first pseudo-parallel interpreters to be built. The next devolutionary step was from deep to flat guards, and moving from synchronizing on dynamic read-only variables to synchronizing on statically-declared arguments, enabling the first efficient implementations to be built. Next were restrictions placed on how variables could be bound: Strand [86] abolished output unification in favor of assignment, similar to moded FGHC [142] which constrains a logical variable to have a single producer. More strict, Doc [93], $\mathcal{A}'\mathcal{UM}$ [145], and Janus [119] constrain a logical variable to have at most a single producer and single consumer. These simplifications facilitate compile-time analysis and optimization of memory usage.

The progressions are further described in subsequent sections. The key point is that languages are refined by reaching an equilibrium between what application writers *demand* and what implementors *supply*. There is not yet full agreement as to where this equilibrium point is for concurrent logic programs,

[14] A concurrent logic program task is like a *thread* in threaded architectures. The task invocation creates a main thread which may split into multiple threads during guard evaluation, all synchronizing at commit, leading to a single clause body thread. The body calls spawn new threads, and so on.

and we think it will be most strongly influenced by fast, portable, and parallel implementations.

6.2 Synchronization

Concurrent logic programs synchronize on logical variables, similar to how non-strict dataflow languages use I-structures [66,106]. If a required input variable is unbound upon procedure invocation, the corresponding clause cannot commit. Furthermore, if no clause defining the procedure can commit and not all clauses fail, it implies some required input variable(s) have not been delivered, and the task must be suspended. Concurrently, if any of these required input variables are bound, the task must be resumed.

Input matching (passive unification) is transformed by compilation into instruction sequences that make matching efficient in general. The ability to synchronize on variables requires temporarily binding certain unbound logical variables to a suspended task to enable subsequent resumption. The efficiency of this infrastructure is the main factor in synchronization performance.[15]

The FCP, JAM and KL1 machine architectures all use similar methods of "hooked" variables, i.e., assigning indirect pointers from suspended variables to process structures [94]. Indirection is required to allow both multiple variables to synchronize the same task, and multiple tasks to be synchronized by the same variable. Variables are infrequent data types: Imai and Tick [96] measured 1–15% of dynamic objects are variables across a KL1 benchmark suite. To our knowledge, no one has measured the prevalence of hooked variables, and the characteristics of those hooks. It is a widely-held belief that hooks are quite simple in structure and rare in frequency. [16] Thus JAM Parlog [80] and Strand [86] allow goals to be hooked to only one variable, thereby obviating the complex bookkeeping structures needed for the general case. JAM exploits shared memory to implement a "hybrid" suspension list to gain this efficiency. Strand initiates all suspensions as if they are single-variable type, and if this most frequent case is violated, a queue of exceptionally suspended goals is used.

An orthogonal issue is how to specify the input variables upon which to synchronize. The most common method is "procedure level" representation wherein synchronizing variables are syntactically specified (explicitly as in Parlog or implicitly as in GHC) on a per clause basis. Alternatively synchronization at a "data level" representation specifies synchronizing variables, e.g., "read-only" variables in CP. The latter method has gone out of favor because, although it

[15] Early systems did not attempt to statically analyze logical variables, e.g., to determine if a variable can possibly be hooked, and if not, how to generate more efficient code for the ask tests. Recent compilers, e.g., [102,144], claim to do global static analysis to determine this and other information.

[16] The former assertion is more strongly supported than the latter — "object-oriented" programs can create many suspensions, as discussed in the remainder of this section. The two such programs measured by Imai and Tick produced far more variables than the other benchmarks.

facilitates certain sophisticated systems programming techniques, it complicates dereferencing and unification, and frustrates static analysis.

The elegant programming techniques it enables are rarely used in applications programming [124], yet the cost of implementation is felt throughout the design [85]. Foster and Taylor [85] measured (for small benchmarks executing on a sequential workstation) that trailing needed to support atomic unification (discussed further in Section 6.5) in FCP caused a 5% degradation in performance compared to flat Parlog (without atomic unification). For programs with suspension ratios (# suspensions/# reductions) of 19–56%, additional degradation of 4–8% was observed, hypothesized as other overheads associated with read-only variables (since flat Parlog and FCP were calibrated except for that).

Another implementation issue is the actual control flow of checking the synchronizing variables (discussed at length in Section 6.3). It was originally believed that parallel execution of the clause tries was beneficial because it implied faster invocation. However, if deep guards are permitted, then parallel clause tries require the ability to sustain multiple environments and incur most of the problems associated with OR-parallel management of bindings under search for a single solution. Furthermore, with flat guards, the little amount of work within the clause tries may not justify the overhead of executing them in parallel. Crammond showed consistently negative speedups (3.8% to −32%) for small, flat-guarded benchmarks on JAM Parlog on a shared-memory multiprocessor [78]. In fact, compilation techniques such as decision graphs [102] remove redundant computations among the clause tries, furthering the argument that parallel tries do not pay for themselves.

Sato and Goto [121] showed, for the shared-memory Panda system, that suspension induces execution overhead of 1–5% for small benchmarks, because of the necessity to redo the clause tries on resumption. Especially with decision-graph compilation techniques, it is not easy to avoid recomputation since there is more sharing among the code generated. For Panda benchmarks with low suspension ratios of 1–8%, depth-first scheduling mechanism effectively suppressed suspensions (with respect to breadth-first scheduling), but the benchmarks were quite simple. The one Panda benchmark with a high suspension ratio of 42% was not suppressed by depth-first scheduling. Taylor [132] measured suspension ratios of 0–56% on the hypercube for small programs, including an assembler. The higher ratios are due to static pragma-driven scheduling on the hypercube, compared to the dynamic scheduling on Panda. Imai and Tick [96] measured 14 medium-sized benchmarks ranging from 0.3–67% suspension ratios, with a geometric mean of 3.7%.

All these statistics taken together indicate, among other things, that a significant number of programs are *active*, i.e., process groups are spawned to implement active objects that compute and communicate until the termination of the algorithm. This object-oriented programming style causes frequent suspensions because the processes composing the active objects are normally suspended, awaking only upon receiving a message upon a stream. Yet all the parallel systems previously mentioned implement *process-oriented scheduling* wherein a goal reduction leads to the enqueuing of its body goals onto runtime work queues, with one of the goals selected for local execution (analogous to tail recursion optimization). Such a model executes active programs inefficiently.

Ueda and Morita proposed an alternative called *message-oriented scheduling* [141,142] for more efficient active program execution. The main idea is to transfer control to a stream consumer at the point when the producer sends the message. In the case where buffering can be avoided, this method of task switching to an active process has less overhead than the standard execution mechanism. Ueda and Morita implemented this method for shared-memory multiprocessors by scheduling from a global work pool. Since control is transferred immediately upon message sending, effectively independent chains of message sends are executed by the processors. Their initial performance results are extraordinarily good: naive reverse executes on a single processor at 3.3 seconds (cf., `jc` (Janus) runs at 3.9 seconds [91] and Monaco (FGHC) runs at 19.2 seconds [135]). Furthermore almost linear speedups are achieved, as well as comparable performance to optimized 'C' (the authors however do not compare the technique with process-oriented scheduling for an equivalent technology base, to determine the utility of the concept).

For example, Kliger [102] reports that a set of 27 small-to-medium size FCP(:, |) benchmarks achieved 21% geometric mean speedup due to a set of optimizations based on global analysis, including reduction of unification into assignment, in-lining arithmetic, manipulating unboxed objects, and excluding procedure clauses that can never commit for any invocation.

6.3 Guards and the Process Structure

Guards, similar in purpose to Dijkstra's *guarded commands*, were introduced to logic programming in the Relational Language [75]. They extend, from simple input matching, the expressivity of how to commit to a clause. In their most general form, guards among clauses defining the same procedure represent disjunctive processes racing to commit. Implementation difficulties occur 1) if these processes are allowed to bind variables, and 2) even if binding is outlawed, if processes are permitted to make nested calls. The former problem is indicative of "unsafe" languages, and the latter problem is indicative of languages with "deep" guards. Considering the range of complex to simple implementations, the languages fall into three basic categories: unsafe and deep (e.g., CP), safe and deep (e.g., Parlog), and safe and flat (e.g., FGHC).

Unsafe clauses may compete with one another in the sense that each may wish to make conflicting bindings to the same variables. This is implemented by restricting bindings to a local environment, for eventual exportation upon commit. Exportation can, however, conflict with concurrent bindings made nonlocally. If this happens, the clause try fails. Detecting inconsistencies is a major implementation problem in these languages — there is a choice among detection *before* commit or *after* commit. The former presents a clearer semantic model to the programmer, but is far more difficult to implement (see Section 6.5).

Deep guards effectively form a process hierarchy or tree, with local environments at each level. Local environments are needed, even if the language is safe, because incoming bindings (to local variables) must be saved across deep guard evaluation. One severe implementation problem is the management of multiple environments (one per deep-guard clause in the same procedure) if guards are evaluated concurrently. This *value access control problem* is similar to that of OR-parallel implementations of Prolog: how to efficiently ensure that only ancestor environments on a path to the root are accessible, and that all other environments are hidden.

Another implementation problem is supporting *fair* execution while descending the hierarchy, while retaining low complexity and cost. If fair execution is not guaranteed then eagerly executed guards may loop, preventing later guards from failing and freeing up the computation. Shapiro [124] states that the inability to achieve fairness at low cost motivated flat languages.[17] He cites early CP implementations (e.g., [108]) as either unfair or of "unacceptable" complexity.

JAM Parlog [80] constrains deep guards to be used only in clauses bracketed by *sequentialized clause separators* (in some languages called *otherwise* guards). Such separators prevent subsequent clauses from being tried until all previous clause tries fail. This restriction obviates concurrent evaluation of deep guards, simplifying management to that of a single local environment per procedure. Crammond [80] states that this restriction allows most of programmers' intended uses of deep guards, e.g., as if-then-else conditionals.

Parlog offers the programmer a sequentialization operator `g & b` that guarantees goal `g` executes to completion before goal `b` is executed. In JAM, the implementation views a clause as compiled above with the guard `g` and body `b`. In other words, the same mechanism used to implement sequential goal execution does double duty for deep guard execution. Deep guards need to be evaluated concurrently to avoid deadlock; however, given mode information, flat guards can often be executed in-line for efficiency. The environment necessary for carrying local bindings over a sequentialization operator are not unlike a Prolog environment.

Restricting the language to only flat (safe) guards engendered decision-graph compilation [102] because clause tries can be compiled in line without transfer of control nonlocally to other goals. A decision graph is composed of if-then-else and switch nodes which transfer local control conditionally upon a test. A graph is formed, rather than a tree, to guarantee space proportional to the number of clauses in the procedure. To ensure space linearity, a clause is propagated down one and only one branch of the graph as code is being generated. Thus

[17] This problem still exists, in a less troublesome form, for unfair flat languages when early builtin guards suspend, preventing or delaying later guard failure. This can only reduce the failure set, which is not important because failure is a catastrophic event in flat languages.

clauses ambiguous to a test are conservatively placed in a *continuation* branch, and sibling branches jump to the continuation upon failure. For a suite of 27 medium-size benchmarks, decision graphs executed 3.2 times faster on average than WAM-like compilation [102]. The code size expanded by 30% on average, with a particularly degenerate program (Salt & Mustard [133]) doubling in size. An interesting problem is how to order the graph nodes, and how to generate optimal code for the tests, conditional branches and switches, to minimize execution time [81].

A main purpose of deep guards is to perform speculative computations that can fail allowing alternative solutions to succeed. Unsafe languages enriched this paradigm allowing bindings to be made along the speculative path. Experience has shown that support of both of these operations is too expensive for the low frequency with which they are used. The devolution to flat languages is complete in the sense that almost all research groups opted to reduce language expressibility in favor of easily-implementable flat guards. In a further extreme, Fleng [114] abolished guards in an effort to streamline execution. In general, guard tests must be pulled up and evaluated at each call site. This allows the optimization wherein certain guard tests need only be evaluated at certain call sites. Global analysis is needed to produce the information required for this optimization. Although FCP has guards, a similar optimization is enabled by Kliger's method of customizing decision-graph clause tries for different call sites [102].[18]

6.4 Reading and Writing Logical Variables

The costs of reading and writing logical variables can be calculated as the *frequency* of operations required, multiplied by the *cost* of the operations. For example, reading a logical variable incurs the incremental cost of suspending the variable at the rate of suspension. In shared-memory multiprocessors, all accesses are "local," so that the relevant overheads are lock traffic on the shared bus and lock contention. Contention, i.e., multiple concurrent requests of the same lock, can be exacerbated when the host does not supply enough physical locks for all objects needing locks.[19] For small benchmarks, Sato and Goto [121] reported that locking accounted for only 1–5% performance degradation on the Sequent Balance. This conveys both the relative efficiency with which locks can be implemented with shared memory, as well as the retained significance of lock overhead. Interestingly, although most of the lock traffic they measured was for protecting bindings, most of the observed lock contention was for bookkeeping locks for scheduling and termination.

[18] Practical definitions of Fleng allow the programmer to specify guards, which are then pulled up to the call sites. Without global analysis, of the complexity required by Kliger, *all* guard tests must be pulled up to *each* site.

[19] Because this hardware attribute cannot be easily modified for a given host, studies of lock contention *vs.* lock granularity have not been performed for concurrent logic languages.

Distributed-memory multiprocessors are significantly more problematic because of the overheads incurred in reading and writing nonlocal variables. Nonlocal reading requires sending a message requesting the variable's value, and receiving a reply. Nonlocal writing requires issuing the binding — the receiver can update the variable locally (without explicit locking) and send either a success or failure acknowledgement. The incremental cost of resuming tasks hooked to the bound variable must be accounted for in a macro view of execution.

There are of course variations on both of these protocols. Taylor [132] discusses a protocol on a hypercube where nonlocal writes first request a remote lock, and upon receiving the lock, issue a remote write. He measured 61%–100% of all messages sent, for six FCP benchmarks, are nonlocal reads. The four smallest benchmarks required an arithmetic average of 99% reads. Although write frequency is seen to be very low, its amplified cost can be felt. For example, Taylor demonstrated that for the incomplete message paradigm (where nonlocal reading and writing occur with equal frequency), the main execution overhead on a multiprocessor was not the sender reading the return value, but rather the receiver locking and writing the return value. In this simple example, locking proved to be extremely expensive (performance degradation of two times on two hypercube nodes) because the latency could not be hidden.

Reducing the cost of reading is critical in distributed implementations. If a complex term is to be read nonlocally, an important design consideration is how much of the term should be eagerly transferred. Taylor also examined the affect of this *copy depth* parameter on performance. For standard paradigms such as producer-consumer and incomplete messages, performance improved significantly for initial increases in copy depth, after which no improvement was seen. The interpretation of these results is that the consumer is "brought up to speed" by increasing transfer size until the point at which it outruns the producer, after which no further improvement can be achieved. Because these are such pervasive programming techniques in concurrent logic programs, it is imperative to find ways to speed them up. Hardware support for message management (packing and unpacking, merging active messages straight into the execution pipeline) to more effectively hide latencies is one approach, similar to the goals of threaded architectures (e.g., [113]). Another idea is to reduce the number of messages sent, either by introducing new programming paradigms, or by dynamically migrating tasks and streams so that communication is local. Yoshida [145] took the latter approach in the design of $\mathcal{A}'\mathcal{UM}$, discussed in Section 6.9.

6.5 Unification

Unification is somewhat controversial because it stands out as one of the few unbounded-time operations required by logic programs compared to conventional languages. In many cases unification can be compiled into simple instructions, as was elegantly shown in the WAM [63,143]. Unification in committed-choice languages can be categorized as either *input* (also: *passive* and *ask*) or *output* (also: *active* and *tell*), reflecting the exportation of bindings. Ask unifications implement head matching either as explicitly compiled match instructions or as invocations of a fully general passive unify routine. Luckily, full passive unification is rarely executed: it occurs only when checking the equality of two incoming arguments. For example, Foster and Taylor [85] measured the execution of 153,800 matching operations and 15,300 general passive unifies (9% of total) in an Assembler benchmark written in Flat Parlog. Furthermore, if sufficient type information is inferred, general passive unifies can be reduced to simpler tests.

At the leaves of unification's recursive descent, rules for unifying primitive data types come into play. Read-only synchronization requires an extended set of rules [108,132] compared to procedure-level synchronization, potentially reducing performance.

Whereas ask unification occurs before commit, the location of tell unification varies among the languages. Unsafe languages require *atomic* tell unification wherein no output bindings are seen until commit. This means that bindings must be locally trailed, and perhaps undone upon failure. Furthermore, atomic unification can bind two variables which are inputs of the same procedure invocation. This raises the issue of whether such bindings should be eagerly acknowledged, although such an implementation causes a performance degradation of 1–29% [85]. This and trailing overheads led to the abandonment of atomic unification for implementation reasons alone. Safe languages place tell unifications after commit (called *body unification*). The implementations are thus free to perform body unifications on the fly, with failure causing procedure invocation failure.

Even body unification is complex when considering multiprocessor implementations. The main problem is to avoid potential race conditions among concurrently executing tasks. Thus any logical variable that needs to be bound must be locked first. Furthermore, if both operands are unbound, they cannot be locked in an arbitrary order under threat of deadlock with a competing task trying to bind the pair in the opposite order (very unlikely, but possible, unless mode restrictions are known, as discussed below). Some ordering must be made, e.g., exploiting the nature of a shared-memory name space.

By making the most frequent case fast, general unification on multiprocessors can be implemented efficiently. The most common case by far is binding a constant to a logical variable that does not require dereferencing. A fast stub can be constructed that tests if one operand is constant, and one is unbound and not hooked. On a shared-memory machine, the variable is then locked, bound, and unlocked. Otherwise, if the initial condition is not met, full unification is required. The two operands to be unified must be locked, dereferenced, and compared. On distributed-memory multiprocessors, the same algorithms can be naively used, with nonlocal accesses potentially at the leaves.

Because of the implementation complexity and potential execution overheads of output unification on distributed-memory multiprocessors, and the evidence that it is infrequently used in its full generality, output unification further devolved in Strand to *assignment*. Thus recursive descent is obviated, and the left-hand side of the assignment is required to be a variable. However, this does not rule out the need to test for exceptions, or hooked variables, which need to be resumed. Thus binding is still expensive compared to imperative assignment.[20] Program Composition Notation (PCN) [71] is a multi-lingual language built from Strand and 'C' that offers both *definition* (logical) and *mutable* variables. Thus safe and efficient assignment to mutable variables can be guaranteed by the programmer. Furthermore, memory usage can be reduced by destructive update of mutable variables. In a sense, PCN is the farthest devolution has progressed in concurrent logic languages.

6.6 Task Scheduling and Priority

There are various philosophies for automatic scheduling of parallel tasks. Compile-time analysis can be attempted to determine a fixed schedule mapping tasks to processors. Runtime profiling information can aid the static analysis. A radical departure is to perform all scheduling dynamically without any static aid, or a hybrid combination of static and dynamic. Another approach is to avoid automation and require the programmer to explicitly distribute tasks.

Automatic scheduling in concurrent logic programming systems is usually dynamic (e.g., JAM, Panda, Monaco) because tasks are too small and numerous to allow practical static analysis. For shared-memory multiprocessors, the main implementation issue is how to efficiently manage the goal queues. A single shared queue would eliminate the need for load balancing, but contention for this scarce resource is too costly. Splitting the queues up, one per processor, removes contention but leads to potential unbalancing. Once queues have been split, there emerges the implementation paradigm, on large-grain process systems such as UNIX, of *task farming*. Here a single UNIX process, often called a "worker," is responsible for coroutining between the execution and scheduling of goals. In *on-demand scheduling*, goals are not eagerly distributed among workers and only an idle worker searches for work, thereby minimally disturbing busy processors. Sato and Goto [121] and Crammond [79] examined variations of on-demand scheduling involving further splitting the local queue into *private* and *public* queues and allowing idle workers to steal only public work. Crammond reported that for eight medium-sized Parlog benchmarks, private/public queues offered slightly better and more consistent speedups than public-only scheduling, on the Symmetry and Butterfly II (on 16 Symmetry PEs, geometric mean efficiencies of 86% and 83%, respectively). These early studies measured multiprocessors with far slower processing elements than are available today.

[20] With compile-time freeness analysis, e.g., [142], and hookedness analysis, e.g., [144], these tests can be safely removed.

There has also been much work within the ICOT FGCS Project exploring automatic load balancing methods, e.g., [88,95,109,121,128]. The most successful experiment has been the *multi-level load balancing* (MLLB) scheme for balancing OR-parallel search programs on a distributed-memory multiprocessor [88]. The idea is to partition the available processors into groups, and allocate one distribution master per group. Slave processors within these groups request work from the master. There is a method of merging groups, and given the regular nature of OR-parallel search, this method has been shown to be quite effective, e.g., speedup of 50 on 64 processor Multi-PSI for the pentomino benchmark [88].

The drawback of MLLB is its limited application domain. Thus even ICOT resorted to explicit user-defined "pragma" in the KL1 language for remote task scheduling on distributed multiprocessors. Strand and PCN also require pragma. In these latter languages, the user is encouraged to design load-distribution management networks, called *motifs*, e.g., MLLB could be specified as such [84,87]. In PCN, motifs consist of several programming constructs implemented in the source language with libraries providing support. Simple pragma are enriched by allowing the definition of *virtual topologies*, which can be embedded within physical topologies. Topologies are collections of nodes, such as a hypercube network, implemented by process structures. User-defined tasks are mapped onto the nodes by passing the tasks as messages for meta-execution. A user program can be written to interface to a single virtual topology, which can then be automatically mapped onto whatever physical topology is offered by the hardware organization. There are several other constructs, such as *templates* and *ports*, which facilitate program creation, but do not present major implementation difficulties.

An issue related to task scheduling is task priority. Early concurrent logic languages specified that goals were required to be executed in a *fair manner*. Fairness is difficult to define in a manner that can be easily implemented. One weak definition is that all tasks which *can* execute *are* attempted at some time. This guarantees avoidance of spurious deadlock. Normally, tail-recursion optimization (TRO) is implemented wherein a selected body goal is directly executed and all others are wrapped up as goal records and enqueued. By extending the life of a thread through a selected child in this manner, efficient use of registers for argument passing can be achieved. Fair execution is emulated in a number of systems by a time-slicing technique, wherein every k reductions, TRO is replaced by enqueueing *all* body goals at the back of the queue, and switching in a goal from the front of the queue. Implementation incurs the overhead of updating a counter comparing it to k for each reduction, as well as enabling queue access from both the front and back.

The KL1 PIM systems took a different approach, discarding the notion of task execution fairness altogether. It is replaced by a goal priority scheme, wherein the scheduler makes its best effort to abide by priorities. This allows programming techniques such as speculative exploration of alternative solutions.

KL1 allows goal pragma that set priorities relative to a parent goal or a collection of goals called a *shoen*. These *logical* priorities, potentially ranging from 0 to 2^{32}, are retained in goal records, but also mapped into a smaller *physical* range for purpose of sorting. For example, if the physical range is 0 to 2^{14}, then the PIMs use an array of 2^{14} queues. The non-empty queues are linked to allow efficient dequeueing across priorities. Insertion of a goal into an empty queue requires a linear search up to the nearest non-empty neighbors to update the links. This algorithm is sufficiently simple for its microcoded implementation, although software-based implementations might be better served by balanced, priority trees.

6.7 Granularity Control

Concurrent logic programs are fine grained: Alkalaj [64] measured from "20 to several hundred single-cycle instructions" per average goal reduction. Taylor [132] measured FCP granularity on a hypercube as a ratio of reductions/messages-passed,[21] ranging from 3.5–220. It is clear that granularity is very much dependent on application and programming style, but even in the best case, granularity is still low compared to conventional approaches to parallel programming in imperative languages.

The advantage of fine-grained concurrent languages is the abundance of potential parallelism. However, the main disadvantage is that too-fine granularity can lead to excess overheads in task management. Alkalaj [64] has shown that 50% of the execution time of large FCP applications is spent on goal management for a reasonable machine execution model. His recommendation was a specialized hardware organization to support this efficiently. Such directions are promising, as echoed for instance in the hardware implementations of threaded architectures, mainly predicated on dataflow languages (e.g., [117]). Special hardware or not, it is necessary for to boost efficiency by "collecting" granularity at compile time [120,138].

Ideas along these lines were developed for logic programs by Debray *et al.* [82], King and Soper [100], and Tick and Zhong [137]. Debray's design seeks to construct, as compile time, estimators of input argument size, and formulate these estimates into granularity estimations. At runtime, a granularity estimate is evaluated for each procedure invocation, and the weight value is used to make dynamic scheduling decisions. For example, if the weight is below a threshold, a task will not be spawned because of excessive overhead.

King [100] discusses an analysis technique with no runtime component. Similar to Debray's method, granularity is modeled as a function of argument size; however, these sizes are estimated by abstract interpretation, associating argument types with a finite lattice of sizes. The result is a purely static determi-

[21] In a distributed memory multiprocessor implementation, messages would be used for communicating values down a stream from producer to consumer, for example.

nation of granularity. King uses this analysis, as an alternative to profiling for example, to drive task sequentialization (see Section 6.8).

Zhong's approach [137] attempts to remove the complexity of argument-size estimation (both at compile time and subsequent runtime evaluation costs) by introducing an abstract "iteration parameter" which is a proxy for relevant granularity information. The remainder of the scheme is similar to Debray's, with the major distinction that the estimators are easier to formulate, cheaper to evaluate, but far less accurate. Furthermore, the weights computed are relative, e.g., it can be estimated that one task is half the weight of another task, but it cannot be determined if either are below some absolute threshold weight.

The verdict is not yet in on the utility of these granularity analyses, because empirical data is sparse. Robust analyzers and larger benchmarks are needed.

6.8 Abstract Instruction Set Architectures

The Warren Abstract Machine (WAM) [63,143] had a great influence on the various concurrent logic language implementations discussed in previous sections. The important differences among the abstract machines developed for committed-choice languages are in their storage models. The primary distinction is whether the heap is based in shared memory [80,91,94,99,102,135], or distributed memory [105,112,122,132]. In general, all variables and terms are stored in a heap, and memory is reclaimed by explicit, periodic garbage collection. Goals are usually represented by heap terms that can be linked into work queues for scheduling. A goal that is suspended can "float" on the heap, to be relinked to a work queue upon binding its hooked variable.

There are several variations to this basic model to gain efficiency. Goal records can be constrained to be fixed size and queued in free lists facilitating memory reuse. Furthermore, all data structures can be partitioned onto heaps corresponding to size, each with its own free list for ease of (de)allocation. Crammond [80] split arguments away from goal records, allocating them on their own stacks to improve locality and reuse. With arguments allocated separately from goal records, goal-record locality improves, and arguments no longer need to be of fixed size if allocated in a stack-based fashion. However, this results in the creation of "holes," i.e., deallocated frames trapped below the top of stack, which can require general garbage collection if they grow too large. Crammond [78] illustrates the extent of this problem for some small benchmarks.

In addition, bookkeeping structures for evaluating deep guards and suspension management are necessary. Deep guards and sequential conjunctions require the use of environments which hold values of variables active throughout the clause try or sequential body evaluation. Practical implementations require at most one environment per invocation [80], which is deallocated upon body completion or guard failure (to be reallocated for the next clause try). In addition, a *trail* is needed for atomic tell unification wherein failure and suspension during unification must "back out" all bindings generated. Suspension management requires a suspension stack holding pointers to input arguments that are needed but are as yet unbound.

A less important distinction among the systems are their abstract instruction sets. The instruction sets of the various machines follow the general WAM model, passing arguments through dedicated registers, and having a set of additional state registers for control and storage management. The instruction sets can be broken down into similar groups. Older models use WAM-like indexing control instructions, whereas decision-graph compilers for flat languages avoid shallow backtracking and much of the required control instructions. Head matching (ask unification) is compiled with **wait** instructions that will push their corresponding argument onto the suspension stack if it is not instantiated. Tell unification is compiled into **get** instructions that will make assignments or invoke a general unifier. Finally, body goals are generated with **put** instructions for loading arguments and enqueueing goal records. As mentioned in Section 6.6, usually a form of tail recursion optimization (TRO) can be implemented by loading the arguments of one of the body goals directly into the argument registers and jumping to the goal code.

Additional instructions are needed for the (de)allocation of local environments (for non-flat languages and/or sequential conjunctions) and heap storage. Goal-management instructions are responsible for terminating a thread (in unit clauses), enqueueing a goal (creating threads), directly executing a goal (TRO, called *promoting* a thread in JAM), and initiating deep guards. Note that flat languages have threads that live very short lives, not counting promotions. Sequential conjunctions, introduced in Parlog, do not lengthen threads in practice because they are necessarily implemented with trees of local environments (see Section 6.3). This stems from the fact that within a sequential conjunction, concurrent goals may execute. If total sequentialization of a goal and all its children can be specified or derived, then these local environments can be stacked, resulting in superior space and execution time efficiency. This would be a true elongation of threads, resulting in increased performance [101,107]. Such an implementation requires a sequential call as well as stack (de)allocation instructions.

Arithmetic instructions and builtin predicates must be able to suspend if executed before commit, or be enqueued as bonefide goals if executed after commit. JAM optimizes this by checking arithmetic operator inputs in the body, and if available, executing the arithmetic in place. Otherwise, a goal is created. In general, a lazy strategy of dynamically checking for data dependencies is taken by all concurrent logic programming systems. Threaded architectures for dataflow are slightly different in that they do static analysis to generate arithmetic threads that cannot internally suspend. However, both research communities treat array accesses in a similar manner: a builtin access goal is spawned, avoiding long latencies for nonlocal access. After all, statically analyzing array indicies is very difficult [70].

A trend towards reduced abstract machine design, following the principles of RISC design, has led to instruction sets such as Carmel [92], SAM [86], the `jc` machine [91], Kliger's machine [102], and the various PIM architectures [105,112,122]. For example, Strand, FGHC, Fleng, Janus, and FCP(:, |) sufficiently simplify the execution model, obviating trailing, environments, atomic tell unification, and a process hierarchy for deep guards and sequential conjuncts. This allows these compilers to concentrate on optimizations, such as decision-graph generation, in-line arithmetic, and global register allocation. Sequential implementations offer further performance gains, obviating locking and allowing the leverage of compilation into 'C'. Debray and Tick measured a mean speedup of 2.4 comparing `jc` with Monaco for six small benchmarks [135], illustrating the potential advantages of sophisticated register allocation, and streamlined binding mechanisms.

Readers interested in concurrent logic language instruction-set design are referred to Crammond [80], Foster and Taylor [85], and Kliger [102] for the most complete expositions.

6.9 Stream Communication, Arrays and Garbage Collection

A major defect in concurrent logic languages and their implementations is inefficient use of memory. This problem is prevalent in the treatment of communication streams and data arrays, and is exacerbated in distributed-memory multiprocessors. A general, after-the-fact, solution to the problem is the construction of ever more efficient garbage collectors, about which we comment at the end of this section. We first discuss pre-emptive solutions, such as making stream communication efficient with buffers and migration.

Streams are second-class citizens in most logic programming languages. Stream communication is programmed by having a producer write messages into a difference list, the head of which is read by a consumer. To nondeterminately merge multiple streams, a chain of active *merge* processes is needed. This methodology was stressed in the original literature because it is elegant and all that is offered at the *language* level. However, defining and implementing streams in this manner is highly inefficient. First, merged streams incur extra process reductions, lengthening transmission *delay*. Second, naive stream merging can result in *unfair* data transmission. Third, if the memory cells comprising a stream and the reader of that stream are located on different processors in a distributed-memory system, then reading a value requires the overhead of sending a request message.[22]

The fairness problem can be solved with more sophisticated, dynamically-balanced merge trees [125], although this is expensive in time. The delay problem has been solved both in software and hardware. In software, a data type,

[22] Analogous to driving all over town to pick up mail at different post offices, instead of having all mail delivered directly to your house.

called a *mutual reference*, interfaces multiple writers to a single reader [126]. Writing to one of the streams will atomically write the merged output stream and update the mutual reference to point to the new output tail. This scheme, originally designed for FCP, is the essence of implementing streams as buffers in other languages also. For example, the PIMs implement mergers, in microcode, in a similar manner [97,140]. The critical difference is that the new data structure is hidden from the language definition. Furthermore, with MRB (discussed below), memory can potentially be reused when merging a new writer into the stream.

The indirection problem could be corrected by locating the buffer with the consumer (similar in intent to message-oriented scheduling, see Section 6.2); however, in most concurrent logic languages, multiple consumers are permitted, and single consumers are not recognized as such by the compiler. Global analysis might be used to determine single consumer streams, or the languages can be restricted. The latter solution is another devolutionary step (notably Janus [119] and $\mathcal{A}'\mathcal{UM}$ [145]) with a "single writer/single reader" restriction and abstract stream semantics that do not constrain implementations.

Janus defines a *bag* data type that can be used as a multiple-writer stream, with no constraint on write order. Janus also defines standard arrays; however, the restriction permits an implementation to automatically reuse array locations. Neither bags, nor reusable arrays, have yet been implemented for Janus. $\mathcal{A}'\mathcal{UM}$-90 is called a stream-based concurrent object-oriented programming language (SCOOL) by its authors [104], emphasizing the first-class citizenship of streams. Streams are implemented as buffer objects that can migrate. The migration policy moves buffers to their (unique) consumers, thereby obviating the overhead of sending read requests. This is implemented by having the producer and consumer initially communicate with **where** and **here** messages, allowing them to locate each other and begin message copying. Future messages are forwarded automatically to the new location. For a generate-&-test prime-number generator, migration achieved a speedup of 12% (on 10 Symmetry processors) compared to no migration [104]. This increased to 70% speedup on two Sparcstations connected over an Ethernet.

The general topic of garbage collection is too large to cover here, but it is important nonetheless. There are fundamentally two approaches to garbage collection: static and dynamic. Static collection requires compiler analysis to determine the guaranteed reusability of a data structure. Code can then be generated directly for memory reuse. Incremental dynamic collection involves runtime checking to determine reuseability of structures. Furthermore, dynamic garbage collections across an entire memory (local or global) are required when the previous incremental methods fail. Examples of these collection types within concurrent logic programming are abstract interpretation for local reuse [129], binary reference counting with multiple reference bits (MRB) [74,115], and several stop & copy schemes (e.g., [89,96,97,110]).

The MRB scheme [74] is a one-bit approximative reference count per data cell. If the flag is off, the cell can be reused because it is guaranteed to have a single reader. However, once the flag is set, it becomes stuck and no reuse is possible. Setting the flag requires nontrivial rules for many of the KL1 abstract instructions that manipulate memory. The advantages, however, can be significant, for example, array copying can be dynamically converted to destructive update by exploiting the MRB method. Nishida *et al.* [115] demonstrated the effectiveness of the MRB scheme for a shared-memory multiprocessor model. Depending on data cache configuration, two small benchmarks displayed bus traffic reduction from 20%–57% on 16 processors. The least beneficial result of 20% reduction clearly showed a drastic increase in cache-to-cache traffic that was indicative of reused cells being transferred between processors. Overall the method achieved significant reduction in memory-to-cache ("swap in") traffic, indicating success at improving locality by reuse. Duvvuru *et al.* [83] showed that MRB running on SUN Sparcstation had associated management overhead of 5–7% for five small benchmarks. With static analysis [129] to determine guaranteed single-referenced cells, this overhead was removed. It is clear however that array indexing can baffle such analysis, thus dynamic schemes still have utility.

The challenges of implementing efficient garbage collection schemes for concurrent logic and functional programs are similar. There are several garbage collection schemes proposed (and some prototyped) for concurrent logic languages. These efforts have concentrated on stop & copy schemes for shared-memory multiprocessors, mirroring the general sophistication of the corresponding runtime systems. Research has recently focused on 1) distributed memory garbage collection schemes that do not require barrier synchronization of all processors within the collector [103]; 2) efficient, parallel stop & copy garbage collectors for shared memory [78,96], and 3) generation-scavenging garbage collectors for reducing collection latency by interning long-lived objects [116]. For generation-scavenging schemes, a problem arises when the "old" space (the interned space) points into the "new" space. This can occur for interned logical variables and reused objects (e.g., via MRB). Methods of trailing these unsafe cells and implementing indirection tables are costly [110], but seemingly unavoidable.

7 Future Work and Conclusions

Issues that are raised: is the pure model of computation really practical? Is it really worthwhile to have to pay all that extra cost for de-referncing every variable when in reality only occassionaly will there be need to suspend. Can clever techniques be developed to hide this under the hood — or is there a basic distinction between synchronization (active) variables, and local (passive) variables (those on which you are guranteed not to suspend). Can some mecahnisms be described to allow the user to provide extra information for the implementation, without letting this information change the semantics of the program, such that the information can be used by the compiler to improve performance?

Relationship to other theories of concurrency. Recently there have been radical new ideas about concurrency. Two of particular note are the so called Chemical Abstract Machine [4], due to Boudol and Berry, and mobile processes, due to Milner and his co-workers [45]. In both of these approaches the key new ingredient is that processes can alter their interactions and are, in efffect, mobile. In our approach the interactions between processes is dynamic too in the sense that there is no predetermined set of agents that a given agent is limited to interact with. The relationships need, however, to be understood carefully. It would be particularly interesting to understand how a lambda abstraction mechanism could be incorporated into the concurrent constraint paradigm. Understanding the relationships with the work on mobile processes or Boudol's gamma calculus would be very helpful, as it is known that the latter can encode the lazy lambda calculus [43,29].

7.1 Real-time languages

7.2 Logic and state-change

7.3 Higher-order languages

Be able to pass processes as parameters. Even though this may be simulated at the level of the language, isn't particularly a nice way of doing it.

7.4 Some broad themes from the viewpoint of programming languages

Why bother to develop stand-alone programming languages.

Software architectures are more important than programming languages. The difference is one of syntax!

Architectures can be embedded in any kind of host computing framework, e.g. C++.

Model-based computing.

Acknowledgements

E. Tick was supported by an NSF Presidential Young Investigator award, with matching funds from Sequent Computer Systems Inc. We thank T. Chikayama, J. Crammond, and S. Debray for sharing their expert knowledge, greatly assisting in the completion of the article.

References

[1] The fifth generation project. volume 36, 1993.

[2] S. Abramsky and C.-H.L. Ong. Full abstraction in the lazy lambda-calculus. Technical report, Dept. of Computing, Imperial College, 1989. To appear in Information and Computation.

[3] Arvind, Rishiyur Nikhil, and Keshav K. Pingali. I-structuers: data-structures for parallel computing. *ACM Transactions on Principles of Programming Languages*, 11(4):598–632, October 1989.

[4] Gerard Boudol and G. Berry. The Chemical Abstract Machine. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 81–94. ACM, 1990.

[5] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the Association for Computing Machinery*, 31(3):560–599, July 1984.

[6] Mani Chandy and Jay Misra. *Parallel Program Design—A foundation*. Addison Wesley, 1988.

[7] K.L. Clark and S. Gregory. A relational language for parallel programming. Technical Report Res Report DOC 81/16, Imperial College, July 1981.

[8] A. Colmerauer. Equations and inequations on finite and infinite trees. In *2d International Conference on Fifth Generation Computer Systems*, pages 85–99, 1984.

[9] Andrea Corradini, Ugo Montanari, and Francesca Rossi. CHARM: Concurrency and hiding in an abstract rewriting machine. In *Proceedings of the Fifth Generation Computer Systems Conference*, 1992.

[10] F. S. de Boer and C. Palamidessi. A fully abstract model for concurrent constraint logic languages. In *Proceedings of CONCUR '90*, 1990.

[11] F. S. de Boer and C. Palamidessi. A fully abstract model for concurrent constraint programming. In *Proceedings of TAPSOFT/CAAP*, volume 493 of *Springer-Verlag Lecture Notes in Computer Science*, pages 296–319, 1991.

[12] F. S. de Boer and C. Palamidessi. A process algebra of concurrent constraint programming. In *Proceedings of the Joint International Logic Programming Conference/Symposium*, 1992. To appear.

[13] Frank de Boer and Catuscia Palamidessi. A fully abstract model for concurrent constraint programming. June 4 1990.

[14] Frank S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. The failure of failures in a paradigm for asynchronous communication. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings of CONCUR '90*, volume 458 of *Lecture Notes in Computer Science*, pages 99–114. Springer Verlag, 1990.

[15] Frank S. de Boer, Joost N. Kok, Catuscia Palamidessi, and Jan J.M.M. Rutten. *Logic Programming Languages Constraints, Functions and Objects*, chapter A paradigm for asynchronous communication and its application to concurrent constraint programming. MIT Press, 1993.

[16] Frank S. de Boer and C. Palamidessi. Concurrent logic languages: Asynchronism and language comparison. In *Proceedings of the North American Conference on Logic Programming*. M.I.T. Press, 1990.

[17] Frank S. de Boer and C. Palamidessi. On the asychronous nature of communication in concurrent logic languages: a fully abstract model based on sequences. In *Proceedings of CONCUR '90*, 1991.

[18] Doug DeGroot and Gary Lindstrom, editors. *Logic Programming: Functions, Relations and Equations.* Prentice Hall, 1986.

[19] Maurizio Gabbrielli and G. Levi. Unfolding and fixpoint semantics of concurrent constraint logic programs. Technical report, University of Pisa, 1990.

[20] Haim Gaifman, Michael J. Maher, and Ehud Shapiro. Reactive behavior semantics for concurrent constraint logic programs. In *North American Logic Programming Conference*. MIT Press, October 1989.

[21] Rob Gerth, Mike Codish, Yossi Lichtenstein, and Ehud Shapiro. A fully abstract denotational semantics for Flat Concurrent Prolog. In *LICS 88*, 1988.

[22] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[23] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Constraint processing in cc(FD). Technical report, Computer Science Department, Brown University, forthcoming 1992.

[24] Masahiro Hirata. Programming language doc and its self-description, or, **x = x** considered harmful. In *3d Conference Proceedings of Japan Society for Software Science and Technology*, pages 69–72, 1986.

[25] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International Series in Computer Science. Prentice-Hall, 1985.

[26] C.A.R. Hoare. A theory of conjunction and concurrency. Oxford PRG, May 1989.

[27] C.A.R. Hoare. Let's make models. In *Proceedings of CONCUR 90*, August 1990.

[28] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 111–119. ACM, January 1987.

[29] R. Jagadeesan and P. Panangaden. A domain-theoretic model of a higher-order process calculus. In M. S. Paterson, editor, *The Seventeenth International Colloquium On Automata Languages And Programming*, pages 181–194. Springer-Verlag, 1990. Lecture Notes In Computer Science 443.

[30] R. Jagadeesan, P. Panangaden, and K. Pingali. A fully abstract semantics for a functional language with logic variables. In *Proceedings of IEEE Symposium on Logic in Computer Science*, pages 294–303, 1989.

[31] R. Jagadeesan, V. Shanbhogue, and V. Saraswat. Angelic non-determinism in concurrent constraint programming. Technical report, System Sciences Laboratory, Xerox PARC, January 1991.

[32] Sverker Janson and Seif Haridi. Programming Paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the 1991 International Symposium.* MIT Press, 1991.

[33] Mark B. Josephs. Receptive process theory. Technical report, Programming Research Group, Oxford University, July 1990.

[34] 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.

[35] S. Kliger, E. Yardeni, E. Shapiro, and K. Kahn. The language **fcp(:,?)**. In *Conference on Fifth Generation Computer Systems*, December 1988.

[36] 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.

[37] J-.L. Lassez, M.J. Maher, and K. Mariott. *Foundations of Deductive Databases and Logic Programming*, chapter Unification Revisited. Morgan Kaufmann Publishers, Inc, 1988.

[38] Giorgio Levi. Models, unfolding rules and fixpoint semantics. In *Proceeedings of the Fifth International Conference and Symposium on Logic Programming, Seattle*, pages 1649–1665, August 1988.

[39] P. Lincoln. Linear logic. *SIGACT Notices*, 23(2):29–37, Spring 1992.

[40] Gary Lindstrom. Functional programming and the logical variable. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 266–280, January 1985.

[41] Michael Maher. Logic semantics for a class of committed-choice programs. In *4th International Conference on Logic Programming.* MIT Press, May 1987.

[42] R. Milner. *A calculus of communicating systems.* Number 92 in Lecture Notes in Computer Science. Springer-Verlag, 1980.

[43] R. Milner. Functions as processes. In M. S. Paterson, editor, *The Seventeenth International Colloquium On Automata Languages And Programming*, pages 167–180. Springer-Verlag, 1990. Lecture Notes In Computer Science 443.

[44] R. Milner. *Handbook of Theoretical Computer Science, Vol. B*, chapter Operational and Algebraic Semantics of Concurrent Processes. Elsevier Science Publishers B.V., 1990.

[45] R. Milner, J. G. Parrow, and D. J. Walker. A calculus for mobile processes, part i and ii. LFCS Report ECS-LFCS-89-85, University of Edinburgh, 1989.

[46] Ugo Montanari and Francesca Rossi. True concurrency semantics for concurrent constraint programminmg. In V. Saraswat and K. Ueda, editors, *Proc. of the 1991 International Logic Programming Symposium*, 1991.

[47] E.-R. Olderog and C.A.R. Hoare. Specification-oriented semantics for communicating processes. *Acta Informatica*, 23:9–66, 1986.

[48] Vijay A. Saraswat. Partial correctness semantics for $\mathbf{cp}(\downarrow, \mid, \&)$. In *Proceedings of the FSTTCS Conference*, number 206 in Lecture Notes in Computer Science, pages 347–368. Springer-Verlag, December 1985.

[49] Vijay A. Saraswat. A somewhat logical formulation of CLP synchronization primitives. In *Proceedings of LP 88.* MIT Press, August 1988.

[50] 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, 1992.

[51] Vijay A. Saraswat. *Concurrent constraint programming languages.* Doctoral Dissertation Award and Logic Programming Series. MIT Press, 1990, forthcoming.

[52] 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.

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

[54] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of Eighteenth ACM Symposium on Principles of Programming Languages, Orlando*, January 1991.

[55] 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.

[56] A. Scedrov. A brief guide to linear logic. *Bulletin of the European Assoc. for Theoretical Computer Science*, 41:154–165, June 1990.

[57] Dana S. Scott. Domains for denotational semantics. In *Proceedings of ICALP*, 1982.

[58] E. Y. Shapiro. Systolic programming: a paradigm of parallel processing. In *Proceedings of the Fifth Generation Computer Systems Conference*, 1984.

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

[60] Strand Software Technologies Inc. *Strand 88 Users Manual*, June 1989.

[61] Stephen Taylor. *Programming tecnhniques for parallel logic programming*. Prentice Hall, 1989.

[62] D.H.D. Warren. The Andorra Principle. Presented at Gigalips workshop, U. of Bristol, 1987.

[63] H. Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, Cambridge, MA, 1991.

[64] L. Alkalaj, T. Lang, and M. Ercegovac. Architectural Support for the Management of Tightly-Coupled, Fine-Grain Goals in Flat Concurrent Prolog. In *International Symposium on Computer Architecture*, pages 292–301, Seattle, June 1990. IEEE Computer Society Press.

[65] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *DFVLR: Conference on Parallel Processing in Science and Engineering*. Bonn-Bad Godesberg, June 1987.

[66] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[67] S. Asano, S. Isobe, and H. Sakai. The Unique Features of PIM/k: A Parallel Inference Machine with Hierarchical Cache System. Technical Report TR-767, ICOT, 1-4-28 Mita, Minato-Ku Tokyo 108, Japan, April 1992.

[68] R. Bahgat. *Pandora: Non-deterministic Parallel Logic Programming*. PhD thesis, Imperial College, Department of Computing, February 1991.

[69] R. Bahgat and S. Gregory. Pandora: Non-deterministic Parallel Logic Programming. In *International Conference on Logic Programming*, pages 471–486. Lisbon, MIT Press, June 1989.

[70] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell MA, 1988.

[71] C. Chandy and S. Taylor. *An Introduction to Parallel Programming*. Jones and Bartlett, Boston MA., 1991.

[72] A. Cheese. *Parallel Execution of Parlog*. Number 586 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1992.

[73] T. Chikayama. A Portable and Reasonably Efficient Implementation of KL1. Unpublished, ICOT, Tokyo, 1993.

[74] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *International Conference on Logic Programming*, pages 276–293. University of Melbourne, MIT Press, May 1987.

[75] K. L. Clark and S. Gregory. A Relational Language for Parallel Programming. In *Conference on Functional Programming Languages and Computer Architecture*, pages 171–178, Portsmouth NH, October 1981. ACM Press.

[76] V. S. Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 83–93. Williamsburg, ACM Press, April 1991.

[77] V. S. Costa, D. H. D. Warren, and R. Yang. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In *International Conference on Logic Programming*, pages 825–839. Paris, MIT Press, June 1991.

[78] J. A. Crammond. *Implementation of Committed-Choice Logic Languages on Shared-Memory Multiprocessors*. PhD thesis, Heriot-Watt University, Endinburgh, May 1988.

[79] J. A. Crammond. Scheduling and Variable Assignment in the Parallel Parlog Implementation. In *North American Conference on Logic Programming*, pages 642–657. Austin, MIT Press, October 1990.

[80] J. A. Crammond. The Abstract Machine and Implementation of Parallel Parlog. *New Generation Computing*, 10(4):385–422, August 1992.

[81] S. Debray, S. Kannan, and M. Paithane. Weighted Decision Trees. In *Joint International Conference and Symposium on Logic Programming*, pages 654–668. Washington D.C., MIT Press, November 1992.

[82] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–188, White Plains, NY, June 1990. ACM Press.

[83] S. Duvvuru, R. Sundararajan, E. Tick, A. V. S. Sastry, L. Hansen, and X. Zhong. A Compile-Time Memory-Reuse Scheme for Concurrent Logic Programs. In *International Workshop on Memory Management*, pages 264–276, St. Malo, September 1992. ACM Press.

[84] I. Foster. Information Hiding in Parallel Programs. Technical Report MCS-P290-0292, Argonne National Laboratory, 1992.

[85] I. Foster and S. Taylor. Flat PARLOG: A Basis for Comparison. *International Journal of Parallel Programming*, 16(2):87–125, 1987.

[86] I. Foster and S. Taylor. Strand: A Practical Parallel Programming Language. In *North American Conference on Logic Programming*, pages 497–512. Cleveland, MIT Press, October 1989.

[87] I. Foster and S. Taylor. A Compiler Approach to Scalable Concurrent Program Design. Technical Report MCS-P306-0492, Argonne National Laboratory, 1992.

[88] M. Furuichi, K. Taki, and N. Ichiyoshi. A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI. In *SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 50–59, Seattle, March 1990. ACM Press.

[89] A. Goto, Y. Kimura, T. Nakagawa, and T. Chikayama. Lazy Reference Counting: An Incremental Garbage Collection Method for Parallel Inference Machines. In *International Conference and Symposium on Logic Programming*, pages 1241–1256. University of Washington, MIT Press, August 1988.

[90] S. Gregory. *Parallel Logic Programming in PARLOG: The Language and its Implementation*. Addison-Wesley Ltd., Wokingham, England, 1987.

[91] D. Gudeman, K. De Bosschere, and S. K. Debray. jc: An Efficient and Portable Sequential Implementation of Janus. In *Joint International Conference and Symposium on Logic Programming*, pages 399–413. Washington D.C., MIT Press, November 1992.

[92] A. Harsat and R. Ginosar. CARMEL-4 The Unify-Spawn Machine for FCP. In *International Conference on Logic Programming*, pages 840–854. Paris, MIT Press, June 1991.

[93] M. Hirata. Programming Language Doc and its Self-Description, or, X=X is Considered Harmful. In *Proceedings of the 3^{rd} Conference of Japan Society of Software Science and Technolog*, pages 69–72, Tokyo, 1986.

[94] A. Houri and E. Y. Shapiro. A Sequential Abstract Machine for Flat Concurrent Prolog. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, pages 513–574. MIT Press, Cambridge MA, 1987.

[95] N. Ichiyoshi and K. Kimura. Asymptotic Load Balance of Distributed Hash Tables. In *International Conference on Fifth Generation Computer Systems*, pages 869–876, Tokyo, June 1992. ICOT.

[96] A. Imai and E. Tick. A Shared-Memory Multiprocessor Garbage Collector and its Evaluation for Committed-Choice Logic Programs. In *Symposium on Parallel and Distributed Processing*, pages 870–877. Dallas, IEEE Computer Society Press, December 1991.

[97] Y. Inamura, N. Ichiyoshi, K. Rokusawa, and K. Nakajima. Optimization Techniques Using the MRB and Their Evaluation on the Multi-PSI/V2. In *North American Conference on Logic Programming*, pages 907–921. Cleveland, MIT Press, October 1989.

[98] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *International Symposium on Logic Programming*, pages 167–183. San Diego, MIT Press, November 1991.

[99] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *International Symposium on Logic Programming*, pages 468–477. San Francisco, IEEE Computer Society Press, August 1987.

[100] A. King and P. Soper. Heuristics, Thresholding and a New Technique for Controlling the Granularity of Concurrent Logic Programs. Technical Report CSTR 92–08, Department of Electronics and Computer Science, University Of Southampton, 1992.

[101] A. King and P. Soper. Schedule Analysis of Concurrent Logic Programs. In *Joint International Conference and Symposium on Logic Programming*, pages 478–492. Washington D.C., MIT Press, November 1992.

[102] S. Kliger. *Compiling Concurrent Logic Programming Languages.* PhD thesis, The Weizmann Institute of Science, Rehovot, October 1992.

[103] H. Koike and H. Tanaka. Generation Scavenging GC on Distributed-Memory Parallel Computers. In *Proceedings of High Performance and Parallel Computing in Lisp.* EUROPAL Workshop, London, November 1990.

[104] K. Konishi, T. Maruyama, A. Konagaya, K. Yoshida, and T. Chikayama. Implementing Streams on Parallel Machines with Distributed Memory. In *International Conference on Fifth Generation Computer Systems*, pages 791–798, Tokyo, June 1992. ICOT.

[105] K. Kumon, A. Asato, S. Arai, T. Shinogi, A. Hattori, H. Hatazawa, and K. Hirano. Architecture and Implementation of PIM/p. In *International Conference on Fifth Generation Computer Systems*, pages 414–424, Tokyo, June 1992. ICOT.

[106] G. Lindstrom. Functional Programming and the Logic Variable. In *SIGPLAN Symposium on Principles of Programming Languages*, pages 266–280. New Orleans, ACM Press, 1985.

[107] B. C. Massey and E. Tick. Sequentialization of Parallel Logic Programs with Mode Analysis. In 4^{th} *International Conference on Logic Programming and Automated Reasoning*, St. Petersburg, July 1993.

[108] T. Miyazaki, A. Takeuchi, and T. Chikayama. A Sequential Implementation of Concurrent Prolog Based on the Shallow Binding Scheme. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, pages 496–512. MIT Press, Cambridge MA., 1987.

[109] T. Nakagawa, N. Ido, T. Tarui, M. Asaie, and M. Sugie. Hardware Implementation of Dynamic Load Balancing in the Parallel Inference Machine PIM/c. In *International Conference on Fifth Generation Computer Systems*, pages 723–730, Tokyo, June 1992. ICOT.

[110] K. Nakajima. Piling GC: Efficient Garbage Collection for AI Languages. In *IFIP Working Conference on Parallel Processing*, pages 201–204. Pisa, North Holland, May 1988.

[111] H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine: PSI-II. In *International Symposium on Logic Programming*, pages 104–113. San Francisco, IEEE Computer Society Press, August 1987.

[112] H. Nakashima, K. Nakajima, S. Kondo, Y. Takeda, Y. Inamura, S. Onishi, and K. Masuda. Architecture and Implementation of PIM/m. In *International Conference on Fifth Generation Computer Systems*, pages 425–435, Tokyo, June 1992. ICOT.

[113] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *International Symposium on Computer Architecture*, pages 156–167. Gold Coast, IEEE Computer Society Press, May 1992.

[114] M. Nilsson and H. Tanaka. FLENG Prolog—The Language which turns Supercomputers into Parallel Prolog Machines. In E. Wada, editor, *Proceedings of the Logic Programming Conference*, number 264 in Lecture Notes in Computer Science, pages 170–179. Springer-Verlag, June 1986.

[115] K. Nishida, Y. Kimura, A. Matsumoto, and A. Goto. Evaluation of MRB Garbage Collection on Parallel Logic Programming Architectures. In *International Conference on Logic Programming*, pages 83–95. Jerusalem, MIT Press, June 1990.

[116] T. Ozawa, A. Hosoi, and A. Hattori. Generation Type Garbage Collection for Parallel Logic Languages. In *North American Conference on Logic Programming*, pages 291–305. Austin, MIT Press, October 1990.

[117] G. M. Papadopoulos and D. E. Culler. Monsoon: an Explicit Token-Store Architecture. In *International Symposium on Computer Architecture*, pages 82–91. Seattle, IEEE Computer Society Press, May 1990.

[118] H. Pinto. Implementing Meta-Interpreters and Compilers for Parallel Logic Languages in Prolog. Master's thesis, University of Edinburgh, Artificial Intelligence Applications Institute, September 1986.

[119] V. A. Saraswat, K. Kahn, and J. Levy. Janus: A Step Towards Distributed Constraint Programming. In *North American Conference on Logic Programming*, pages 431–446. Austin, MIT Press, October 1990.

[120] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* MIT Press, Cambridge MA., 1989.

[121] M. Sato and A. Goto. Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor. In *IFIP Working Conference on Parallel Processing*, pages 305–318. Pisa, North Holland, May 1988.

[122] M. Sato, K. Kato, K. Takeda, and T. Oohara. Exploiting Fine Grain Parallelism in Logic Programming on a Parallel Inference Machine. Technical Report TR-676, ICOT, 1-4-28 Mita, Minato-ku Tokyo 108, Japan, August 1991.

[123] E. Y. Shapiro. A Subset of Concurrent Prolog and Its Interpreter. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 1, pages 27–83. MIT Press, Cambridge MA., 1987.

[124] E. Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[125] E. Y. Shapiro and C. Mierowsky. Fair, Biased, and Self-Balancing Merge Operators: Their Specification and Implementation in Concurrent Prolog. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, pages 392–413. MIT Press, Cambridge MA., 1987.

[126] E. Y. Shapiro and S. Safra. Multiway Merge with Constant Delay in Concurrent Prolog. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, pages 416–420. MIT Press, Cambridge MA., 1987.

[127] W. Silverman, M. Hirsch, A. Houri, and E. Y. Shapiro. The Logix System User Manual, Version 1.21. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, pages 46–77. MIT Press, Cambridge MA, 1987.

[128] M. Sugie, M. Yoneyama, N. Ido, and T. Tarui. Load-Dispatching Strategies on Parallel Inference Machines. In *International Conference on Fifth Generation Computer Systems*, pages 987–993, Tokyo, November 1988. ICOT.

[129] R. Sundararajan, A. V. S. Sastry, and E. Tick. Variable Threadedness Analysis for Concurrent Logic Programs. In *Joint International Conference and Symposium on Logic Programming*, pages 493–508. Washington D.C., MIT Press, November 1992.

[130] A. Takeuchi. *Parallel Logic Programming.* John Wiley & Sons, Inc., New York, 1992.

[131] K. Taki, K. Nakajima, H. Nakashima, and M. Ikeda. Performance and Architectual Evaluation of the PSI Machine. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 128–135. Palo Alto, IEEE Computer Society Press, October 1987.

[132] S. Taylor. *Parallel Logic Programming Techniques.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[133] E. Tick. *Memory Performance of Prolog Architectures.* Kluwer Academic Publishers, Norwell MA., 1987.

[134] E. Tick. Appraisal of Parallel Processing Research at ICOT. *Future Generation Computer Systems*, 9(2), 1993. Also available as University of Oregon CIS-TR-93-03.

[135] E. Tick and C. Banerjee. Performance Evaluation of Monaco Compiler and Runtime Kernel. In *International Conference on Logic Programming.* Budapest, MIT Press, June 1993. Also available as University of Oregon CIS-TR-92-21.

[136] E. Tick and J. A. Crammond. Comparison of Two Shared-Memory Emulators for Flat Committed-Choice Logic Programs. In *International Conference on Parallel Processing*, volume 2, pages 236–242, Penn State, August 1990.

[137] E. Tick and X. Zhong. A Compile-Time Granularity Analysis Algorithm and its Performance Evaluation. *New Generation Computing*, 11(3–4), June 1993. Also available as University of Oregon Technical Report CIS-TR-91-19a.

[138] K. R. Traub and D. E. Culler. Global Analysis for Partitioning Non-Strict Programs into Sequential Threads. In *Conference on Lisp and Functional Programming*, pages 324–334. San Francisco, ACM Press, 1992.

[139] R. Trehan. A Comparison of Committed Choice Non-Determinate Logic Languages Parallelism in a Mathematical Equation Solver (PRESS). Master's thesis, University of Edinburgh, Artificial Intelligence Applications Institute, September 1986.

[140] K. Ueda and T. Chikayama. Efficient Stream/Array Processing in Logic Programming Language. In *International Conference on Fifth Generation Computer Systems*, pages 317–326, Tokyo, 1984. ICOT.

[141] K. Ueda and M. Morita. Message-Oriented Parallel Implementation of Moded Flat GHC. In *International Conference on Fifth Generation Computer Systems*, pages 799–808, Tokyo, June 1992. ICOT.

[142] K. Ueda and M. Morita. Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, 1993. In press.

[143] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

[144] K. Yanoo. An Optimizing Compiler for a Parallel Inference Language. In H. Tanaka, editor, *Annual Report of the Research on Parallel Inference Engine*, pages 71–94. University of Tokyo, April 1992. (in Japanese).

[145] K. Yoshida and T. Chikayama. A'UM: A Stream-Based Object-Oriented Language. In *International Conference on Fifth Generation Computer Systems*, pages 638–649, Tokyo, November 1988. ICOT.