Nordic Journal of Computing

A LOGICAL VIEW OF CONCURRENT CONSTRAINT PROGRAMMING

Nax P. Mendler*
Department of Mathematics
University of Ottawa
Ottawa, Ontario, Canada

Prakash Panangaden†
School of Computer Science
McGill University
Montreal, Quebec, Canada

P.J. Scott‡
Department of Mathematics
University of Ottawa
Ottawa, Ontario, Canada

R.A.G. Seely§
Department of Mathematics
McGill University
Montreal, Quebec, Canada

Abstract. Concurrent Constraint Programming (CCP) has been the subject of growing interest as the focus of a new paradigm for concurrent computation. Like logic programming it claims close relations to logic. In fact CCP languages *are* logics in a certain sense that we make precise in this paper. In recent work it was shown that the denotational semantics of determinate concurrent constraint programming languages forms a fibred categorical structure called a hyperdoctrine, which is used as the basis of the categorical formulation of first order logic. What this shows is that the combinators of determinate CCP can be viewed as logical connectives. In this paper we extend these ideas to the operational semantics of such languages and thus make available similar analogies for a much broader variety of languages including indeterminate CCP languages and concurrent block-structured imperative languages.

Key words: Concurrent constraint programming, simulation, logic, categories, hyperdoctrines

* Research supported in part by NSERC and Ontario-Quebec Bilateral Exchange, Ontario Ministry of Education.
† Research supported in part by NSERC and FCAR.
‡ Research supported in part by NSERC and Ontario-Quebec Bilateral Exchange, Ontario Ministry of Education.
§ Research supported in part by NSERC and FCAR.

Received June 1998.

CR Classification: Each paper should provide the CR Classification (see the appropriate issue of *Computing Reviews*).

1. Introduction

Concurrent constraint programming (CCP) has emerged as an important paradigm within the realm of asynchronous computation, with close ties to logic. Though logic and computation have always been related, the synergy between these two fields has been sharply increasing in the last decade. For example the proofs-as-programs paradigm [Girard 1989], long exemplified for functional languages by the simply-typed lambda calculus, has grown to include a whole family of typed lambda calculi, linear calculi and more recently typed process calculi, through Abramsky's work on "proofs-as-processes" and linear realizability algebras [Abramsky 1991, 1993]. However there is another relationship between logic and computation that one sees in concurrent constraint programming, and logic programming in general: not through the proofs-as-programs view but rather through programs as "proof search" [**?** **?**].

In this paper we present yet another connection between logic and computation that may be summarized by the slogan "Concurrency *is* Logic". In the CCP framework we show that one can think of processes as formulas and process combinators (*e.g.* hiding and parallel composition) as logical connectives. This correlation is not merely vague analogy; rather, concurrent constraint programs turn out to be instances of an abstract, fibred categorical presentation of first-order logic *via* the structure called a hyperdoctrine [Lawvere 1969]. It is one of the fundamental results in categorical logic that hyperdoctrines correspond to, indeed *are*, logics with quantifiers. We shall develop that viewpoint in detail for CCP.

In earlier work [Panangaden et al. 1993], we discovered this hyperdoctrinal framework for *determinate* CCP modeled by closure operators. At the time, it seemed this view was an interesting coincidence arising from the fact that closure operators carry a logic-like structure. However, in the present paper we show the phenomenon to be far more pervasive. First, we extend our earlier hyperdoctrinal model to include the operational semantics of CCP languages. This involves a detailed study of simulation, with particular emphasis on how CCP programs interact with the environment. We then show how to apply these results to the indeterminate case. In the final sections we indicate that key parts of our modelling extend to other concurrent languages that are not remotely like concurrent constraint programming.

Our slogan "Concurrency is Logic" involves three identifications, to be contrasted with the realizability (or proofs-as-processes) viewpoint (for example see Abramsky [1991]):

Concurrency is Logic	Proofs-as-Processes
• Processes are Formulas	• Types are Formulas
• Combinators are Connectives	• Type Constructors are Connectives
• Simulations are Proofs	• Processes are Proofs

Note the fundamental distinction: processes as formulas versus processes as proofs. The basic dichotomy being reflected here is between logic programming and functional programming. Logic programming arises from proof construction and concurrent constraint programming comes from this tradition, whereas computation in functional programming is identified as proof normalization.

On the categorical side the inspiration for our work comes from the profound insights of Lawvere [1969, 1970]. He realized that first-order logic could be elegantly captured categorically, with quantifiers interpreted as certain functors adjoint to substitution. This is to be contrasted with the *ad hoc* algebraization of first-order logic achieved by Tarski and his followers [Henkin et al. 1971]. We present Lawvere's ideas in expository form in Section 3. Given this abstract categorical view one can pin down precisely the relationship to logic as was done by Seely [1983]: there is a natural bijection between Lawvere hyperdocrines and first-order logics, in a precise sense familiar to categorical logicians [Lambek and Scott 1986]. Thus, these categories *are* logics. This work extends to a host of important theories: Martin-Löf type theories, polymorphic lambda calculi, linear logic, *etc.*

The categorical viewpoint permits treating syntax and semantics uniformly: a semantical interpretation is simply a morphism from a (syntactically-presented) hyperdoctrine to a "model"-hyperdoctrine. This view is illustrated in the case of the closure operator hyperdoctrine for determinate CCP in Example 3.2 and extended to a natural class of fibred categories sufficient for the operational semantics of indeterminate CCP.

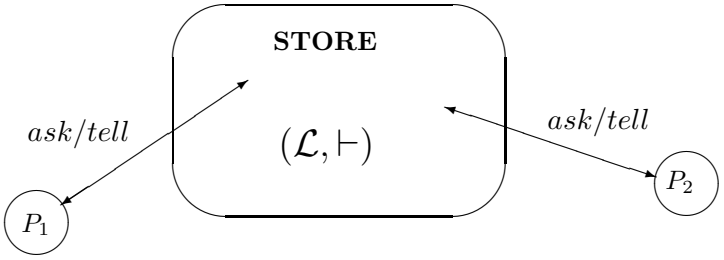
2. Concurrent Constraint Programming

In this section we give a brief summary of the denotational semantics of determinate concurrent constraint languages [Saraswat and Rinard 1990, Saraswat et al. 1991]. This semantics is given in terms of closure operators and leads to a very pleasant hyperdoctrine. This led to the subsequent research that forms the main thrust of the present paper. A detailed discussion of programming idioms within the paradigm of concurrent constraint programming, both determinate and indeterminate, is contained in the book by Saraswat [1993] based on his CMU dissertation.

The main point of concurrent constraint programming is to put, in Saraswat's words, "partial information in the hands of the programmer". Imagine a system consisting of concurrent processes interacting via shared data in some data base. The shared data can be thought of as a collection of assertions in some fragment of first order logic $(\mathcal{L}, \vdash)$.

Processes communicate by adding information to the common pool of data (a "tell" operation) or by asking whether an assertion is entailed by the existing pool of data (an "ask" operation). Thus a CCP language includes:

- A data/store language $(\mathcal{L}, \vdash)$ for describing the assertions one may make (*the constraint system*).
- An ask-tell process language for describing how processes interact with the data pool.



By analogy with imperative programming we call the collection of shared assertions (or constraints) the "store". In contrast with imperative programming, however, a store might only give partial information about the values of variables. For example rather than saying that a variable has the value 1994, it might merely say that the variable lies between 1729 and 4104. The ask-tell language will be a simple process calculus, based upon the primitive notions of ask and tell. The tell operation is the mechanism for communication: it takes a constraint ϕ and adds it to the common data pool. The ask construct is the mechanism for synchronization. Given a constraint ϕ, $ask(\phi)$ succeeds or fails depending upon whether the store entails ϕ. In the former case, the process continues, in the latter case the process suspends until (if ever) more data becomes available. The "ask" construct is what gives the programmer the ability to manipulate partial information since one can query the store despite the information being only partially defined.

2.1 Constraint Systems

It is convenient to go to a more abstract level and introduce constraint systems $\langle D, \vdash \rangle$ in the style of Dana Scott's information systems [Scott 1982], the only difference being we have no notion of consistency. From this view there is a set, D, of assertions that can be made. Each assertion is a syntactically denotable object in the programming language. The set D is equipped with a (compact) entailment relation $\vdash$, by which we mean that if an assertion p is entailed by any subset of D, it must be entailed by a finite subset of D. We write $\mathcal{P}_f(D)$ for the set of *finite* subsets of D and write $\sigma \subseteq_f \tau$ to mean that σ is a finite subset of τ. This leads us to:

DEFINITION 2.1. *A* **constraint system** *is a structure* $\langle D, \vdash \rangle$*, where* D *is a non-empty (countable) set of* **assertions** *or* **(primitive) constraints** *and* $\vdash \subseteq \mathcal{P}_f(D) \times D$ *is a relation, called the* **entailment relation** *satisfying:*
C1 $\sigma \vdash p$ *if* $p \in \sigma$
C2 $\sigma \vdash q$ *if* $\sigma \vdash p$ *for all* $p \in \tau$ *and* $\tau \vdash q$.

We extend $\vdash$ to a relation on $\mathcal{P}_f(D) \times \mathcal{P}_f(D)$ by defining:

$$\sigma \vdash \tau \quad \text{iff} \quad \sigma \vdash p \text{ for all } p \in \tau$$

In this notation, **C2** becomes:
C2′ $\sigma \vdash q$ if $\sigma \vdash \tau$ and $\tau \vdash q$.

A constraint system is propositional in the sense that its assertions or constraints are quantifier free. Indeed, a canonical example of a constraint system is any quantifier-free fragment of a first-order language, equipped with the usual notion of entailment. However processes will contain constraints, and thus involve local variables; in this sense propositions effectively get existentially quantified. Thus in our discussion we will talk about existential quantification even though the language of assertions does not have explicit quantifiers.

The store is equipped with a query-answering system that answers entailment queries. The exact mechanism of this subsystem is a parameter of our theory and is abstracted out of the discussion. Thus the issues of how to resolve constraints efficiently are orthogonal to our concerns.

DEFINITION 2.2. *The* **elements** *of a constraint system* $\langle D, \vdash \rangle$ *are those subsets* σ *of* D *such that* $p \in \sigma$ *whenever* $\tau \subseteq_f \sigma$ *and* $\tau \vdash p$*. The set of all such elements is denoted by* $|D|$.

A store is thought of as its entailment closure, so the denotations of stores are precisely elements of the constraint system. As is well known, $(|D|, \subseteq)$ is a complete algebraic lattice; the top element represents the inconsistent store.

2.2 Ask-Tell Languages

The following discussion in this subsection is a condensation of the discussion in Saraswat et al. [1991]. The syntax and operational semantics of the language are given in Table I. We assume given a constraint system $(D, \vdash)$. We use the letter σ to stand for an element of the constraint system. The basic combinators are the **ask** and **tell** written $ask(\sigma) \rightarrow P$ and $tell(\sigma)$ respectively. Intuitively, $ask(\sigma) \rightarrow P$ executes by asking the store whether σ holds, if it does then P executes, otherwise the process suspends; $tell(\sigma)$ adds σ to the constraints already in D. We shall assume that there is a process NIL with no transitions. As far as the effect on the store is

Syntax.

$$P ::= NIL \mid tell(\sigma) \mid ask(\sigma) \to P \mid P \parallel P \mid \nu x.P \mid \sum_{i=1}^{n} ask(\sigma_i) \to P_i$$

Operational Semantics.

$Tell$	$tell(\sigma) \xrightarrow{(\tau, \sigma \wedge \tau)} NIL$
Ask	$(ask(\sigma) \to P) \xrightarrow{(\tau,\tau)} P$ if $\tau \vdash \sigma$
$Parallel$	$\dfrac{P \xrightarrow{(\sigma,\tau)} P'}{P \parallel Q \xrightarrow{(\sigma,\tau)} P' \parallel Q} \quad \dfrac{P \xrightarrow{(\sigma,\tau)} P'}{Q \parallel P \xrightarrow{(\sigma,\tau)} Q \parallel P'}$
$Hiding$	$\nu x.P \xrightarrow{(\sigma,\sigma)} P$, where x is fresh.
$Guarded\,Choice$	$[\sum_{i=1}^{n} ask(\sigma_i) \to P_i] \xrightarrow{(\sigma,\sigma)} P_j$, where $\sigma \vdash \sigma_j$

Above, σ, τ range over assertions in some constraint system $(D, \vdash)$ while P and Q range over processes. The notation $P \xrightarrow{(\sigma,\tau)} Q$ means that P with store σ becomes Q with store τ.

TABLE I: Operational semantics for Ask-and-Tell cc languages

concerned, NIL could be simulated by a process like $tell(\mathsf{true})$ which does have a transition but makes no difference to the store.

In the description of the operational semantics we have transitions between processes. The transitions carry labels that describe the store before and after the transition. The transition rules should be self evident; the guarded choice makes the language indeterminate. The only point that needs explanation is the notion of a *fresh* variable. When we say that a variable is fresh we mean that it is completely new, *i.e.* it is not used as a local variable by any other process and certainly does not occur as a global variable. We assume that bound variables can be changed at will (α-conversion) to ensure this. Thus in particular two parallel components have no local variables in common. This is how the term, "fresh" is used in the semantics of block-structured languages and in ordinary logic or the lambda calculus.

A more careful presentation of the operational semantics of hiding follows. We assume that the store contains a list of "private" variables. These are the variables that appear as the result of hiding. Any information pertaining to these variables is available only to the process that created this private variable. In particular the environemnt cannot see any of these private variables. Existential quantification provides the precise notion of "hiding the information". Thus if $\vec{u}$ are the private variables in a store σ the globally visible part of the store is $\exists \vec{u}.\sigma$. In most of the discussion we will suppress this explicit mention of the notion of private variables and simply use existential

quantification to capture the visible part of the store.

In earlier presentations of the operational semantics we used the following presentation of this rule:

$$\frac{A \xrightarrow{(\exists_x \sigma,\, \tau)} B}{\nu x.A \xrightarrow{(\sigma,\, \sigma \wedge \exists_x \tau)} \nu x.(\tau, B)}$$

The intuitive explanation is as follows. Suppose that we have $\nu x.A$ in the store σ. Now A cannot see any references to the x in the store σ hence we have to consider the transitions of A in the store $\exists x.\sigma$ as is shown above the line. The resulting store τ may contain references to x but this x cannot be visible to the global environment and hence the global store seen is $\sigma \wedge \exists x.\tau$, the σ is back precisely because the x referred to in it *is* visible to the environment. Finally the information in τ needs to be still available to B so we have the notion of a "private store" for B represented by the pair (τ, B).

This explanation is of couse filled with the explicit discussion of variables and scoping that the notion of "fresh" variables avoids. In the present version we would say simply that the store is τ but that x is a "private variable" and hence the visible part of the store is $\exists x.\tau$. We need not existentially quantify the original store σ since x is a variable that has never appeared before. We also assume that alpha conversion of bound variables can be freely done as needed. We end the discussion with a tiny example. Consider the process $\nu x.tell(x = 1)$ in the initially empty store. It does the hiding step generating the fresh variable x, henceforth designated private (and used). Then it adds the formula $x = 1$ to the store. The resulting store is $x = 1$ with x designated private. The part of the store visible to other processes (and the environment) is thus $\exists x.x = 1$ which is logically equivalent to the trivial proposition true. Thus in terms of visible effect this process is the same as NIL.

The denotational semantics in Table II refers to the determinate fragment. The basic idea of this semantics is to model processes as certain special functions, closure operators, acting on stores. In order to model a process *compositionally*, it suffices to record its resting points. Mathematically, this is mirrored by the fact that a closure operator is completely specified by its set of fixed points. Given this representation of closure operators, we can define some operations on sets of fixed points that are clumsy to state in terms of closure operators *qua* functions. Most notably, one can define intersection of sets of fixed points of closure operators. It is quite awkward to write down this combinator in terms of functions; roughly speaking it describes "interleaving". It turns out that this operation is exactly what one needs to model parallel composition.

The motivation for using closure operators is as follows. A closure operator is extensive (increasing), which reflects the fact that the processes add information to the store. A closure operator is also idempotent, reflecting the intuition that classical entailment is not affected by having multiple