Semantic foundations of concurrent constraint programming

Vijay A. Saraswat, Xerox PARC

Martin Rinard, Stanford University

Prakash Panangaden, McGill University

(Preliminary Report)
(To appear in POPL 91)

October 1990

Abstract

Concurrent constraint programming [Sar89,SR90] is a simple and powerful model of concurrent computation based on the notions of *store-as-constraint* and *process as information transducer*. The *store-as-valuation* conception of von Neumann computing is replaced by the notion that the store is a constraint (a finite representation of a possibly infinite set of valuations) which provides partial information about the possible values that variables can take. Instead of "reading" and "writing" the values of variables, processes may now *ask* (check if a constraint is entailed by the store) and *tell* (augment the store with a new constraint). This is a very general paradigm which subsumes (among others) nondeterminate data-flow and the (concurrent)(constraint) logic programming languages.

This paper develops the basic ideas involved in giving a coherent semantic account of these languages. Our first contribution is to give a simple and general formulation of the notion that a constraint system is a system of partial information (*a la* the information systems of Scott). Parameter passing and hiding is handled by borrowing ideas from the cylindric algebras of Henkin, Monk and Tarski to introduce diagonal elements and "cylindrification" operations (which mimic the projection of information induced by existential quantifiers).

The second contribution is to introduce the notion of determinate concurrent constraint programming languages. The combinators treated are ask, tell, parallel composition, hiding and recursion. We present a simple model for this language based on the specification-oriented methodology of [OH86]. The crucial insight is to focus on observing the *resting points*

of a process—those stores in which the process quiesces without producing more information. It turns out that for the determinate language, the set of resting points of a process completely characterizes its behavior on all inputs, since each process can be identified with a closure operator over the underlying constraint system. Very natural definitions of parallel composition, communication and hiding are given. For example, the parallel composition of two agents can be characterized by just the intersection of the sets of constraints associated with them. We also give a complete axiomatization of equality in this model, present a simple operational semantics (which dispenses with the explicit notions of renaming that plague logic programming semantics), and show that the model is fully abstract with respect to this semantics.

The third contribution of this paper is to extend these modelling ideas to the nondeterminate language (that is, the language including bounded, dependent choice). In this context it is no longer sufficient to record only the set of resting points of a process—we must also record the path taken by the process (that is, the sequence of ask/tell interactions with the environment) to reach each resting point. Because of the nature of constraint-based communication, it turns out to be very convenient to model such paths as certain kinds of closure operators, namely, bounded trace operators. We extend the operational semantics to the nondeterminate case and show that the operational semantics is fully consistent with the model, in that two programs denote the same object in the model iff there is no context which distinguishes them operationally.

This is the first simple model for the cc languages (and *ipso facto*, concurrent logic programming languages) which handles recursion, is compositional with respect to all the combinators in the language, can be used for proving liveness properties of programs, and is fully abstract with respect to the obvious notion of observation.

1 Introduction

The aim of our enterprise is simple—to develop the semantic foundations of a new paradigm for concurrent computing [Sar89,SR90].

The basic paradigm. The crucial concept underlying this paradigm is to replace the notion of *store-as-valuation* behind imperative programming languages with the notion of *store-as-constraint*. By a constraint we mean a (possibly infinite) subset of the space of all possible valuations in the variables of interest. For the store to be a constraint rather than a valuation means that at any stage of the computation one may have only partial information about the possible values that the variables can take. We take as fundamental the possibility that the state of the computation may only be able to provide partial information about the variables of interest.

This paradigm shift renders the usual notions of of (imperative) "write" and

"read" incoherent. For example, there may be no single, finitely-describable value left to return as the result of a "read" operation on a variable. Similarly, an assign operation is only capable of prescribing a fully formed, concrete value for a variable and the new value may have nothing to do with the previous value. This runs into difficulties with the notion that the store specifies some constraints that must always be obeyed by the given variables.

Instead, [Sar89] proposes the replacement of read with the notion of *ask* and write with the notion of *tell*. 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; the store is *monotonically refined*.

The idea of monotonic update is central to the theoretical treatment of I-structures in Id Nouveau [JPP89]. I-structures were introduced in order to have some of the benefits of in-place update without introducing the problems of interference. It is interesting that the concurrent constraint paradigm can be seen as arising as a purification of logic programming [Sar89], an enhancement to functional programming and as a generalization of imperative programming. From the viewpoint of dataflow programming, the concurrent constraint paradigm is also a generalization in that the flow of information between two processes is *bidirectional*. It might relate to the goal of a more symmetrical theory of computation advocated by Girard [Gir87,Gir89].

Central to our notion of constraint system is a theory of partial information and entailment between partial information. Such a theory exists in the form of Scott's treatment of information systems [Sco82]. In our case it is natural to imagine two concurrent processes imposing inconsistent constraints on the store. Thus, we need to represent the possibility of inconsistent information.

The approach of this paper makes the possibilities for concurrency 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.[1] If desired,

[1] Which of course, is not to say that systems with changable state are not describable in the

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 these ideas are extremely simple, they conceal a powerful programming paradigm which derives its strength from the versatility of its communication mechanism. It is not possible in a short introduction to describe the many interesting communication schemes possible here. We shall just attempt to indicate the basic ideas and refer the reader to more detailed treatments such as [Sar89].

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 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 a lot of computation can be expressed merely by posting constraints.

Even in a constraint system (Herbrand) over a domain as mundane as that of finite trees, such a communication scheme leads to many interesting idioms such as incomplete messages, short-circuits, difference-lists, "messages to the future" etc. [Sar89]—the techniques that have colloquially been referred to as

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[SKL90]) there is considerable hope that such mechanisms for representing state change can be competitive in performance with traditional assignment-based techniques.

stemming from "the power of the logical variable".[2] 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. This makes for an elegant form of dynamic reconfigurability, of a kind which is difficult to achieve in a simple way within frameworks such as those of CCS and CSP. The practicality of this framework is attested to by the fact that several implemented cc languages are now available, including at least one commercial implementation [FT89].

This paper focusses on the use of constraints for communication and control in concurrent constraint programming languages. It is worth pointing out that the concurrent constraint programming framework is, however, concerned with much more than just concurrency—it takes the first step towards a general architecture for computing with constraints that is dependent only on the form of constraint systems, not on their particular details (following [JL87]). As such, it provides one coherent attempt to articulate the vision of *constraint programming* manifest in the work of Sutherland, Sussman, Steele, Borning and others. In particular, the cc framework is also concerned with many other combinators for introducing and controlling logic-programming style "search nondeterminism".[3]

One of the goals of our research is defining a general class of *constraint systems* to which the concurrent constraint paradigm applies. A beginning has been made in the present paper. The presence of constraint systems in computational and engineering problems is very widespread. For example, several applications to AI are usefully viewed in terms of constraints. A general enough definition would allow one to define constraint systems as over data types as diverse as finite trees, streams, intervals of rational numbers, various types of functions spaces and data types derived from knowledge rerpesentation applications. Indeed the design of constraint systems of use in computational systems is limited only by our imaginations. It is not difficult to consider any data-structure of interest in computer science — enumerate types, records, bags, arrays, hash-tables, graphs, binary search trees — and devise constraint systems of interest over them which can usefully and interestingly be embedded within a cc language.

Generality. The cc framework is parametrized by an arbitrary constraint system. This schematic treatment brings with it all the usual advantages: results need to be proven only once and are immediately applicable to all members of

[2]The kind of constraint-based communiation schemes we are describing here are the essence of computation in the logic programming style. Indeed, the origins of the cc framework are in concurrent logic programming and in the notion of constraint logic programing introduced by [JL87,Mah87]. See [Sar89] for further details.

[3]Throughout this paper when we talk of the "cc languages" we shall mean the cc languages with Ask and Tell as the only primitive constraint-related operations. Other coherent and very useful primitives can be (and have been, in [Sar89]) defined, but they are outside the scope of this paper.

the class. In particular the models we develop in this paper are in fact a class of models, for a large class of programming languages.

We place very few restrictions on the nature of a constraint system: we demand only that there be some system of partial information, some notion of what it means for various pieces of partial information to be true about the same objects (the notion of *consistency*), and what it means for certain pieces of information to always hold, given that some other pieces of information must also hold (the notion of entailment).

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 [BB90], due to Boudol and Berry, and mobile processes, due to Milner and his co-workers [MPW89]. 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 [Mil90,JP90].

1.1 Contributions of this paper

The central task of this paper is to develop the semantic foundations of the programming paradigm discussed above. Towards this end, we formalize the basic notion of a constraint system, and present operational and denotational semantics for the determinate and nondeterminate `cc` languages. The next several paragraphs discuss each of these contributions in detail.

Constraint systems. We formalize the notion of constraint system, generalizing and simplifying previous treatments in [Sar89,JL87,SR90]. The basic insight is to treat constraint systems as systems of partial information, in the style of Dana Scott's development of information systems, together with operators to express hiding of information. From a programming point of view such operations allow the introduction of parameter passing in constraint programming languages, and the introduction of "local variables" within procedures.

Philosophy of modeling. The models developed in this paper are based on the philosophy of modeling discussed in [OH86,Hoa90]. In the next few paragraphs we summarize the basic ideas and ontological commitments made in this style.

Crucial to this philosophy is the identification of the notion of *observation* of a process. A process is then equated with the set of observations that can be made of it. The set of processes is identified by presenting a collection of naturally motivated closure conditions on sets of observations.

It is important that the observations of a process include all ways in which the process can "go wrong", that is, fail to meet obligations imposed on it by the environment. Once a process "goes wrong", further detailed modeling of the process is irrelevant, since the emphasis of this approach is on guaranteeing that as long as the environment honors its obligations to the process, the process cannot "go wrong". This ontological commitment is usually captured in the slogan "divergence is catastrophic", or, "anything can happen" once the process goes wrong.

One way in which the process can go wrong is by engaging in "internal chatter", that is, in an infinite sequence of internal actions (possibly continually producing output in the meantime), without asking the environment for some input (and thereby giving the environment the ability to control its execution by denying it that input). Within the `cc` framework, another way in which a process can "go wrong" is if it causes the shared store to become inconsistent — for in such a state the process is again completely uncontrollable from the environment. No action that the environment can take can then influence the course of execution of the process, since the inconsistent store can answer *any* ask request. Hence the process is free to follow any branch that it could potentially follow, without more input from the environment. In particular, if the process recursively calls another process, then it can engage in an infinite execution sequence because every ask operation guarding a recursive procedure call will be answered successfully.

This suggests that in the semantic model the process that can produce `false` should not be distinguished form the process that can engage in an infinite execution sequence; both such processes should be treated as "catastrophic". This is indeed the approach followed in the main body of the paper. However, other alternative treatments of failure and divergence are possible, and in Section 3 we shall indicate some of the possibilities and how they can be treated.

Determinate constraint programming. Sixteen years ago, Kahn gave a theory of determinate data-flow [Kah74]. Today this theory (with some extensions, e.g. to concrete data structures) 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 program-

ming 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 nondeterminate concurrent logic programming, they are all both inherently determinate and usefully expressible in determinate contexts.

This paper presents a simple theory of determinate computation which can model all of the above idioms and more: our theory preserves the essential features of Kahn's theory while substantially extending its domain of applicability. From the contraint programming point of view it is useful and interesting to focus on the determinate subset because a mathematically much simpler treatment can be given, without sacrificing any essential novelty—the major semantic and computational ideas underlying the cc paradigm can already be illustrated in the determinate case.

We present a simple algebraic syntax for determinate concurrent constraint programs. We also present an operational semantics based on a labelled transition system in which a configuration is just an agent and a label is a pair of constraints (the store in which the transition is initiated and the store in which it terminates). The transition system is able to completely avoid using substitutions and "variable-renamings", thereby considerably simplifying the treatment. Past theoretical treatments of (constraint) logic programming have had to use various *ad hoc* techniques for dealing with this problem.

The denotation of a process is taken to be a set of constraints (the "resting points" of the process) satisfying certain properties. Various combinators are defined on such processes, corresponding to ask and tell operations, to running two agents in parallel, and to creating a new "local" communication channel. We show that the denotational semantics is fully abstract with respect to the operational semantics. We also give a sound and complete axiomatization of equality for finite programs, so that finite program equivalence can be established purely by equational reasoning. We also develop a slightly different model in which limits of (fair) infinite execution sequences are observed.

Models for nondeterminate cc languages. The denotation of a nondeterminate process is 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*. The denotation of a process is the set of all its failures. The set of all processes is identified by presenting some naturally motivated closure conditions on sets of failures, following [Jos90]. (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. In par-

ticular, we believe that this treatment gives a completely satisfactory account of the semantics of concurrent logic programming languages.

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 [Sar85,Lev88,GL90], [GCLS88], [GMS89] 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 various kinds of closure operators. To capture various operational notions of interest, we introduce the concept of *trace operators*, and *bounded closure operators*, and present some portions of their theory relevant to this paper.

We also establish a very simple relationship between the nondeterminate and determinate semantics, showing that the two semantics are essentially identical for determinate programs over finitary constraint systems.

In summary, we present 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.

1.2 Related work

The basic concepts of concurrent constraint programming were introduced in [Sar88,Sar89]. Subseqently, we developed an operational semantics and a bisimulation semantics in [SR90]. This line of research extends and subsumes (for all intents and purposes) earlier work on the semantics of concurrent logic programming languages.[4]

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 [Lin85,DL86,ANP89,JPP89]. 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 [JPP89], which discusses a subset of the functional programming language *Id Noveau* with "logical" arrays [ANP89]. Their semantic treatment also identifies a program as a

[4]The semantics presented in this paper does not account for the language Concurrent Prolog studied in [GCLS88]; 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 [KYSK88,GMS89].

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 [Hoa89], in a setting related to Unity [CM88]. 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 [SR90,GMS89,GL90,dBP90a]. 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 studied in the set-up of logic programming languages in [Mah87], but no characterizations of the kind we provide here were given.

[GMS89] 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 [dBP90a,dBP90b] 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 [dBP90b] is not compositional with respect to the choice operator.

Our development follows closely Mark Josephs' treatment of receptive processes in [Jos90]. 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.

2 Constraint Systems

Our presentation here is simplified and generalized from the presentation in [SR90]. More details may be found in [SPRng].

What do we have when we have a constraint system? First, of course, there must be a *vocabulary* of assertions that can be made about how things can be — each assertion will be a syntactically denotable object in the programming language. Postulate then a set D of *tokens*, each giving us partial information about certain states of affairs. At any finite state of the computation, the program will have deposited some finite set u of such tokens with the embedded constraint-solver and may demand to know whether some other token is *entailed* by u. Postulate then a *compact* entailment relation $\vdash \subseteq pD \times D$ (pD is the set of finite subsets of D), which records the inter-dependencies between tokens. The intention is to have a set of tokens v entail a token P just in case for every state of affairs for which we can assert every token in v, we can also assert P. This leads us to:[5]

Definition 2.1 A *simple constraint system* is a structure $\langle D, \vdash \rangle$, where D is a non-empty (countable) set of *tokens* or *(primitive) constraints* and $\vdash \subseteq pD \times D$ is an *entailment relation* satisfying (where pD is the set of *finite* subsets of D:

C1 $u \vdash P$ whenever $P \in u$, and,

C2 $u \vdash Q$ whenever $u \vdash P$ for all $P \in v$, and $v \vdash Q$.

Extend $\vdash$ to be a relation on $pD \times pD$ by: $u \vdash v$ iff $u \vdash P$ for every $P \in v$. Define $u \approx v$ if $u \vdash v$ and $v \vdash u$. □

Of course, in any implementable language, $\vdash$ must be decidable—and as efficient as the intended class of users of the language demand. Compactness of the entailment relation ensures that one has a semi-decidable entailment relation. If a token is entailed, it is entailed by a finite set and hence if entailment holds it can be checked in finite time. If the store does not entail the constraint it may not be possible for the constraint solver to say this at any finite stage of the computation.

Such a treatment of systems of partial information is, of course, well-known, and underlies Dana Scott's information systems approach to domain theory [Sco82]. A simple constraint system is just an information system with the consistency structure removed, since it is natural in our setting to conceive of the possibility that the execution of a program can give rise to an inconsistent state of affairs.

[5] In reality, the systems underlying most concrete concurrent constraint programming languages have slightly more structure to them, namely they are ask-and-tell constraint systems [Sar89]. The additional structure arises because it is possible to state at the level of a constraint system that the imposition of certain constraints can be delayed until such time as some associated constraint is entailed by the store ("implicit Ask-restriction"). However, this additional structure is not crucial, and can be easily handled by extending the techniques presented in this paper.

Following standard lines, states of affairs (at least those representable in the system) can be identified with the set of all those tokens that hold in them.

Definition 2.2 The *elements* of a constraint system $\langle D, \vdash \rangle$ are those subsets c of D such that $P \in c$ whenever $u \subseteq_f c$ (i.e. u is a finite subset of c) and $u \vdash P$. The set of all such elements is denoted by $|D|$. For every $u \subseteq_f D$ define $\bar{u} \in |D|$ to be the set $\{P \in D \mid u \vdash P\}$. □

As is well known, $(|D|, \subseteq)$ is a complete algebraic lattice, the compactness of $\vdash$ gives us algebraicity of $|D|$, with least element $\mathtt{true} = \{P \mid \emptyset \vdash P\}$, greatest element D (which we will mnemonically denote $\mathtt{false}$), glbs (denoted by $\sqcap$) given by intersection and lubs (denoted by $\sqcup$) given by the closure of the union. The lub of chains is, however, just the union of the members in the chain. The finite elements of $|D|$ are just the elements generated by finite subsets of D; the set of such elements will be denoted $|D|_0$. We use a,b,c,d and e to stand for elements of $|D|$; $c \geq d$ means $c \vdash d$. Two common notations that we use when referring to the elements of $|D|$ or $|D|_0$ are $\uparrow c = \{d | c \leq d\}$ and $\downarrow c = \{d | d \leq c\}$.

The alert reader will have noticed that the constraint system need not generate a *finitary*[6] algebraic lattice since, in general, Scott information systems do not generate finitary domains. Indeed many common constraint systems are not finitary even when the data type that they are defined over is finitary. For the interpretation of determinate concurrent constraint programs we do not need the constraint system to be finitary but we do for the nondeterminate case. If we drop the requirement that the entailment relation is compact we will generate, in general, lattices that are not algebraic. We always need the entailment relation to be compact since we do not know, as yet, whether these ideas can be extended to nonalgebraic constraint systems.

In what follows, by a *finite constraint* we shall mean a finite set of tokens. We also take the liberty of confusing a finite constraint u with $\bar{u} \in |D|_0$.

Example 2.1 Generating constraint systems.
For any first-order vocabulary $\mathcal{L}$, and countably infinite set of variables $\mathtt{Var}$, take D to be an arbitrary subset of open $(\mathcal{L}, \mathtt{Var})$-formulas, and $\vdash$ to be the entailment relation with respect to some class Δ of $\mathcal{L}$-structures. That is, $\{P_1, \ldots, P_n\} \vdash Q$ iff for every structure $\mathcal{M} \in \Delta$, an $\mathcal{M}$-valuation realizes Q whenever it realizes each of $P_1, \ldots, P_n$. Such a $\langle D, \vdash \rangle$ is a simple constraint system. □

Example 2.2 The Kahn constraint system.
More concretely, let us define the Kahn constraint system $\mathcal{D}(\mathcal{B}) = \langle D, \vdash_{\mathcal{D}} \rangle$ underlying data-flow languages [Kah74], for $\mathcal{B} = \langle B, \vdash_{\mathcal{B}} \rangle$ so some underlying constraint system on a domain of data elements, E. Let $\mathcal{L}$ be the vocabulary consisting of the predicate symbols $= /2, c/1$ and the function symbols $f/1, r/1, a/2, \Lambda/0$. Postulate an infinite set $(X, Y \in)\mathtt{Var}$ of variables. Let the

[6] This means that a finite element dominates only finitely many elements.

set of tokens D consist of *atomic* $(\mathcal{L}, \mathtt{Var})$ formulas. Let Δ consist of the single structure with domain of interepretation B^ω the set of (possibly infinite) sequences over B, (including the empty sequence Λ) and interpretations for the symbols in $\mathcal{L}$ given by:

- $=$ is the equality predicate,
- **c** is the predicate that is true of all sequences except Λ.
- **f** is the function which maps Λ to Λ, and every other sequence s to the unit length sequence whose first element is the first element of s,
- **r** is the function which maps Λ to Λ, and every other sequence s to the sequence obtained from s by dropping its first element,
- **a** is the function which returns its second argument if its first argument is Λ; otherwise it returns the sequence consisting of the first element of its first argument followed by the elements of the second argument.

Now, we can define $\vdash_{\mathcal{D}}$ by:

$$\{c_1, \ldots, c_n\} \vdash_{\mathcal{D}} c \iff \Delta \vdash_{\mathcal{D}} (c_1 \wedge \ldots c_n \Rightarrow c)$$

thus completing the definition of the constraint system $\mathcal{D} = \langle D, \vdash_{\mathcal{D}} \rangle$.

Note that in this constraint system the set of elements are not finitary. The constraint $X = Y$, which is finite, entails infinitely many constraints of the form $f(r^n(X)) = f(r^n(Y))$. Since the constraint system has a compact entailment relation we will have algebraicity. In the lattice generated by the entailment closed sets of tokens the set consisting of the entailment closure of $\{X = Y\}$ will contain all the tokens of the form $f(r^n(X)) = f(r^n(Y))$; the set consisting of all the latter, however, will not contain $X = Y$. It is possible to define a variant system that is finitary. The data type of streams is, of course, finitary. □

Example 2.3 The Herbrand constraint system.
We describe this example quickly. There is an ordinary first-order language L with equality. The tokens of the constraint system are the atomic propositions. Entailment can vary depending on the intended use of the predicate symbols but it must include the usual entailment relations that one expects from equality. Thus, for example, $f(X, Y) = f(A, g(B, C))$ must entail $X = A$ and $Y = g(B, C)$. If equality is the only predicate symbol then the constraint system is finitary. With other predicates present the finitariness of the lattice will depend on the entailment relation. □

Example 2.4 Rational intervals.
The underlying tokens are of the form $X \in [x, y]$ where x and y are rational

numbers and the notation $[x, y]$ means the closed interval between x and y. We assume that every such membership assertion is a primitive token.

The entailment relation is the one derived from the obvious interpretation of the tokens. Thus, $X \in [x_1, y_1] \vdash X \in [x_2, y_2]$ if and only if $[x_1, y_1] \subseteq [x_2, y_2]$. Whether this yields a compact entailment relation is a slightly delicate issue. If we assume the usual definition of intersection and unions of infinite families of intervals, we will definitely not have a compact entailment relation. For example, since $\bigcap_{n>0}[0, 1 + 1/n] = [0, 1]$, we would have $\{X \in [0, 1 + 1/n] | n > 0\} \vdash (X \in [0, 1])$ but no finite subset of $\{X \in [0, 1+1/n] | n > 0\}$ would entail $X \in [0, 1]$. We may take the definition $u \vdash X \in [x, y]$ to mean that u must be a finite collection of intervals. In this case the entailment relation is, by definition, compact and the lattice generated is algebraic. It will, however, appear slightly peculiar with respect to one's normal expectations about intervals. The join of a family of assertions involving membership of X in a nested family of intervals will not yield an assertion about membership in the intersection of the family. Instead there will be a new element of $|D|$ that sits below the intersection. Thus, for example, the join $\bigsqcup_{n>0} X \in [0, 1 + 1/n]$ will not be $X \in [0, 1]$ but rather a new element that sits below $X \in [0, 1]$. Clearly the lattice is not finitary but the entailment relation is compact and the lattice is indeed algebraic. It is worth noting that this example shows that we can model determinate computations over domains that are not *incremental*[7]. In fact we have an *order-dense subset*[8]. We can work with such a lattice when we model the determinate language but, as yet, we cannot model the nondeterminate languages over such constraint systems. It is known that the closure operators over a lattice with an order-dense subset cannot form an algebraic complete partial order[9]. Thus, the extension of these ideas to higher-order programming will be challenging. □

Hiding in constraint systems. Any reasonable programming language supports modularity by providing a notion of hiding the internal structure of an agent from its context. We support this hiding with a family of hiding operators on the underlying constraint system; these operators capture the notion of projecting away information. In this we use the axiomatization of cylindric algebra[HMT71]. In future research we plan to give a more principled account of hiding and of the choice of axioms using notions from categorical logic.

Definition 2.3 A *cylindric constraint system* is a structure $\langle D, \vdash, \mathtt{Var}, \{\exists_X \mid X \in \mathtt{Var}\}\rangle$ such that:

- $\langle D, \vdash\rangle$ is a simple constraint system,

[7] A prime interval is a pair of finite elements such that there is no finite element properly between them. An incremental domain is one in which between every two related finite elements there is a finite sequence of prime intervals that interpolates between them.

[8] This means that between any two elements there is a another distinct element.

[9] We are indebted to Michael Huth for pointing this out.

- `Var` is an infinite set of *indeterminates* or *variables*,
- For each variable $X \in \texttt{Var}$, $\exists_X : pD \to pD$ is an operation satisfying:

  E1 $u \vdash \exists_X u$

  E2 $u \vdash v$ implies $\exists_X u \vdash \exists_X v$

  E3 $\exists_X(u \cup \exists_X v) \approx \exists_X u \cup \exists_X v$,

  E4 $\exists_X \exists_Y u \approx \exists_Y \exists_X u$

□

For every variable X, $\exists_X$ is extended to be a continuous function from $|D| \to |D|$ in the obvious way:

$$\exists_X c = \{P \mid \exists_X u \vdash P, \text{ for some } u \subseteq_f c\}$$

Example 2.5 Let the token set consist of some subclass of $(\mathcal{L}, \texttt{Var})$ formulas closed under existential quantification of finite conjunctions. Each operator $\exists_X$ is then interpreted by the function which maps each finite set $\{P_1, \ldots, P_n\}$ of tokens to the set of tokens $\{\exists X.P_1 \wedge \ldots \wedge P_n\}$. It is easy to see that the four conditions above will be satisfied. □

Diagonal elements. For all useful programming languages in this framework, it is necessary to consider procedures with parameter passing. In usual logic programming languages, parameter passing is supported by using substitutions. We use a trick due to Tarski and his colleagues. For the class of constraint systems discussed above, this trick can be illustrated by providing, as tokens, the "diagonal" formulas $X = Y$, for $X, Y \in \texttt{Var}$. Now, the formula $\phi[Y/X]$ is nothing else but the formula $\exists X.(X = Y) \wedge \phi$. More generally, for an arbitrary constraint system, we can axiomatize the required properties of the diagonal elements, following [HMT71]. We demand that the token set D contain, for every pair of variables $X, Y \in \texttt{Var}$, the token d_{XY} satisfying the properties:

D1. $\emptyset \vdash d_{XX}$

D2. if $X \neq Y$, $\{d_{XY}\} \approx \exists_Z \{d_{XY}, d_{YZ}\}$

D3. $\{d_{XY}\} \cup \exists_X (u \cup \{d_{XY}\}) \vdash u$

The defect of this axiomatization is that it "appears out of thin air". In particular, categorical logic, as expounded by Lambek and Scott [LS86] has a thorough analysis of variables and also of variable-free calculi and the relationship between them. If we cast the notion of constraint system in categorical terms, we would be able to use the vast body of results about categorical logic [LS86] in the course of our development of the concept of constraint system. The axioms for hiding would emerge from fundamental principles. Investigations into

catgorical logic suggest that the logic implicit in our treatment is a form of coherent logic [MR97][10].

3 The determinate language

... the most important observations are those which can be made only indirectly by a more or less elaborate experiment ... a successful choice of the right kind of indirect observation can provide a remarkably coherent and general explanation of a wide range of diverse phenomena.

— C.A.R. Hoare (1990)

Our basic semantic insight is that the crucial observation to make of a process is the set of its *resting points*. A process is an information processing device that interacts with its environment via a shared constraint representing the known common information. A resting point of a process is a constraint c such that if the process is initiated in c, it will eventually halt without producing any more information.[11]

While the basic idea is the same, the semantics for the nondeterminate language is significantly more complex than the semantics for the determinate language. The reason is simple. For the determinate language, it turns out to be *sufficient* to store with a process just the set of its resting points, as we now discuss.

To be determinate, the process must always produce the same output constraint when given the same input constraint. We can therefore identify a process with a function:

$$f : |D|_0 \to |D|_0$$

This function maps each input c to `false` if the process when initiated in c engages in an infinite execution sequence, and to d if the process ultimately quiesces having upgraded the store to d. [12] It turns out that *there is sufficient information in the resting points of the process to uniquely determine its associated function.* This property highlights the semantic simplicity of the Ask-and-Tell communication and synchronization mechanism.

Let f be the operator on $|D|_0$ corresponding to a given process. Now, the only way in which this process can affect the store is by adding more information to it. Therefore, f must be *extensive*:

[10] We are indebted to Robert Seely and Phil Scott for this observation.

[11] Strictly speaking, this is not true. In the continuous closure operator semantics we consider later in this section, a resting point of a process is a constraint c such that if the process were to be initiated in c, it would not be able to ouptut any new information without receiving some more information from the outside. The point is that the process may engage in an infinite execution sequence in c, as long as it produces no new information.

[12] Thus we shall confuse the process that on input c produces `false` and halts with the process that on input c engages in an infinite execution sequence. Recall from the introduction that this is in tune with the specification-oriented approach to semantics.

$$\forall c.c \leq f(c) \tag{1}$$

Second, the store is accessible to the process as well as its environment: therefore if on input c a process produces output d and halts, it must be the case that d is a resting point, that is, on input d the process cannot progress further (because if it could, then it wouldn't have stopped in d). That is, f must be idempotent:

$$\forall c.f(f(c)) = f(c) \tag{2}$$

Finally, consider what happens to the output of such a function when the information content of the input is increased. If the invocation of the function corresponds to the imposition of a constraint, then as information in the input is increased, information in the output should not decrease. Such a function is called *monotone*:

$$\forall c, d.c \leq d \Rightarrow f(c) \leq f(d) \tag{3}$$

An operator over a partial order that is extensive, idempotent and monotone is called a *closure operator* (or, more classicaly, a *consequence* operator, [Tar56]). Closure operators are extensively studied in [GKK+80] and enjoy several beautiful properties which we shall exploit in the following. Within computer science, continuous closure operators have been used in [Sco76] to characterize data-types.

We list here some basic properties. The most fundamental property of a closure operator f over a lattice E is that it can be represented by its range $f(E)$ (which is the same as its set of fixed points). f can be recovered from $f(E)$ by mapping each input to the least element in $f(E)$ above it. This is easy to see: by extensiveness, $f(c)$ is above c; by idempotence, $f(c)$ is a fixed-point of f, and by monotonicity, it is the least such fixed-point. This representation is so convenient that in the following, we shall often confuse a closure operator with its range, writing $c \in f$ to mean $f(c) = c$. In fact, a subset S of E is the range of a closure operator on E if every lower-bounded subset [13] of S has a greatest lower bound which is in S. The range of every closure operator is non-empty, since it must contain $\texttt{false} = \sqcap\emptyset$.

Partial order. For a closure operator f over $|D|_0$, let df be the *divergences* of f, that is, those inputs in which the process diverges. As discussed above, the divergences are exactly those constraints which are mapped to `false` by f, that is, $f^{-1}(\texttt{false})$. The partial order on determinate processes of interest to us will be based partly on the processes' divergences. The intention is that a process f can be improved to a process g iff the divergences of g are contained in those of f and at every point in the convergences of f, f and g take on identical values. In terms of fixed points, this yields:

[13] A subset of E is lower-bounded if it has a lower bound in E.

Definition 3.1 The divergence order on closure operators over $|D|_0$ is given by:

$$f \leq g \iff f \subseteq g \subseteq f \cup df$$

□

The bottom element of this partial order is $\{\texttt{false}\}$ which diverges everywhere. It is not hard to see that this order is complete, with limits of chains given by unions of the set of fixed-points.

3.1 Process Algebra

In this section we develop a simple language, the determinate `cc` language, for expressing the behavior of concurrent determinate constraint processes. We consider agents constructed from tells of finite constraints, asks of finite constraints, hiding operators, parallel composition and procedure calls. Throughout this section we shall assume some fixed cylindrical constraint system (with diagonal elements) D. As usual, $|D|$ denotes this constraint system's set of elements, while $|D|_0$ denotes its set of finite elements.

We also define a quartic *transition* relation

$$\longrightarrow \subseteq Env \times (|D|_0 \times |D|_0) \times A \times A$$

which will be used to define the operational semantics of the programming language. (Here Env is the set of all partial functions from procedure names to (syntactic) agents.) Rather than write $\langle \rho, (c,d), A, B \rangle \in \longrightarrow$ we shall write $\rho \vdash A \xrightarrow{(c,d)} B$ (omitting the "$\rho \vdash$" if it is not relevant) and take that to mean that when initiated in store c, agent A can, in one uninterruptible step, upgrade the store to d, and subsequently behave like B. In the usual SOS style, this relation will be described by specifying a set of axioms, and taking the relation to be the smallest relation satisfying those axioms.

The syntax and semantics of the determinate language are given in Table 1. We discuss these semantic definitions in this section. For purposes of exposition we assume that procedures take exactly one variable as a parameter and that no program calls an undefined procedure. We also systematically confuse the syntactic object consisting of a finite set of tokens from D with the semantic object consisting of this set's closure under $\vdash$.

Tells. The process c augments its input with the finite constraint c. Thus it behaves as the operator $\lambda x.c \sqcup x$, which in terms of fixed points, is just:

$$c = \{d \in |D|_0 \mid d \geq c\}$$

The operational behavior of c is described by the transition axiom:

$$c \xrightarrow{(d, c \sqcup d)} \texttt{true} \quad (c \neq \texttt{true}) \tag{4}$$

Syntax.

$$P ::= D.A$$
$$D ::= \epsilon \mid p(X) :: A \mid D.D$$
$$A ::= c \mid c \rightarrow A \mid A \wedge A \mid \exists X A \mid p(X)$$

Semantic Equations.

$$\mathcal{A}(c)e = \{d \in |D|_0 \mid d \geq c\}$$
$$\mathcal{A}(c \rightarrow A)e = \{d \in |D|_0 \mid d \geq c \Rightarrow d \in \mathcal{A}(A)e\}$$
$$\mathcal{A}(A \wedge B)e = \{d \in |D|_0 \mid d \in \mathcal{A}(A)e \wedge d \in \mathcal{A}(B)e\}$$
$$\mathcal{A}(\exists X A)e = \{d \in |D|_0 \mid \exists c \in \mathcal{A}(A)e . \exists_X d = \exists_X c\}$$
$$\mathcal{A}(p(X))e = \exists_\alpha(d_{\alpha X} \sqcup e(p))$$
$$\mathcal{D}(\epsilon)e = e$$
$$\mathcal{D}(p(X) :: A.D) = \mathcal{D}(D)e[p \mapsto \exists_X(d_{\alpha X} \sqcup \mathcal{A}(A)e)]$$
$$\mathcal{P}(D.A) = \mathcal{A}(A)(\ \underline{\texttt{fix}}\ \mathcal{D}(D))$$

Above, c ranges over basic constraints, that is, finite sets of tokens. P is the syntactic class of programs, D is the syntactic class of sequences of procedure declarations, and A is the syntactic class of agents. α is some variable in the underlying constraint system which is not allowed to occur in user programs. (It is used as a dummy variable during parameter passing.) e maps procedure names to processes, providing an environment for interpreting procedure calls. We use the notation $c \sqcup f$ to stand for $\{c \sqcup d \mid d \in f\}$.

Table 1: Denotational semantics for the Ask-and-Tell Determinate cc languages

corresponding to adding the information in c to the shared constraint in a single step.[14]

Asks. Let c be a constraint, and f a process. The process $c \rightarrow f$ waits until the store contains at least as much information as c. It then behaves like f. Such a process can be described by the function $\lambda x.if\ x \geq c\ then f(x)\ else\ c$. In terms of its range:

$$c \rightarrow f = \{d \in |D|_0 \mid d \geq c \Rightarrow d \in f\}$$

The ask operation is monotone and continuous in its process argument. It satisfies the laws:

$$(L1) \quad c \rightarrow d = c \rightarrow (c \wedge d)$$
$$(L2) \quad c \rightarrow \texttt{true} = \texttt{true}$$
$$(L3) \quad c \rightarrow d \rightarrow A = (c \sqcup d) \rightarrow A$$
$$(L4) \quad \texttt{true} \rightarrow A = A$$

The rule for $c \rightarrow A$ is:

[14] Throughout the rest of this paper, depending on context, we shall let c stand for either a syntactic object consisting of a finite set of tokens, the constraint obtained by taking the closure of that set under $\vdash$, the (semantic) process that imposes that constraint on the store, or the (syntactic) agent that imposes that constraint on the store.

$$c \rightarrow A \xrightarrow{(d,d)} A \quad \text{if} \quad d \geq c \tag{5}$$

Parallel composition. Consider the parallel composition of two processes f and g. Suppose on input c, f runs first, producing $f(c)$. Because it is idempotent, f will be unable to produce any further information. However, g may now run, producing some more information, and enabling additional information production from f. The system will quiesce exactly when *both* f and g quiesce. Therefore, the set of fixed points of $f \wedge g$ is exactly the intersection of the set of fixed points of f with the set of fixed points of g:

$$f \wedge g = f \cap g$$

It is straightforward to verify that this operation is well-defined, and monotone and continuous in both its arguments.

While the argument given above is quite simple and elegant,[15] it hides issues of substantial complexity. The basic property being exploited here is the *restartability* of a determinate process. Suppose an agent A is initiated in a store c, and produces a constraint d before quiescing, leaving a "residual agent" B to be executed. To find out its subsequent behavior (e.g., to find out what output it would produce on a store $e \geq d$), it is *not* necessary to maintain any explicit representation of B in the denotation of A. Rather, the effect of B on input $e \geq d$ can be obtained simply by running the *original* program A on e**!** Indeed this is the basic reason why it is possible to model a determinate process accurately by just the set of its resting points.

As we shall see in the next section, this restartability property is not true for nondeterminate processes. Indeed, we cannot take the denotation of a process to be a function (nor even a relation) from finite stores to finite stores; rather it becomes necessary to also preserve information about the *path* (that is, the sequence of ask/tell interactions with the environment) followed by the process in reaching a resting point.

From this definition, several laws follow immediately. Parallel composition is commutative, associative, and has an identity element.

$$\begin{array}{ll} (L5) & A \wedge B = B \wedge A \\ (L6) & A \wedge (B \wedge C) = (A \wedge B) \wedge C \\ (L7) & A \wedge \mathtt{true} = A \end{array}$$

Telling two constraints in parallel is equivalent to telling the conjunction.

[15] And should be contrasted with most definitions of concurrency for other computational models which have to fall back on some sort of interleaving of basic actions.