Concurrent Constraint Programming

Vijay A. Saraswat
Xerox PARC
Martin Rinard
Stanford University

(Extended Abstract)

Abstract

This paper presents a new and very rich class of (concurrent) programming languages, based on the notion of computing with *partial information*, and the concommitant notions of consistency and entailment.[1] In this framework, computation emerges from the interaction of concurrently executing agents that communicate by placing, checking and instantiating constraints on shared variables. Such a view of computation is interesting in the context of programming languages because of the ability to represent and manipulate partial information about the domain of discourse, in the context of concurrency because of the use of constraints for communication and control, and in the context of AI because of the availability of simple yet powerful mechanisms for controlling inference, and the promise that very rich representational/programming languages, sharing the same set of abstract properties, may be possible.

To reflect this view of computation, [Sar89] develops the cc family of languages. We present here one member of the family, cc($\downarrow,\rightarrow$) (pronounced "cc with Ask and Choose") which provides the basic operations of blocking Ask and atomic Tell and an algebra of behaviors closed under prefixing, indeterministic choice, interleaving, and hiding, and provides a mutual recursion operator. cc($\downarrow,\rightarrow$) is (intentionally!) very similar to Milner's CCS, but for the radically different underlying concept of communication, which, in fact, provides a general—and elegant—alternative approach to "value-passing" in CCS. At the same time it adequately captures the framework of committed choice concurrent logic programming languages. We present the rules of behavior for cc agents, motivate a notion of "visible action" for them, and develop the notion of c-trees and reactive congruence analogous to Milner's synchronization trees and observational congruence. We also present an equational characterization of reactive congruence for Finitary cc($\downarrow,\rightarrow$).

[1]The class is founded on the notion of "constraint logic programming" [JL87,Mah87], fundamentally generalizes concurrent logic programming, and is the subject of the first author's dissertation [Sar89], on which this paper is substantially based.

1 Introduction

Almost since its inception, computing science has been dominated by a view of computation based on the von Neumann memory model. Following [Dij76] we may say that in this view, the state of the system is specified by a *store*, which is a vector V of n variables of interest, and a *valuation* assigning to each variable in V a fully-formed (completely known) value in its domain. Thus each store describes a point in the n-dimensional *state space* for the system, obtained by taking the product of all possible values for the variables X_i. Let us define a *constraint* to be a subset of the state space, that is, a set of valuations. When designing a (possibly non-deterministic) algorithm, we are concerned with specifying a mechanism, such that when it is initiated in a store that satisfies a given constraint (the "pre-condition") it terminates in a store that satisfies another given constraint (the "post-condition"). Hence in any formal notation ("programming language") used to describe such algorithms, some basic operations are necessary to transform one store to another. In imperative programming languages, the basic operations executed by an agent are *Read*, which obtains the value of some variable in the store, and *Write* (or assignment) which translates the store parallel to some axis by *changing* the value of some variable (possibly based on applying some functions to the values obtained from prior reads).

One way to describe the notion of *constraint program-*

ming as presented here is to contrast it with the view presented above. The fundamental difference is that in constraint programming the store is itself a constraint, a set of valuations which provides *partial information* about the possible values for the given variables. The basic operations an agent may execute are *Ask* (instead of read), which checks whether or not the store *entails* a given constraint (that is, whether every valuation permitted by the store is permitted by the constraint), and *Tell* (instead of write), which adds a given constraint to the store, provided that the resultant store is *consistent* (that is, permits at least one valuation—note that the set of valuations corresponding to a conjunction of two constraints is the intersection of the set of valuations of each conjunct). Thus, a basic step does not change the value of a variable, but may rule out certain values that were previously possible; we say that the store is *monotonically refined*, since at any step of the computation, the set of possible values for the variable is contained in the set of possible values at any prior step. As above, an algorithm in this framework is designed in such a way that when it is initiated in a store that satisfies (entails) a given constraint, the pre-condition, it terminates in a store that satisfies (entails) another given constraint, the post-condition. In this way, the entire constraint programming framework is based on extremely general notions of partial information, with its concommitant notions of consistency and entailment.

A fundamental consequence of this radically different view of the store—which, incidentally, one may regard as the essence of (definite clause) logic programming—is that it enables a simple, elegant but very powerful paradigm for concurrent computation. Briefly, one may imagine multiple agents of the kind described above, executing concurrently in a shared store. Synchronization is achieved through a *blocking* Ask: an agent may block if it attempts to ask whether a constraint is entailed by the given store, and it is not yet so; it remains blocked until such time as (if ever) some other concurrently executing agents add constraints to the store such that the resulting store is strong enough to either entail the query or its negation.

The development of this notion of *concurrent constraint programming* is the subject of this paper, and the disseration [Sar89] on which it is based. Conceptually, the notion provides a theoretical footing for the intuitive, pre-theoretic visions about computing with constraints, evidenced in some of the work of such people as Sutherland, Sussman, Steele, Borning and some of their colleagues. Further, a coherent development of this notion leads to a synthesis and extension of some of the work on *concurrent logic programming*, which has occupied a number of researchers for the better part of this decade, and the more recent notion of *constraint logic programming* due to [JL87,Mah87]. Their union properly generalizes both: the former because its basic concepts of communication and synchronization are now seen to be applicable in a far more general setting, and the latter because it introduces a simple but powerful framework for control where earlier there was none. More generally, a major promise of this approach is that it will enable a very simple treatment of thorny issues in the semantics of concurrent, non-deterministic programming languages, and that, too, in the context of a versatile and expressive computational framework.

The rest of this paper is as follows. We flesh out the basic framework in more detail, and introduce the cc($\downarrow$,$\rightarrow$) family of languages. We discuss a wide number of constraint systems with respect to which the languages in the cc family may be instantiated, and their computational relevance. We discuss the notion of observation for the cc languages, and present characterizations for reactive equality and reactive congruence for these languages.

2 The basic framework

In general, we construe a constraint system to be any system of partial information that supports the notions of consistency and entailment. As such this notion is very weak, and can be used to capture a very broad class of interesting computational systems. For the present, we shall be content with a very simple class of first-order constraint systems:

Definition 2.1 A *simple first-order constraint system* is a quadruple $C = \langle \Sigma, \mathcal{A}, \mathtt{Var}, \Phi \rangle$ where Σ is a many-sorted first-order vocabulary, with associated set of sorts S and ranking function ρ, $\mathcal{A}$ is a Σ-structure, $\mathtt{Var}$ is an S-sorted set of *object-variables* and Φ, the set of *admissible constraints*, is some non-empty subset of $(\Sigma, \mathtt{Var})$-formulas, closed under conjunction. □

In order to execute programs in a concurrent constraint language, the implementation needs to ensure only that Ask (entailment) and Tell (consistency) operations can be performed on admissible constraints, with respect to a store which is itself a conjunction of admissible constraints. However, when presenting the "reactive" semantics for these languages in Section 4, we will find it necessary to apply these operations (and the operation that determines if a constraint is true for all valuations) on an extended set of constraints which is also closed under implication, and existential and universal quantification. Hence *reasoning* about programs using this semantics may require a more sophisticated constraint system than is necessary to implement these programs.

The most popular constraint system in use in (concurrent) constraint/logic programming languages is the system that we call Herbrand.[2] It interpretes a given vocabulary Σ over the free Σ-algebra. The constraints permitted are possibly existentially quantified conjunctions of equalities. Consistency in such a system can be checked by executing the (first-order) unification algorithm, and entailment by executing (a variation of) the matching algorithm.[3] More examples of computationally interesting constraint systems will be given in Section 3.

In the following, the many-sorted nature of a constraint system is not important, and we shall implicitly assume that various operations "preserve sorts". As usual, an *$\mathcal{A}$-valuation* is a mapping from Var to the domain of $\mathcal{A}$. In what follows, we will ambiguously regard a constraint c as denoting the possibly infinite set of valuations which realizes the constraint. Thus two constraints will be deemed equal if they yield the same set of valuations, even though their syntactic presentations may be different. (For example, if we interpret the operations and relations $\times, +, <$ in the standard manner, over the integers, the constraints $\mathtt{X} \times \mathtt{X} + \mathtt{Y} \times \mathtt{Y} = 25 \land 0 < \mathtt{X} < \mathtt{Y}$ and $\mathtt{X} = 3 \land \mathtt{Y} = 4$ will be deemed identical.) In the following, for any syntactic object χ (e.g., constraint) we shall use the notation $\mathbf{var}(\chi)$ to refer to the set of variables in χ. Further, if V is a finite subset of $\mathbf{var}(\chi)$, and $\mathbf{var}(\chi) \setminus V = X$, we shall use the notation $\delta V.c$ to refer to the constraint $\exists X.c$, which may be thought of as "projecting" out the information on the "hidden variables" X. Finally, we use the notation $(\exists)(c)$ to refer to the existential closure of c, that is, to $\delta\emptyset.c$.

2.1 The basic operations on store-as-constraint

Given that the store is a pool of partial information about the variables of interest, what sort of interesting basic operations can be defined on it?

We have already mentioned Ask and Tell. An "Ask c" operation succeeds if c is entailed by the store, fails if it is not, and suspends until it can either succeed or fail. A "Tell c" operation succeeds iff the conjunction of the store and c is consistent, and the store is atomically augmented with c if it is. (Note that both Ask and Tell are are *stable* operations, in the following sense. If at any stage in the past, the store successfully answered c, then the current store will successfully answer c; and if at some stage in the past some agent had succeeded in (atomically) telling c to the store, then it is always possible to tell c in the current store. Hence Ask-and-Tell languages possess the following stable property: if an agent executes an Ask or Tell at some stage of the computation, then at all subsequent stages it will be able to re-execute the same Ask or Tell.[4])

We mention in passing that work on concurrent logic programming [Ued85,CG86,Str89] has yielded some operations which may be simplified and generalized in this setting as refinements of atomic Tell. In order to model distributed programming languages, it turns out to be convenient to conceptualize not a *single* shared store, but to imagine that multiple agents are executing concurrently at different sites, each with its own local store. Atomic publication now corresponds to the publication of a constraint *simultaneously* at all sites. In general this operation can be rather expensive, and the more "loosely coupled" notion of *eventual publication* has been found to be very useful [Ued85,Sar89]:[5] namely, the constraint being published is added at the local site, and will be communicated to other sites asynchronously and over time. Computation will abort in case during this propagation an inconsistency is discovered. Hence eventual publication may be regarded as an "unsafe" approximation to atomic publication, motivated from efficiency concerns. The nice surprise of [Ued86] is that in many situations eventual publication can be used safely and predictably.[6]

Drawing on an early version of Parlog [CG86], [FT89a] loosens the requirements of publication even further, as not to require any consistency-checking at run-time.[7] Their ":=" operator is an instantiation, over a given constraint system, of the following more basic and more general operator. The Initialization operator takes a constraint $c(X, \bar{Y})$ and a variable X, sus-

[2]Though per [Dav83] there may be some justification for calling it Skolem.

[3]In fact, [Mah88] shows that checking the consistency of arbitrary first-order constraints in this setting is decidable.

[4]This property of the language was used with great effect in [SWKS88] to define the notion of a "live global snapshot" in this setting, and to show that it could be obtained with a very simple protocol. Such a snapshot can be used to determine whether a network of such agents satisfies some stable property.

[5][Ued86] discusses in detail the property of "antisubstitutability" that unification in GHC satisfies. The notion of eventual publication was extracted from this analysis in [Sar88,Sar89].

[6]In retrospect, perhaps eventual and atomic publication should have been called local and global publication. But the terminology is now well-established, and we shall stick with it. The local/global distinction also gives rise to two versions of Ask, with different implementation costs: one which checks whether the constraint is entailed from the local store and another which checks if the constraint is entailed in every store.

[7]In the case of *Strand*, which is defined over a slightly restricted subset of the Herbrand constraint system, this implies that no unification needs to be done at run-time.

pends until the store σ entails $\exists X.c$, and then succeeds iff $C \models \delta X.\sigma$ (ignoring c). On success, c is added to the store, without checking for consistency, which is guaranteed. In some settings [FT89a], these conditions can be checked very efficiently. (However, it must be kept in mind that if this operation is executed on the local store—as opposed to being executed at all sites simultaneously—it could still lead to computation being aborted, because of the asynchronous nature of inter-site communication.)

Some "unstable" basic operations (Check, Inform, Instantiate) of use in specialized situations are also discussed in [Sar89]. However, by far the most fundamental operations are Ask and (various approximations to) Tell, and it is on the Ask-and-Tell language cc($\downarrow$,$\rightarrow$) that we shall focus in this paper.

2.2 Building up more complex programs from simpler ones

Hitherto, we have examined in some detail a novel view of the store with two operations on it, Ask and Tell. Turning a blind eye to the connection with logic programming for a moment, it is interesting to speculate on the question: in what ways may these basic "Read" and "Write" operations be combined into more complex behaviors? That is, what are the combinators on agents one may wish to provide in order to obtain a reasonable (concurrent) programming language?

At this level of generality, this question has been asked and answered in earlier work by Milner [Mil83]:

> *In a definitive calculus there should be as few operators or combinators as possible, each of which embodies some distinct and intuitive idea, and which together give completely general expressive power ... If we disregard the recursion construction (some such construction is essential for defining infinite behaviour) we have reduced our combinators to the following four, with manifestly distinct roles:*
>
> 1. Product, *for combining agents concurrently ...;*
> 2. Action, *a combinator representing the occurrence of a single indivisible event;*
> 3. Summation, *the disjunctive combination of agents, allowing alternate courses of action;*
> 4. Restriction, *for delimiting the interface through which an agent interacts with others.*

Indeed the cc($\downarrow$,$\rightarrow$) languages have precisely these four combinators, though the definition of restriction is quite different in flavor because of the different underlying communication mechanism. (In fact, the cc framework does define other combinators, for non-deterministic choice, user-defined atomic transactions, reconciliation etc [Sar89]. However, most of these combinators are of use in the more general setting of (concurrent) indeterministic *and* non-deterministic ("backtracking") languages, which Milner was quite definitely not addressing. In this paper we confine our attention to cc($\downarrow$,$\rightarrow$), which does not allow non-deterministic choice.)

In the following, we use the meta-variables $A, A_1, \ldots$ to range over the set of agents, $b, b_1, \ldots$ to range over the basic actions, $X, X_1, \ldots$ to range over object-variables, $p, p_1, \ldots$ to range over procedure names, and R to range over parameter lists (lists of object-variables). In addition, we use the notation $\bar{\chi}$, for some meta-syntactic variable χ to stand for some finite, comma-separated list of the form $(\chi_1, \ldots, \chi_n)$.

The set of agent expressions may be defined inductively by the little grammar:[8]

$$\begin{aligned} \mathbf{A} ::= \; & \mathtt{nil} \mid \mathtt{stop} \mid c\downarrow \;\rightarrow A \mid c\star \rightarrow A \\ & \mid A + A \mid A \parallel A \mid \exists X.A \\ & \mid pR \mid (\mathtt{fix}_j \bar{p}\bar{R}\bar{A})R \end{aligned} \tag{1}$$

Programs must satisfy the syntactic condition that the length of parameter lists for any procedure is identical to the length of the list of actual arguments. (*Finitary* cc($\downarrow$,$\rightarrow$) is cc($\downarrow$,$\rightarrow$) without recursion.)

Intuitively, an agent can execute the basic action $c\downarrow$ in a store σ if σ entails c; otherwise the agent is suspended. An agent can execute the basic action $c\star$ in σ if σ accepts c, that is, if $\sigma \wedge c$ is consistent; otherwise it cannot progress. The agent $b \rightarrow A$ (for b some basic action) can behave like the agent A, provided that it can do the basic action b first. $A_1 \parallel A_2$ behaves like A_1 and A_2 executing in parallel, so that their basic actions are interleaved. $A_1 + A_2$ is the agent that may behave either like A_1 or like A_2, but the choice must be made when either A_1 or A_2 executes its first step. (Thus the choice may be influenced by the environment, unlike Hoare's "internal non-determinism".) $\exists X.A$ (read: "A, with X restricted") behaves like A, except that the variable X is "bound" in A and must be replaced at run-time with a new variable "local" to A that is, "renamed apart" from any other variable in the environment.[9] As a con-

[8] In actual programming languages in the cc family, we also introduce a binary sequential operator (":") which allows the generation of composite basic actions (to be executed atomically) from other simple and composite atomic actions. However, at least when the only basic operators allowed are Ask and Tell, the addition of such an operator does not increase the expressiveness of the language, and we omit it from discussion here.

[9] Such dynamic renaming can usually be implemented effi-

sequence, X is also ***hidden*** in A, in the sense that the environment cannot influence A through X, and A cannot influence the environment by posting constraints on X. This is the natural notion of hiding in cc languages since variables serve as the conduit for communication between concurrently executing agents. Next, pR is a procedure call on p, with the argument list R (a list of object-variables). $(\mathtt{fix}_j \bar{p}\bar{R}\bar{A})R'$ is the agent A_j, with its formal parameters R_j replaced by R', and with occurrences of the procedure variables $\bar{p}$ "bound" by the given declaration.

Finally, `nil` is the agent that has no behaviors. The primitive `stop` agent is necessary because we would like to distinguish successful termination of the system from unsuccessful termination. Intuitively, a `stop` agent executes a special visible action $c\surd$ indicating termination, where c is a projection of the current store, and then behaves like `nil`. However, the rules of behaviors of other agents are defined in such a way that agents $(A_1 \parallel A_2)$ can so indicate successful termination only when both A_1 and A_2 have; similarly for other combinators.

Example 2.1 (nrev/2 in this syntax) The usual "naive reverse" program may be written as:

```
(fix1(nrev, append)((X1, X2)(X1, X2, X3))
 ((null(X1) ↓  → (X1 = X2)⋆ → stop
   + cons(X1) ↓
      → ∃X3.∃X4.((X4 = [car(X1)], X3 = cdr(X1))⋆
                 → ∃X5.(nrev(X3, X5)
                        || append(X5, X4, X2)))),
  (null(X1) ↓  → (X3 = X2)⋆ → stop
   + cons(X1) ↓
   → ∃X4.∃X5.(car(X3) = car(X1),
              X4 = cdr(X1), X5 = cdr(X3))⋆
          → append(X4, X2, X5))))(A, B).
```

In the above, the constraints `null`, `cons` etc are used with their obvious interpretation. Intuitively, this agent expects its environment to constrain its first argument (`A`) to be a list, over time, and in return, it will constrain its second argument (`B`) to be the reverse of its first argument.

In reality such a program is written, with obvious shorthand, as:

```
(fix1(nrev, app)((X1, X2)(X1, X2, X3))
 ((null(X1) ↓  → (X1 = X2)⋆ → stop
   + cons(X1) ↓  → ∃X5.(nrev(cdr(X1), X5)
                        || append(X5, [car(X1)], X2))),
  (null(X1) ↓  → (X3 = X2)⋆ → stop
   + cons(X1) ↓  →(car(X3) = car(X1))⋆
                  → append(cdr(X1), X2, cdr(X3)))
 ))(A, B).
```

And this, in turn, is not too far from the "Ask-tell" clausal syntax—closer to conventional logic programming syntax—employed in [Sar88]:

```
nrev(X1, X2) ← null(X1) : (X1 = X2) | stop.
nrev(X1, X2) ← cons(X1) : true |
   nrev(cdr(X1), X5), append(X5, [car(X1)], X2).
append(X1, X2, X3) ← null(X1) : (X2 = X3) | stop.
append(X1, X2, X3) ← cons(X1) : car(X1) = car(X3) |
   append(cdr(X1), X2, cdr(X3)).
```

□

Programs in the "algebraic" syntax introduced above may readily be translated to programs in the usual "clausal" syntax for logic programming languages, and vice versa. (We take it to be a strength of Milner's analysis that it is applicable even to the seemingly unrelated context of (concurrent) constraint programming languages.) Nevertheless we prefer to work in this paper with the algebraic notation which is more convenient for the task at hand, and it makes the connections with CCS/CSP more obvious. Indeed the algebraic notation presented above may well be regarded as abstract syntax for the concrete "clausal" syntax of logic programming.

2.3 The behavior of cc(↓,→) agents

To define the behaviors of agents in this language, we employ an indexed labelled transition system, with set of configurations (also called *processes*) $(P, Q \ldots \in)\Gamma = \{\langle A, \sigma\rangle \mid \sigma \text{ consistent}\}$ and the set of labels $(\lambda \in)\Lambda$ (defined below) and indexed by $\{V \subseteq_f \mathbf{Var}\}$, the set of finite sets of variables. Formally, a store σ consists of a constraint and a set of variables; $c : V$ represents the store with constraint c and set of variables V. We require that $\mathbf{var}(c) \subseteq V$ for all stores $c : V$, and extend some operations on constraints to operations on stores

ciently in distributed systems, without the kind of global knowledge that might seem necessary. Indeed, its effect is to provide a distributed, uniform, virtual name-space across all sites participating in the computation.

in the following manner:

$$\begin{array}{lcl}
c:V \Rightarrow c' & \equiv & c \Rightarrow c' : V \cup \mathbf{var}(c') \\
c' \Rightarrow (c:V) & \equiv & c' \Rightarrow c : V \cup \mathbf{var}(c') \\
c:V \Rightarrow c' : V' & \equiv & c \Rightarrow c' : V \cup V' \\
c:V \wedge c' & \equiv & c \wedge c' : V \cup \mathbf{var}(c') \\
c' \wedge (c:V) & \equiv & c' \wedge c : V \cup \mathbf{var}(c') \\
c:V \wedge c' : V' & \equiv & c \wedge c' : V \cup V' \\
\mathcal{C} \models c : V & \equiv & \mathcal{C} \models c \\
\forall X.c : V & \equiv & \forall X.c \\
\exists X.c : V & \equiv & \exists X.c
\end{array}$$

The relation $V \vdash P \xrightarrow{\lambda} Q$ is to be read as: "P is observed to take the action λ, on the visible variables V, and then behave like Q". In this section we define a *transformational* semantics, which is adequate to capture the behavior of the program executed in isolation from its environment. (In Section 4, we will consider which basic actions taken by an agent should be considered visible.) Hence the only observation allowed is the observation of a constraint c on successful termination of the store. This corresponds to the usual situation in logic programming languages when a user interacts with the system by executing an agent and waiting for the "answer bindings" (here: constraints) to be printed on the screen. Hence we set $\Lambda = \{\tau\} \cup \{c\surd\}$, where we take τ to stand for the "silent" action, indicating that the process has made internal progress.

The transition relation is defined inductively in the usual SOS style. A basic action can be executed when the relevant conditions are satisfied:

$$\frac{\mathcal{C} \models (\exists)(\sigma \wedge c)}{V \vdash \langle c\star \rightarrow A, \sigma\rangle \xrightarrow{\tau} \langle A, \sigma \wedge c\rangle} \qquad \frac{\mathcal{C} \models \sigma \Rightarrow c}{V \vdash \langle c\downarrow \rightarrow A, \sigma\rangle \xrightarrow{\tau} \langle A, \sigma\rangle} \tag{2}$$

(From now on, we shall write $P \xrightarrow{\lambda} Q$ for the assertion $V \vdash P \xrightarrow{\lambda} Q$ whenever each assertion in a transition rule uses the same index V.) The definition of interleaving and choice are standard. Note that we use "Milner's choice", as opposed to Hoare's internal and external indeterministic choices [BHR84]. It turns out to be much more convenient to implement, and all concurrent logic programming languages implement this version of choice:

$$\frac{\langle A_1, \sigma\rangle \xrightarrow{\lambda} \langle A_1', \sigma'\rangle \quad (\lambda \neq c\surd \text{ for some } c)}{\begin{array}{c}\langle A_1 \parallel A_2, \sigma\rangle \xrightarrow{\lambda} \langle A_1' \parallel A_2, \sigma'\rangle \\ \langle A_2 \parallel A_1, \sigma\rangle \xrightarrow{\lambda} \langle A_2 \parallel A_1', \sigma'\rangle \\ \langle A_1 + A_2, \sigma\rangle \xrightarrow{\lambda} \langle A_1', \sigma'\rangle \\ \langle A_2 + A_1, \sigma\rangle \xrightarrow{\lambda} \langle A_1', \sigma'\rangle\end{array}} \tag{3}$$

The axioms for successful termination are equally straightforward:

$$\frac{\begin{array}{c}\langle A_1, c:V\rangle \xrightarrow{c\surd} \langle \mathtt{nil}, c:V\rangle \\ \langle A_2, c:V'\rangle \xrightarrow{c\surd} \langle \mathtt{nil}, c:V'\rangle\end{array}}{\langle A_1 \parallel A_2\rangle \xrightarrow{c\surd} \langle \mathtt{nil}, c: V \cup V'\rangle} \qquad \frac{\langle A_1, \sigma\rangle \xrightarrow{c\surd} \langle \mathtt{nil}, \sigma\rangle}{\begin{array}{c}\langle A_1 + A_2, \sigma\rangle \xrightarrow{c\surd} \langle \mathtt{nil}, \sigma\rangle \\ \langle A_2 + A_1, \sigma\rangle \xrightarrow{c\surd} \langle \mathtt{nil}, \sigma\rangle\end{array}} \tag{4}$$

Next we consider the behaviors of $\exists X.A$.

$$\frac{\langle A[Y/X], c: V' \cup \{Y\}\rangle \xrightarrow{\lambda} Q \quad Y \notin V'}{V \vdash \langle \exists X.A, c:V'\rangle \xrightarrow{\lambda} Q} \tag{5}$$

The axiom for mutual recursion is the usual "stay one step ahead of the execution" one, modified in the obvious way to allow parameter transmission. As is conventional, we use the notation $A[\mathtt{fix}\bar{p}\bar{R}\bar{A}/\bar{p}]$ to indicate the agent obtained by simultaneously replacing all occurrences of p_i in A by $\mathtt{fix}_i\bar{p}\bar{R}\bar{A}$ and renaming bound (object and procedure) variables to avoid capture. In the following rule, R_j is presumed to be some list of variables $\bar{X}$.

$$\frac{\langle A_j[\bar{Y}/\bar{X}][\mathtt{fix}\bar{p}\bar{R}\bar{A}/\bar{P}], \sigma\rangle \xrightarrow{\lambda} Q}{\langle(\mathtt{fix}_j\bar{p}\bar{R}\bar{A})\bar{Y}, \sigma\rangle \xrightarrow{\lambda} Q} \tag{6}$$

Finally, the single axiom for **stop** is evident—in store σ **stop** can make a single transition, publishing the constraint $\delta V.\sigma$ (the store projected onto the visible variables):

$$V \vdash \langle \mathtt{stop}, \sigma\rangle \xrightarrow{\delta V.\sigma\surd} \langle \mathtt{nil}, \sigma\rangle \tag{7}$$

nil has no behaviors.

The *computations* of a process are defined in the standard way, as sequences of transitions. Terminating computations may yield answers. The *set of answers* of a process P, $SS(P)$ is the set:

$$SS(P) = \{c \mid \mathbf{var}(P) \vdash P(\xrightarrow{\tau})^\star \xrightarrow{c\surd} Q, \text{ for some } Q\ \}$$

where we use the notation $P\mathcal{R}_1\mathcal{R}_2Q$ to indicate relational composition ,"$\star$" is the Kleene star.

2.4 Comparison with previous work

cc($\downarrow$,$\rightarrow$), as we have sketched it above, is essentially a "committed choice" concurrent logic programming language, parameterized over the embedded constraint system, and with the operations of blocking Ask and atomic Tell. Indeed, all the major concurrent logic programming languages neatly fit into the cc/Herbrand framework. Flat Parlog [Gre87] and Flat GHC [Ued86] are revealed as Eventual **Herbrand**, that is, as cc languages over the **Herbrand** constraint system, with Blocking Ask and Eventual Tell operations. **FCP**$(\downarrow, |)$ [Sar85] (modulo a minor adjustment discussed in [Sar87a]) is revealed as Atomic **Herbrand**, that is, the cc language over **Herbrand**, with Blocking Ask and Atomic Tell operation. Strand, the first commercially available programming language in this framework [Str89], is the cc language with Blocking Ask and Initialize basic operations, over a slightly simplified version of **Herbrand**.[10]

The treatment of the transformational semantics for the cc languages is based on [Sar89]. It considerably simplifies and generalizes the treatment of "flat" languages in [Sar87b], besides presenting it in an algebraic setting. The relationship of this work with some more recent work on the semantics of concurrent logic programming languages is discussed in Section 4.

[Mah87] first suggested, in the context of the design of the language ALPS (which may be thought of as being a cleaner "logical" variant of FGHC) how the synchronization condition for FGHC could be modelled logically. This paper was very influential in our development of the concurrent constraint programming framework. [Sar88] first presented the Ask and Tell metaphor, but it concentrated on showing how a number of synchronization mechanisms in concurrent logic programming languages could be expressed in this constraint-based setting. [Sar89] developed the concurrent constraint programming framework as presented here. The debts of this work to Milner's development of CCS are obvious.

More generally, from a concurrent programming point of view, there are tantalizing connections with the UNITY work [CM88]. cc languages are essentially nondeterministic transition systems over partially defined stores. The focus in this paper (and in [Sar89]), is to show that very sophisticated and usable programming languages emerge provided that the notion of communication and synchronization is suitably enriched.

3 Constraint systems

Concurrent logic programming languages, which have been thoroughly studied in the last few years, are currently the most well-developed exemplars of the power of constraint-based communication. Many programming idioms for these languages have been discovered, which deal, for example, with incomplete messages (messages whose responses can automatically—and efficiently—be routed back to the receiver), recursive doubling (technique for path-shortening), short circuits (used for the detection of stable properties of networks [SWKS88]), producer consumer synchronization, many-to-one communication using bags, techniques for modelling agents with mutable local state, etc. Note that these languages already have the power to communicate embedded ports in messages, without having to resort to infinite summations, as has variously been proposed for CCS. Further, these idioms allow the natural expression of fine-grained concurrency. Many of these idioms are discussed in the literature in such places as [Sha83], [Kah89], [FT89b],[Sar89], [Gre87],[Ued86] etc.

However, **Herbrand** is not the only kind of constraint system based on trees which is useful in this setting. For example [Sar89] shows that with a suitable choice of vocabulary, it is possible to get fixed-width arrays, extensible arrays, infinite arrays along two dimensions, records, "feature-structures" [AK84,Smo88] etc. in very natural ways.

More generally, constraint-based systems are endemic in AI. They range from systems based on Boolean algebras in use in knowledge representation systems with sophisticated inheritance schemes,[11] to systems arising from the so-called "hybrid reasoning" approach in AI, to explicitly tailored systems used in various vision algorithms [MR88], to discrete constraint systems [DvH$^+$88] used in various search problems, to propositional caculus based constraint sytems used in conjunction with various kinds of "truth maintenance systems" to control problem-solvers, to constraint systems used in qualitative reasoning about the physical world, and so on.

Of course the use of constraint systems in traditional algebra and geometry settings is already well-known, and has been exploited in the constraint programming setting in [JLL].

We end with a few slogans which capture the flavor of constraint-based communication. These slogans are expanded on in much more depth in [Sar89].

Constraints specify partial information. Agents do not synchronize only on the availability of complete information about objects (variables). Two or more agents may simultaneously produce non-redundant pieces of information about the same variable.

Communication is additive. Only constraints that are consistent with previously placed constraints can

[10] We remark in passing that Flat Concurrent Prolog is not covered by the cc framework. We are not concerned however (on the contrary!) since that language was, in retrospect, a rather poorly designed language, with a very obscure synchronization mechanism. (The arcane details may be found in [Sar89].) Indeed, Shapiro and his colleagues have now moved over to Atomic Herbrand, essentially abandoning Flat Concurrent Prolog [KYSK88].

[11] Indeed, [Sar89] borrowed the Ask and Tell terminology from [BFL85]!

be added to the system. Once a constraint is added to the system, it stays forever. This leads to the stability of the Ask and Tell operations.

Constraints are types. The types-as-sets view makes types monadic constraints. More sophisticated type systems allow inheritance (implicational hierarchies), type specialization (conjunctions) and classification (exclusive disjunctions). The constraints view provides a coherent conceptual framework for a much richer space of type specifications, including multiadic types.

Communication channels may be embeddable. In the Herbrand constraint system, variables may be communicated in messages between agents with the same ease as data.

Communication is open. Whenever an agent places a constraint on a variable, any agent with access to that variable is affected. There is no a priori distinction betwen producers and consumers. In order to be affected by the environment, an agent does not need to know *how* the communication occurred: who generated it, when was it generated, etc.

4 Abstract semantics for cc($\downarrow$,$\rightarrow$) languages

In Section 2 we gave an operational semantics for cc($\downarrow$,$\rightarrow$), which associates with each agent the set of "answers" obtained by running the agent. However, such a semantics is obviously not compositional, since not enough information is stored with the semantic objects. In this section we take the first steps towards identifying the basic ideas that must be used in the semantic modelling of the cc($\downarrow$,$\rightarrow$) languages. We introduce the notion of "visible variables" of an agent—the set of variables that form the interface between the agent and its "environment"—and analyze the notion of "visible actions" for a cc($\downarrow$,$\rightarrow$) agent, axiomatize it in the standard SOS style, define c-trees to be the derivation trees obtained by "unrolling" the one-step transition relation, introduce two "bisimulation" equivalences on c-trees from which we derive two congruences (with respect to cc($\downarrow$,$\rightarrow$) combinators), axiomatize these congruences for Finite cc($\downarrow$,$\rightarrow$), and argue that one of them is the "finest reasonable" congruence on cc($\downarrow$,$\rightarrow$) agents.

4.1 Visible actions and reactive semantics

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—may not be directly relevant.

In the following, we analyze this problem in detail. We define the (indexed) *reactive transition system*, over the set of configurations $\Gamma_r = \Gamma = \{\langle A, \sigma\rangle \mid \sigma \text{ consistent}\}$ (indexed as before by $\{V \subseteq_f \mathtt{Var}\}$) and the set of labels $\Lambda_r = \Lambda \cup \{c\downarrow, c\star\}$ extended to include the basic actions. The interpretation of the transition $V \vdash P \xrightarrow{c\star} Q$ is that the process P may in one step add the new information c (in the visible variables V) to the store, and then behave like Q. Similarly, the interpretation of $V \vdash P \xrightarrow{c\downarrow} Q$ is: in order to progress, P must assume that the environment supplies a constraint at least as strong as c (in the visible variables), where c is information new to the store.

Let us now consider the axioms for $\longrightarrow$. Suppose an agent engages in the basic action $c\star$ in the store σ. Clearly, this action is invisible to the environment if $\mathcal{C} \models \sigma \Rightarrow \delta V.(\sigma \wedge c)$, where V is the set of visible variables. For, any action that the environment can engage in before the agent executes this action, it can engage in after the agent executes this action. (Note that c may add information on "non-visible" variables, but this can never be detected directly by the environment.) Even though the agent has made progress, the ability of the environment to progress has not been affected in any way; hence the agent has made *silent* progress. We thus have the axiom:

$$\frac{\mathcal{C} \models (\exists)(\sigma \wedge c) \quad \mathcal{C} \models (\sigma \Rightarrow \delta V.(\sigma \wedge c))}{V \vdash \langle c\star \rightarrow A, \sigma\rangle \xrightarrow{\tau} \langle A, \sigma \wedge c\rangle} \tag{8}$$

Otherwise, if the constraint can in fact be published (that is, is consistent with the store) then progress by the agent has resulted in new information being added to the store. The information added is the same as that added by any other agent engaging in a basic action $c'\star$ which is identical to $c\star$ (as far as the set of visible variables V is concerned) in the context of σ, i.e., is such that $\delta V.(\sigma \wedge c) \iff \delta V.(\sigma \wedge c')$. Subsequent to this action, the environment is prohibited from engaging in any action inconsistent with $\delta V.(\sigma \wedge c)$; hence we say that the agent has engaged in the visible action $\delta V.(\sigma \wedge c)\star$:

$$\frac{\mathcal{C} \models (\exists)(\sigma \wedge c) \quad \mathcal{C} \not\models \sigma \Rightarrow \delta V.(\sigma \wedge c)}{V \vdash \langle c\star \rightarrow A, \sigma\rangle \xrightarrow{\delta V.(\sigma\wedge c)\star} \langle A, \sigma \wedge c\rangle} \tag{9}$$

Similar considerations apply if an agent engages in the action $c\downarrow$. If the store σ is such that $\mathcal{C} \models \sigma \Rightarrow c$, then the agent has made silent progress; for, no (consistent) action taken by the environment can either influence this progress or be influenced by it. Assume now that $\mathcal{C} \not\models (\sigma \Rightarrow c)$. Let X be the set of variables in the store less the visible variables V. Now it is possible for the agent to progress iff it is possible for the environment to supply some new constraint c_0' on V such that when conjoined with σ, it entails c. Hence c_0' must satisfy the properties:

1. $\mathcal{C} \models (\exists)(\sigma \wedge c_0')$, and,
2. $\mathcal{C} \models c_0' \Rightarrow (\sigma \Rightarrow c)$.

If there is a constraint c_0' on the variables V which satisfies these properties, then the weakest such constraint is: $c_0 \equiv \forall X.(\sigma \Rightarrow c)$. [12] Hence we say that if $\mathcal{C} \not\models \sigma \Rightarrow c$, and $\mathcal{C} \models (\exists)(\sigma \wedge c_0)$, an agent $c \rightarrow A$ may, in store σ, engage in the new visible action $\delta V.(\sigma \wedge c_0)\downarrow$. The resulting store, $\sigma \wedge c_0$ is exactly strong enough to entail c. Thus we have the axioms:

$$\frac{\mathcal{C} \models (\exists)(\sigma \wedge c) \quad \mathcal{C} \models \sigma \Rightarrow c}{V \vdash \langle c\downarrow \ \rightarrow A, \sigma\rangle \xrightarrow{\tau} \langle A, \sigma\rangle} \tag{10}$$

$$\frac{\begin{array}{l}\mathcal{C} \not\models \sigma \Rightarrow c \\ c' : V \sqcup X \equiv \sigma \quad c_0 \equiv \forall X.(\sigma \Rightarrow c) \\ \mathcal{C} \models (\exists)(\sigma \wedge c_0)\end{array}}{V \vdash \langle c\downarrow \ \rightarrow A, \sigma\rangle \xrightarrow{\delta V.(\sigma\wedge c_0)\downarrow} \langle A, \sigma \wedge c_0\rangle} \tag{11}$$

The axioms for **stop**, choice, interleaving, the restriction operation and recursion (Rules 3,4, 5, 6 and 7) remain unchanged. This rounds up the list of axioms that $\longrightarrow$ satisfies.

4.2 Derivation trees

The above rules implicitly define the tree of actions that a cc agent engages in (obtained by "unrolling" the one step "$\xrightarrow{\lambda}$" relation, indexed by V, the set of free variables of the agent). Such trees, which we shall call *c-trees*, play the same fundamental role for cc that synchronization trees play for CCS. In brief, c-trees are trees whose arcs are labelled with elements of Λ^r and which satisfy the conditions:

- if an arc is not labelled with τ then the constraint labelling it is consistent and strictly stronger than any constraint labelling an ancestor arc, and,
- if the arc entering a node is labelled with $c\surd$ then the node has no outgoing arcs.

If two agents have the same c-tree, then they have the same success set; in fact the success set can be obtained in an obvious way by considering the "ask-free" branches of the c-tree.

Next we consider when two agents may be said to have c-trees that are "similar":

Definition 4.1 A relation $\sim$ on processes is a *rigid V-bisimulation* iff for all processes P and Q such that $P \sim Q$, if $V \vdash P \xrightarrow{\lambda} P'$ then $V \vdash Q \xrightarrow{\lambda} Q'$ and $P' \sim Q'$, and vice versa. □

Definition 4.2 (Reactive equality) Two agents A and A' in free variables V and V' respectively are said to be *reactively equal* (written $A \simeq A'$) iff there exists a rigid $(V \cup V')$-bisimulation $\sim$ such that $\langle A, \emptyset : V \cup V'\rangle \sim \langle A', \emptyset : V \cup V'\rangle$. Two processes P and Q are *reactively equal on V* (written $V \vdash P \simeq Q$) iff there exists a rigid V-bisimulation $\sim$ such that $P \sim Q$. □

The theory of bisimulation has been explored in a number of places e.g. [Mil84] and is mathematically very attractive.

Theorem 4.1 *Reactive Equality is a congruence for Finitary* cc($\downarrow$,$\rightarrow$). *That is, if $A_1 \simeq A_1'$ and $A_2 \simeq A_2'$ then:*

$$\begin{array}{ll} b \rightarrow A_1 \simeq b \rightarrow A_1' & A_1 \parallel A_2 \simeq A_1' \parallel A_2' \\ A_1 + A_2 \simeq A_1' + A_2' & \exists X.A_1 \simeq \exists X.A_1' \end{array}$$

The congruence proofs for the various cases consist of constructing a relation containing the appropriate processes, and then using several key lemmas (some of which are given below), to show that this relation is a bisimulation.[13]

First we need some definitions. Say that two constraint c, c' are *V-identical* iff $\mathcal{C} \models \delta V.c \iff \delta V.c'$. (From now on we shall abuse notation and write $c_1 = c_2$

[12]There may be no such constraint because it may not be possible to answer c from σ given that only a V-constraint can be added to σ. For example, if σ is `true`, c is `Y=1`, and V is any set of variables which does not contain `Y`.

[13]Detailed proofs of this result and others in this paper may be found in [SRng].

to mean that $\mathcal{C} \models c_1 \iff c_2$.) For a constraint c and store σ, say that c *extends* σ *on* V iff $\mathcal{C} \models (\exists)(\sigma \wedge c)$ and if X is the set of variables occurring in σ less V, then $\mathcal{C} \models c \iff \exists X.c$. Thus c may constrain variables other than the non-visible ones of σ.

Let $P \equiv V \vdash \langle A, \sigma\rangle \xrightarrow{\lambda_1} Q$ and $P' \equiv V \vdash \langle A, \sigma'\rangle \xrightarrow{\lambda_2} Q'$ be two transitions. We say they *correspond* iff each is generated from the same subterm of A. This notion can be generalized in the obvious way if the agents in P and P' are not identical but are variants of each other.

Lemma 4.2 (Anti-augmentation lemma)
If $V \vdash \langle A, \sigma \wedge d\rangle \xrightarrow{\lambda_1} \langle A_1, \sigma_1\rangle$ *and* d *extends* σ *on* V, *then there exists a corresponding transition* $V \vdash \langle A, \sigma\rangle \xrightarrow{\lambda_2} \langle A_1, \sigma_2\rangle$, *where* $\sigma_1 = \sigma_2 \wedge d$.

Lemma 4.3 *Let* $V \vdash \langle A, \sigma \wedge d\rangle \xrightarrow{\lambda_1} \langle A_1, \sigma_1\rangle$ *be a transition, where* d *extends* σ *on* V. *Let* $V \vdash \langle A, \sigma\rangle \xrightarrow{\lambda_2} \langle A_1, \sigma_2\rangle$ *be a corresponding transition. Then, for any agent* A', *store* σ' V*-identical to* σ, *and constraint* d' V*-identical to* d *and extending* σ' *on* V, *if* $V \vdash \langle A', \sigma'\rangle \xrightarrow{\lambda_2} \langle A'_1, \sigma'_2\rangle$ *then we must have* $V \vdash \langle A', \sigma' \wedge d'\rangle \xrightarrow{\lambda_1} \langle A'_1, \sigma'_2 \wedge d'\rangle$.

Lemma 4.4 (Augmentation lemma for $\simeq$)
Let $V \vdash \langle A, \sigma\rangle \simeq \langle A', \sigma'\rangle$, *where* σ, σ' *are* V*-identical. Let* d, d' *be two constraints such that* d *extends* σ *on* V, d' *extends* σ' *on* V, *and* d *is* V*-identical to* d'. *Then* $V \vdash \langle A, \sigma \wedge d\rangle \simeq \langle A', \sigma' \wedge d'\rangle$.

We give an equational characterization of reactive equality for Finitary cc($\downarrow$,$\rightarrow$)in Table 1.

Theorem 4.5 *The laws* $(A1)$-$(A8')$ *and* $(B1)$-$(B8)$ *are complete for reactive equality for Finitary* cc($\downarrow$,$\rightarrow$).

We prove the theorem by defining a normal form, showing that each agent is related to its normal form by equational reasoning using the given laws, and showing that bisimilar agents have identical normal forms. The normal form of an agent differs from its c-tree only in that some branches of the normal form may terminate in stop, whereas corresponding branches of the c-tree terminate in a label of the form $c\surd$.

Each agent can be converted to its normal form by using $(A8')$ and $(B1)$ to "push down" $\|$ constructs, and $(B5)$-$(B8)$ to push down constructs of the form $\exists X$. $(B2)$ and $(B3)$ propagate constraints down the tree, ensuring that each branch of the tree has the correct labels, while $(B4)$ trims inconsistent branches and converts each Ask and Tell that adds no new information to the visible variables into a τ.

Law $(B6b)$ deserves futher explanation. Because applying this law hides any information about X in c from

Generic laws for reactive equality are given below. They are applicable when the index set $I = \emptyset$, since we take `nil` to be the zero for disjunction. All the laws numbered An are from [Mil84]. We use the notation $[b]$ to refer to b's constraint.

$$
\begin{array}{ll}
(A1) & x + (y + z) = (x + y) + z \\
(A2) & x + y = y + x \\
(A3) & x + x = x \\
(A4) & x + \mathtt{nil} = x \\
(A8') & u \parallel v = \Sigma_{i\in I} b_i \rightarrow (A_i \parallel v) \\
 & \qquad\qquad + \Sigma_{j\in J} b'_j \rightarrow (u \parallel A'_j) \\
 & \quad \text{where } u = \Sigma_{i\in I}(b_i \rightarrow A_i) \\
 & \quad \text{and } v = \Sigma_{j\in J}(b'_j \rightarrow A'_j)
\end{array}
$$

The following laws are specific to cc($\downarrow$,$\rightarrow$). The term τ is to be thought of as shorthand for the basic operation `true`$\star$.

$$
\begin{array}{ll}
(B1) & \mathtt{stop} \parallel A = A = \mathtt{stop} \parallel A \\
(B2) & b \rightarrow ((c\downarrow \rightarrow A') + A) = \\
 & b \rightarrow (([b] \wedge c)\downarrow \rightarrow A') + A) \\
 & b \rightarrow ((c\star \rightarrow A') + A) = \\
 & b \rightarrow (([b] \wedge c)\star \rightarrow A') + A)
\end{array}
$$

$$
\begin{array}{ll}
(B3) & b \rightarrow ((c\downarrow \rightarrow A') + A) = \\
 & b \rightarrow (([b] \Rightarrow c)\downarrow \rightarrow A') + A) \\
 & b \rightarrow ((c\star \rightarrow A') + A) = \\
 & b \rightarrow (([b] \Rightarrow c)\star \rightarrow A') + A)
\end{array}
$$

$$
(B4)\quad \frac{\mathcal{C} \models \neg[b]}{(b \rightarrow A = \mathtt{nil})} \qquad \frac{\mathcal{C} \models [b]}{(b \rightarrow A = \tau \rightarrow A)}
$$

$$
(B5)\quad \exists X.\Sigma_{i\in I}(b_i \rightarrow A_i) = \Sigma_{i\in I}\exists X.(b_i \rightarrow A_i)
$$

$$
(B6a)\quad \exists X.(c\star \rightarrow \mathtt{stop}) = (\exists X.c)\star \rightarrow \exists X.\mathtt{stop}
$$

$$
(B6b)\quad \frac{\begin{array}{l}\forall i \in I.\mathcal{C} \models c_i \Rightarrow c \\ \forall j \in J.\mathcal{C} \models c \vee c'_j \\ \forall j \in J.\ \exists X.((c \wedge \forall X.c'_j)\star \rightarrow A'_j) = \\ \qquad (((\exists X.c) \wedge \forall X.c'_j)\star \rightarrow \exists X.A'_j)\end{array}}{\exists X.(c\star \rightarrow A) = (\exists X.c)\star \rightarrow \exists X.A}
$$
$$
\text{where } A = (\ \Sigma_{i\in I} c_i\star \rightarrow A_i + \Sigma_{j\in J} c'_j\downarrow \rightarrow A'_j)
$$

$$
(B7)\quad \exists X.\mathtt{stop} = \mathtt{stop}
$$

$$
(B8)\quad \exists X.(c\downarrow \rightarrow A) = (\forall X.c)\downarrow \rightarrow \exists X.A
$$

Table 1: Axiomatization of Reactive Equality

A, we must ensure that this information has been propagated to A before applying the law. The first clause of the law's antecedent makes sure that each c_i can be written as $c \wedge c'$ for some c', which guarantees that the information can be propagated through A_i. The second clause ensures that each c'_j can be written as $c \Rightarrow c'$ for some c', which guarantees that $(B8)$ will produce the correct Ask when the $\exists X$ is pushed down. Note, however, that c'_j does not contain the information in propagatable form. Therefore, the third clause makes sure that this information has been propagated to A'_j by verifying that $(B8)$ can be legally applied to $\exists X.c'_j \downarrow \; \rightarrow A'_j$ in the given context. Think of $(B6b)$ as a one step rule with lookahead.

4.3 Identifying c-trees

Reactive equality can be coarsened further—exactly as in CCS— by taking into consideration the interpretation of "τ" as indicative of silent, internal progress.

Definition 4.3 If $\lambda \neq \tau$ and
$V \vdash \langle A,\sigma\rangle(\xrightarrow{\tau})^{\star}\langle A_1,\sigma_1\rangle \xrightarrow{\lambda} \langle A_2,\sigma_2\rangle(\xrightarrow{\tau})^{\star}\langle A',\sigma'\rangle$
then $V \vdash \langle A,\sigma\rangle \overset{\lambda}{\Longrightarrow} \langle A',\sigma'\rangle$.
If $V \vdash \langle A,\sigma\rangle(\xrightarrow{\tau})^{\star}\langle A',\sigma'\rangle$ then $V \vdash \langle A,\sigma\rangle \Longrightarrow \langle A',\sigma'\rangle$.
□

Definition 4.4 A relation $\sim$ on processes is a *V-bisimulation* iff for all processes P and Q such that $P \sim Q$, if $V \vdash P \overset{\lambda}{\Longrightarrow} P'$ then $V \vdash Q \overset{\lambda}{\Longrightarrow} Q'$ and $P' \sim Q'$, and vice versa, and if $V \vdash P \Longrightarrow P'$ then $V \vdash Q \Longrightarrow Q'$ and $P' \approx Q'$, and vice versa. □

Definition 4.5 (Reactive equivalence) Two agents A and A' in free variables V and V' respectively are said to be *reactively equivalent* (written $A \approx A'$) iff there exists a $(V \cup V')$-bisimulation $\sim$ such that $\langle A, \emptyset : V \cup V'\rangle \sim \langle A', \emptyset : V \cup V'\rangle$.

Two processes P and Q are *V-bisimilar* (written $V \vdash P \approx Q$) iff there exists a V-bisimulation $\sim$ such that $P \sim Q$. □

Reactive equivalence for cc is the counterpart for observational equivalence for cc, under the assumption that the interactions of the agent with the store are observed. Since this is not the usual notion of observation for cc processes (see, e.g. the discussion of "visible" actions in Section 4.1), we have avoided the use of the phrase "observational equivalence". However, as was the case in CCS, reactive equivalence is not a congruence for cc and for precisely the same reason: it is not respected by "+" contexts. We have:

Theorem 4.6 *Reactive equivalence is preserved by the prefixing, hiding and interleaving operations.*

The laws characterizing reactive congruence for Finitary cc are the laws in Table 1 and:

$$
\begin{array}{ll}
(A5) & A + \tau \rightarrow A = \tau \rightarrow A \\
(A6) & b \rightarrow (A_1 + \tau \rightarrow A_2) = \\
 & \quad b \rightarrow (A_1 + \tau \rightarrow A_2) + b \rightarrow A_2 \\
(A7) & b \rightarrow \tau \rightarrow A = b \rightarrow A
\end{array}
$$

Table 2: Laws characterizing reactive congruence for Finitary cc

The proof of this theorem is similar to the proof of Theorem 4.1 with the exception that the lemmas must be applied to the multiple "$\xrightarrow{\lambda}$" transitions that make up a single "$\overset{\lambda}{\Rightarrow}$", instead of only a single "$\xrightarrow{\lambda}$" transition.

Define reactive congruence for cc to be the largest congruence contained in reactive equivalance. For Finitary cc($\downarrow,\rightarrow$), it can be characterized merely by adding the appropriate "τ-laws" from CCS. (See Table 2.)

Theorem 4.7 *The laws* $(A1)$-$(A8')$ *and* $(B1)$-$(B8)$ *are complete for reactive congruence for Finitary* cc($\downarrow,\rightarrow$).

The key idea here is to use the laws $(B1)$-$(B8)$ and $(A8')$ to convert agents to labelled trees. Once this is done, Milner's proofs [Mil84] go through essentially unchanged.

Reactive congruence—the largest congruence contained in reactive equivalence—seems the finest interesting congruence for cc($\downarrow,\rightarrow$) agents. Certainly two agents that are bisimilar have the final set of observables. Bisimulation has elegant mathematical properties, and is similar to ideas on "corresponding derivations" used in logic programming [Llo84].

It is known that bisimulation congruence is too fine a congruence for CCS for "reasonable" notions of testing [BIM88]. Research by many people in the last several years has established a number of coarser congruences—such as simulation, ready simulation, refusal congruence, failures congruence etc.—and has succeeded in tying these notions to reasonable notions of testing. We believe that it will be possible to coarsen reactive congruence for cc in a fashion similar to that done for CCS. Similarly, we believe that it will be possible to extend the notions of testing for cc to take account of the richer scenarios suggested for example in [Abr89]. However we believe it will be considerably harder to tie such testing notions to the corresponding congruences in cc because of the contextual nature of communication. We look forward to considerable work in this area in the near future.

4.4 Comparisons with other work

Recently [GMS89] have proposed a notion of "reactive behaviors". A reactive behavior is just the notion of input-output record which one of us introduced in [Sar85], generalize to our setting of concurrent constraint languages: it corresponds to a "trace" of visible actions, with all information about silent transitions and hidden branches removed. They present a rather limited "full abstraction" result for $cc(\downarrow,\rightarrow)$ that ignores the subtleties associated with hidden branching. [dBK88] explores a denotational semantics based on metric domains for the Herbrand instantiation of $cc(\downarrow,\rightarrow)$. (In fact they consider "deep guards" as well, and also point out connections with the semantics of imperative concurrent languages.) Connections with some of the other work on the semantics of concurrent logic programming languages are made in [Sar89].

5 Future work, and conclusions

Within the context of concurrent logic programming research, the notion of concurrent constraint programming and the cc languages, as developed in [Sar89] and summarized here, promise to have a fundamental impact. The realization that the synchronization mechanisms which had earlier been defined in very *ad hoc*, messy ways had nothing to do with unification, substitutions, idempotence etc.—those were mere implementation details—has cleared the way for the definition and implementation of simpler, conceptually cleaner and more powerful languages. At the semantic end, it is now obvious how the technically very convoluted results in [Sar85], [GCLS88] and [UF88] can be dramatically simplified and generalized. Indeed, we are confident that the concurrent logic programming field will be subsumed by, and merge into, the more fundamental and general concurrent constraint programming field.

With the connection between cc and CCS/CSP intuitively clear, we foresee an explosion of work in this field in the future. Perhaps the best tack would be to proceed in the "specification-oriented" framework set up by [OH86] for relating various models with each other. We look forward to the presentation of such models for cc and the clarification of their relationships in the near future. However, it seems that a prime target has to be the development of a denotational semantics which is fully abstract with respect to the standard notion of observation for logic programs (that is, observe the "success set", possibly with deadlock and divergence). This was attempted in [GCLS88] but for an artificially powerful language, which does not lie in the $cc(\downarrow,\rightarrow)$ framework.

We conclude with a quotation from [Hen88]:

As this avenue of research is developed to include languages with more sophisticated features we will require a more sophisticated mathematical framework, and it is unlikely that this will be found in the existing literature. I give two examples to illustrate the point. ... As a second example, we could take the language in Hoare 1978 or the full version of CCS in Milner 1980. Here values are passed along the channels so that the actions are no longer uninterpreted. Rather, the port names now act as variable binders in the same way as λ-abstraction in the λ-calculus. An adequate semantic treatment of these languages would therefore require a mathematical framework of Σ-domains which supports equational theories, variable binders, and, presumably, equational theories incorporating these binders.

Concurrent constraint programming may well provide an alternate simple, elegant, and practical framework for the development of these ideas. The framework we have presented above already caters for a very sophisticated and useful form of communication and synchronization. Indeed, the promise of the present approach is that it will enable the construction of concrete, implemented, practical languages which provide very rich facilities for communication and which seem destined to be widely used in the future, and which are at the same time theoretically simple and admit elegant semantic treatment.

Acknowledgements. There are too many debts that the first author owes to too many people for him to even think of acknowledging them here. He has tried to make amends in the preface to [Sar89].

References

[Abr89] S. Abramsky. Tutorial on concurrency. Presented at 1989 POPL, January 1989.

[AK84] Hassan Ait-Kaci. *A lattice theoretic approach to computation based on a calculus of partially ordered type structures*. PhD thesis, University of Pennsylvania, Computer and Information Science Department, 1984.

[BFL85] Ronald J. Brachman, Richard E. Fikes, and Hector J. Levesque. *Readings in Knowledge Represenation*, chapter KRYPTON: A functional approach to Knowledge Representation, pages 411 - 429. Morgan Kaufmann Publishers, 1985.

[BHR84] 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.

[BIM88] Bard Bloom, Soren Istrail, and Albert R. Meyer. Bisimulation can't be traced: Preliminary report. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 229–239, 1988.

[CG86] K. L. Clark and S. Gregory. Parlog: parallel programming in logic. *TOPLAS*, 8(1):1–49, January 1986.

[CM88] Mani Chandy and Jay Misra. *Parallel Program Design—A foundation*. Addison Wesley, 1988.

[Dav83] Martin Davis. *Automation of Reasoning: Classical papers on computational logic*, chapter The prehistory and early history of automated deduction. Springer Verlag, 1983.

[dBK88] J.W. de Bakker and J.N. Kok. Uniform abstraction, atomicity and contractions in the comparative semantics of Concurrent Prolog. In *Proceedings of the Fifth Generation Computer Systems Conference*, December 1988.

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

[DvH^{+}88] M. Dincbas, P. van Hentenryck, H. Simonis F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the FGCS Conference*, November 1988.

[FT89a] Ian Foster and Steve Taylor. Strand: A practical parallel programming language. In *North American Logic Programming Conference*, 1989.

[FT89b] Ian Foster and Steve Taylor. *Strand: New concepts in parallel programming*. Prentice Hall, 1989.

[GCLS88] Rob Gerth, Mike Codish, Yossi Lichtenstein, and Ehud Shapiro. A fully abstract denotational semantics for Flat Concurrent Prolog. In *LICS 88*, 1988.

[GMS89] Haim Gaifman, Michael J. Maher, and Ehud Shapiro. Reactive behavior semantics for concurrent constraint logic programs. In *North American Logic Programming Conference*, October 1989. To appear.

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

[Hen88] Matthew Hennessy. *Algebraic theory of processes*. MIT Press, 1988.

[JL87] 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.

[JLL] J. Jaffar, J-.L. Lassez, and C. Lassez. Constraint logic programming: Tutorial. IEEE SLP 87 Tutorial.

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

[KYSK88] S. Kliger, E. Yardeni, E. Shapiro, and K. Kahn. The language **FCP**(:,?). In *Conference on Fifth Generation Computer Systems*, December 1988.

[Llo84] J.W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation series. Springer Verlag, 1984.

[Mah87] Michael Maher. Logic semantics for a class of committed-choice programs. In *4th International Conference on Logic Programming*. MIT Press, May 1987.

[Mah88] Michael Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. Technical report, IBM T.J. Watson Research Center, 1988.

[Mil83] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267 – 310, 1983.

[Mil84] Robin Milner. Lectures on a calculus for communicating systems. In S. Brookes, A. Roscoe, and G. Winskell, editors, *Seminar on Concurrency*, LNCS 197, 1984.

[MR88] Alan Mackworth and Ray Reiter. The logic of depiction. Technical report, University of British Columbia, October 1988.

[OH86] E.-R. Olderog and C.A.R. Hoare. Specification-oriented semantics for communicating processes. *Acta Informatica*, 23:9–66, 1986.

[Sar85] Vijay A. Saraswat. Partial correctness semantics for cp(↓,|, &). In *Proceedings of the FSTTCS Conference*, number 206, pages 347–368. Springer-Verlag, December 1985.

[Sar87a] Vijay A. Saraswat. Compiling **CP**(↓, | , &) on top of Prolog. Technical Report CMU-CS-87-174, Computer Science Department, Carnegie Mellon University, October 1987.

[Sar87b] Vijay A. Saraswat. The concurrent logic programming language **CP**: definition and operational semantics. In *Proceedings of the SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 49–62. ACM, January 1987.

[Sar88] Vijay A. Saraswat. A somewhat logical formulation of CLP synchronization primitives. In *Proceedings of LP 88*. MIT Press, August 1988.

[Sar89] Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1989. Also to be published by MIT Press, 1989.

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

[Smo88] Gert Smolka. A feature logic with subsorts. Technical Report Lilog Report 33, IBM Deutschland, May 1988.

[SRng] Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. Technical report, Xerox PARC, forthcoming.

[Str89] Strand Software Technologies Inc. *Strand 88 Users Manual*, June 1989.

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

[Ued85] K Ueda. Guarded horn clauses. Technical Report TR-103, ICOT Technical report, June 1985.

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

[UF88] K. Ueda and K. Furukawa. Tranformation rules for GHC programs. In *3d International Conference on Fifth Generation Computer Systems*, 1988.