A Hyperdoctrinal View of Constraint Systems

Prakash Panangaden[*], McGill University
Vijay Saraswat, Xerox PARC
P. J. Scott[†], University of Ottawa
R. A. G. Seely[‡], McGill University and John Abbott College

Abstract

We study a relationship between logic and computation via concurrent constraint programming. In previous papers it has been shown how a simple language for specifying asynchronous concurrent processes can be interpreted in terms of constraints. In the present paper we show that the programming interpretation via closure operators is intimately related to the logic of the constraints. More precisely we show how the usual hyperdoctrinal description of first order logic can be functorially related to another hyperdoctrine built out of closure operators. The logical connectives map onto constructions on closure operators that turn out to model programming constructs, specifically conjunction becomes parallel composition and existential quantification becomes hiding of local variables.

1 Introduction

In this paper we develop a category theoretic view of the relationship between concurrent constraint programming and logic. One may think of this as an explication of the relationship between logic and what is often called logic programming. More significantly, however, this is a semantical account of constraint programming in which concurrency fits naturally. Indeed parallel composition of processes is one of the easiest combinators to define. As far as the programmer is concerned the most important point of the concurrent constraint program paradigm is that *the programmer can work directly with the notion of partial information.*

The basic thesis is this: a computational account of (first-order) logic should spell out the computational significance of entailment in a minimal logical setting. Fascinating issues

[*]Research supported by an operating grant from the Natural Sciences and Engineering Research Council and a team grant from Fonds pour la Formation de Chercheurs et l'Aide á la Recherche (Québec).

[†]Research supported by an operating grant from the Natural Sciences and Engineering Research Council, a team grant from Fonds pour la Formation de Chercheurs et l'Aide á la Recherche (Québec), and the Ontario Ministry of Colleges and Universities Ontario-Québec Exchange.

[‡]Research supported by Fonds pour la Formation de Chercheurs et l'Aide á la Recherche (Québec) and an operating grant from the Natural Sciences and Engineering Research Council

like "how does one efficiently decide whether an instance of entailment holds" or "what theorem-proving strategy is used" or "how do the other logical connectives fit in" come later. Thinking in terms of a weak logic with just conjunction and existential quantification led to the basic process calculus presented in [19] and studied extensively in [17]. In [20] a variety of denotational semantics, all based on closure operators, are introduced and studied.

In the present paper we show that the various constructions on closure operators, used in modeling the basic process combinators, arise as the functorial image of logical connectives. Thus from the computational point of view there is an intimate relation between the mathematical structures that arise in the study of closure operators and the logical conenctives. From a certain point of view, this correspondence is as significant as the model theoretic semantics of logic programming languages [14, 1] because it shows how the notion of partial information enters naturally into the computational setting. Furthermore there is a tight correspondence between intersection of sets of fixed points of closure operators, parallel composition of processes and conjunction. Thus, the framework that we present can be seen as the starting point of a general study of asynchronous processes. The idea of using closure operators to model logic variables in a parallel functional language was origially thought of by Pingali and discussed in detail in [10].

The computational paradigm can be described in the following way. 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 shift to partially specified values renders the usual notions of (imperative) "write" and "read" incoherent.

Instead, [17] 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 [10]. 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 [17], 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*.

2 Concurrent Constraint Languages

In this section we give a brief summary of the theory of concurrent constraint languages [19, 20]. A detailed discussion of programming idioms within this paradigm is contained in the forthcoming book by Saraswat [17].

The basic picture is as follows. Consider a system of concurrent processes interacting via shared data. The shared data can be thought of as a collection of assertions in some first order language. 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). In a concurrent constraint language one has a language for describing processes or agents and a language for describing the assertions that one may make. Such a language equipped with an entailment relation is called a *constraint system.*

We have a constraint system given as a logical language using a very weak positive logic. This is used to state assertions about the data that are used in the programming language. There are some minimal logical connectives provided, i.e. conjunction and existential quantification, while on the programming side there are some process combinators, e.g. parallel composition and hiding of local variables. The point of the logic being so weak is that the constraint system itself is not forced to use a powerful theorem prover of some sort. It is the minimal structure needed to get a notion of concurrency and synchronization via the imposition of constraints and requires only a simple notion of answering entailment queries.

By previous work [20], we know that the denotational semantics is fully abstract with respect to a traditional operational semantics. We first describe what is meant by a constraint system and give some basic lemmas that we use later. In the next subsection we give an operational semantics in the style of the Chemical Abstract Machine [3] or CHAM. Finally we sketch the results that show that the program combinators are the functorial image of the logical connectives.

An Informal View of Constraint Systems

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 $\vdash \subseteq \mathcal{P}_{fin}(D) \times D$ ($\mathcal{P}_{fin}(D)$ 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:

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 $\vdash \subseteq \mathcal{P}_{fin}(D) \times D$ is an* entailment relation *satisfying (where $\mathcal{P}_{fin}(D)$ 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 $\mathcal{P}_{fin}(D) \times \mathcal{P}_{fin}(D)$ 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. 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 [22]. 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.

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 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 reader will have noticed that the constraint system need not generate a *finitary* 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.

A concretely presented constraint system is basically a first order theory in the conjunctive calculus with only existential quantification. Unlike the case of simple constraint systems, it is not so obvious, a priori, to move from this to a structure that captures the notion of information in the way that one passes from an information system to a Scott domain or from a simple constraint system to a complete algebraic lattice. In particular we need to know how to carry over the structure described by the variables and the existential quantification. In previous work we used ideas from cylindric algebras [6] to define this algebraically. In the next section we give a presentation based on hyperdoctrines. We conclude this section with some semi formal examples.

Example 2.1 The Kahn constraint system.
The Kahn constraint system $\mathcal{D}(\mathcal{B}) = \langle D, \vdash_{\mathcal{D}} \rangle$ underlies data-flow languages [13], 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 $(X, Y \in)$`Var` *of variables. Let the 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,*
- $\mathbf{c}$ *is the predicate that is true of all sequences except Λ.*
- $\mathbf{f}$ *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,*
- $\mathbf{r}$ *is the function which maps Λ to Λ, and every other sequence s to the sequence obtained from s by dropping its first element,*
- $\mathbf{a}$ *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 $f(r^n(X)) = f(r^n(Y))$. 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 2.2 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 2.3 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 $[x_1, y_1] \subseteq [x_2, y_2]$. In the lattice generated by these assertions we will have lots of elements that we cannot think of in the usual set theoretic way. For example, since $\bigcap_{n>0}[0, 1+1/n] = [0, 1]$, we would normally 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 $X \in [0, 1]$. 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 $X \in [0, 1]$.

Syntax.

$$A ::= c \mid c \to A \mid A || A \mid \textsf{new } x \textsf{ in } A$$

Reaction Equations.

$$\begin{array}{l} c \rightharpoonup c, d \text{ if } c \geq d \\ c \to A, c \rightharpoonup A, c \\ (A||B) \rightharpoonup A, B \\ (\textsf{new } x \textsf{ in } A) \rightharpoonup < A >_x \\ < A >_x, c \rightharpoonup < A, \exists x.c >_x \\ < c, d >_x \rightleftharpoons < c >_x, < d >_x \\ < c >_x \rightharpoonup \exists x.c \end{array}$$

Above, c and d range over constraints while A and B range over processes.

Table 1: CHAM Operational semantics for the Ask-and-Tell cc languages

Semantics of Concurrent Constraint Languages

The discussion in this subsection is a condensation of the discussion in [20]. The syntax and operational semantics of the language are given in Table 1. We use the letter c to stand for an element of the constraint system. The basic combinators are the **ask** and **tell** written $c \to A$ and c respectively. Intuitively, $c \to A$ executes by asking the store whether c holds, if it does than A executes otherwise the process suspends; we also have an indeterminate generalization of the ask construct that simultaneously asks whether several constraints hold. The **tell** combinator c simply asserts the constraint c. The parallel composition of two agents is written $A_1||A_2$. Hiding is written new x in A. Finally we have procedure calls, including possibly recursive procedures but we exclude them from the present discussion.

The operational semantics is given as a set of reaction rules. We assume that the dynamics occurs in a "solution" in which one all the assertions in the store and all the assertions entailed by those in the store. Furthermore one has processes in the solution as well. We use the following chemical imagery to describe existential quantification. A process may be shielded by a "membrane" that allows facts to enter but may filter out information in the process of letting facts enter and leave. We write $< P >_x$ for a process P shielded by an x-membrane. A reaction that can happen can also happen inside a membrane.

The syntax and operational semantics of the language are given in Table 1.

The first equation says that the solution is entailment closed. One can see that the interaction between the processes constraints does not destroy the constraints, thus the idempotence of the process behaviour is built into the operational semantics. The second rule says that if the solution contains enough information to satisfy an ask the process makes the transition to the body. If one has the parallel composition of two process they just dissociate and work independently. The last four rules describe how block structuring in the programming language interacts with existential quantification. The explanation uses the notion of membrane discussed above. The key points are that when a shielded process comes into contact with a constraint, the constraint must first penetrate the membrane in order to react with the process. In so doing all information about the variable(s) being shielded will be hidden, as is shown by the existential quantification of the constraint. Similarly when a

Syntax.

$$A ::= c \mid c \to A \mid A \parallel A \mid \mathsf{new}\ X\ \mathsf{in}\ A$$

Semantic Equations.

$$\begin{aligned}
\mathcal{A}[\![c]\!] &= \{d \in |D| \mid d \geq c\} \\
\mathcal{A}[\![c \to A]\!] &= \{d \in |D| \mid d \geq c \Rightarrow d \in \mathcal{A}[\![A]\!]\} \\
\mathcal{A}[\![A \parallel B]\!] &= \{d \in |D| \mid d \in \mathcal{A}[\![A]\!] \wedge d \in \mathcal{A}[\![B]\!]\} \\
\mathcal{A}[\![\mathsf{new}\ X\ \mathsf{in}\ A]\!] &= \{d \in |D| \mid \exists c \in \mathcal{A}[\![A]\!].\exists_X d = \exists_X c\}
\end{aligned}$$

Above, c ranges over basic constraints, that is, finite sets of tokens.

Table 2: Denotational Semantics for the Ask-and-Tell cc languages

constraint leaves a membrane it has its shielded variable quantified out.

The basic idea of the denotational semantics is to model processes as closure operators. Operationally, the important point is that 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 as 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. It turns out that this operation is exactly what one needs to model parallel composition.

To be determinate, the process must define a function. This function maps each input c to a new store that corresponds to the result of the process augmenting the store. If the process, when initiated in c, engages in an infinite execution sequence we map c to the store that is the limit of the information added in this infinite process. Otherwise we map c to d if the process ultimately quiesces having upgraded the store to d. Intuitively the motivation for using closure operators is as follows. A closure operator is extensive (increasing), which reflects the fact that the processes add information. A closure operator is also idempotent. The fact that the processes are modeled by idempotent functions means that once they add the information that they are going to add the store is not going to be affected by adding the same information again. Finally we require monotonicity (and continuity) for the usual computability reason. The denotational semantics is given in 2. We have left out details like the definition of the environment mechanism and procedures. The closure operators are described by giving their set of fixed points.

3 Constraint Systems as Hyperdoctrines

In this section we review the connection, elucidated by Lawvere originally[15], between ordinary first-order logic and category theory through the use of hyperdoctrines. We will give a minimal and simplified account that only encompasses conjunction and existential quantification. These are the two connectives that any constraint system must have. Subsequent investigations will build on the present work to incorporate other logical connectives and their corresponding program combinators. Our main point in this section is to show that a constraint system is in fact a hyperdoctrine. The various axioms that we found necessary in our analysis of constraint programming languages [20] all follow from the adjunction between existential quantification and substitution. There is nothing original in this section; we follow the ideas in Seely's discussion [23] of the connection between natural deduction and hyperdoctrines.

Hyperdoctrines have, until recently, not received a great deal of attention; the main arena for categorical logic being elementary toposes. There has, however, been a surge of interest starting with the recent categorical description of models of the polymorphic lambda calculus [24]. There is also a recent trend to using more general fibred categories in describing dependent type systems, see for example the recent papers of Hyland and Pitts [7], Jacobs [8] and Pavlović [16].

Recall that constraint systems are given by a first order language interpreted over some structure and that they come equipped with a notion of entailment, conjunction and substitution. Our main task is to introduce existential quantification in terms of substitution. In order to facilitate the presentation, we do not use the most general definitions possible; for example, we assume that the constraint system is one sorted.

In the hyperdoctrinal presentation, one has a family of categories, called the *fibres*, indexed by the objects of a cartesian category called the *base* category. Corresponding to the arrows of the base category are functors between the fibres. Thus, in general, a hyperdoctrine over the base category is a contravariant functor[1] to **CAT**. For our purposes it will be sufficient to consider the fibres to be preordered sets.

We define a constriant system as a simplified hyperdoctrine.

Definition 3.1 *A constraint system is a contravariant functor $\mathcal{P}() : \mathbf{B}^{op} \longrightarrow \wedge - \mathbf{Preord}$, where $\wedge - \mathbf{Preord}$ is the category of meet-preorders, and $\mathbf{B}$ is cartesian (i.e. has all finite products). We assume that for each arrow f in $\mathbf{B}$, the (monotone) function $\mathcal{P}(f)$ (often written f^*) preserves meets and has a left adjoint, written $\exists_f$. We also require the following two conditions:*

1. **Beck condition** *If the following diagram is a pullback*

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "g"'] & B \arrow[d, "h"] \\ A' \arrow[r, "k"'] & B' \end{tikzcd}$$

and $\phi \in \mathcal{P}(B)$ then $\exists_g(f^*\phi) \sim k^*(\exists_h\phi)$, where $\sim$ means that we have two way entailment.

2. **Frobenius Reciprocity:** *For each $f : A \longrightarrow B$ in $\mathbf{B}$ and $\phi \in \mathcal{P}(A)$ and $\psi \in \mathcal{P}(B)$ we have $\exists_f(f^*\psi \wedge \phi) \longrightarrow \psi \wedge \exists_f\phi$.*

[1]Generally these are pseudofunctors but in the posetal situation that we consider it does not make any difference.

In the case where we have a constraint system syntactically presented as a concrete theory in first order conjunctive logic with existential quantification the hyperdoctrine conditions are easy to check [23]. In the next several paragraphs we describe the passage from a syntacticall presented first order theory to a (hyperdoctrinal) constraint system.

At the least, the base category is generated by a set of basic data values V : the objects of B are all finite products of V, including the empty product 1, and the arrows are the smallest set of arrows containing all the projections and identity arrows, and closed under composition and pairing. In general there may be other arrows between objects corresponding to terms. For example, if there is a function symbol f of arity two, this will appear as an arrow from V^2 to V. The role of the two conditions will be explained after we develop some elementary properties of the hyperdoctrine.

Convention 3.1 *Since the objects of* $\mathbf{B}$ *are indexed by nonnegative integers we will usually just write* $\mathbf{n}$ *when we mean the object* V^n *of* $\mathbf{B}$.

The base category essentially contains information about "terms" that describe individuals.

Information about the formulas, the "predicates", lives in the fibres indexed by the objects of $\mathbf{B}$. Rather than view the fibres as general categories we use the category of preorders equipped with binary meets, called $\wedge - \mathbf{Preord}$ here.

Convention 3.2 *We will use greek letters like* ϕ, ψ *etc. to stand for formulas.*

Definition 3.2 *Given a concrete constraint system we define a functor* $\mathcal{P}()$ *from* $\mathbf{B}^{op}$ *to* $\wedge - \mathbf{Preord}$ *as follows. The functor* $\mathcal{P}()$ *takes an object* $\mathbf{n}$ *to the formulas with* $\mathbf{n}$ *free variables constructed out of terms, variables, predicate symbols, conjunction and existential quantification. The preorder describes the entailment relation. Thus, if* $\phi \vdash \psi$ *we define* $\phi \leq \psi$, *or, equivalently,* $\phi \rightarrow \psi$. *If* f *is an arrow from* $\mathbf{m}$ *to* $\mathbf{n}$ *in* $\mathbf{B}$, *we define* $\mathcal{P}(f)$ *from* $\mathcal{P}(\mathbf{n})$ *to* $\mathcal{P}(\mathbf{m})$ *by* $\mathcal{P}(f)(\phi)[\vec{X}] = (\phi)[f(\vec{Y})]$, *where* $\vec{X}$ *and* $\vec{Y}$ *are vectors of variables.*

Note that the arrows $\mathcal{P}(f)$ are clearly meet preserving when f is one of the projection maps, because all this means in this case is that adjoining dummy variables is defined structurally. The entailment relation was originally defined between finite subsets of formulas and single formulas. One can easily redefine it in terms of pairs of formulas by introducing finitary conjunction.

Now we define existential quantification as the left adjoint to substitution and show how it corresponds to the usual definition in terms of variables. Consider, for definiteness, the fibres over $\mathbf{1}$ and $\mathbf{2}$, i.e. $\mathcal{P}(\mathbf{1})$ and $\mathcal{P}(\mathbf{2})$. Recall that $\mathbf{2}$ is just $\mathbf{1} \times \mathbf{1}$, let p be the first projection from $\mathbf{2}$ to $\mathbf{1}$. The functor (monotone function between preorders), $\mathcal{P}(p)$, written p^* by convention, from $\mathcal{P}(\mathbf{1})$ to $\mathcal{P}(\mathbf{2})$ is, according to the above, just $p^*(\phi)[X] = \phi[p(X, Y)]$. Because p^* preserves meets, it has a left adjoint[2], written $\exists_p$, a functor from $\mathcal{P}(\mathbf{2})$ to $\mathcal{P}(\mathbf{1})$. In fact,

$$\exists_p.\psi[X, Y] = \exists X' \exists Y' (p(X', Y') = X \wedge \psi[X', Y']).$$

By Tarski's trick, this equals $\exists Y' \psi[X, Y']$. It is easy to prove directly (for intuitionistic first order logic with equality) that $\exists_p$ is left adjoint to subsitution p^*.

[2]This is easily proved for Galois connections between posets, see for example [5].

Slightly more generally, in the multisorted case, we may allow our base category $\mathbf{B}$ to be the cartesian category generated by a set of Sorts $U, V, \ldots$, in which any sorted function symbols $f : U \longrightarrow V$ become arrow-forming operations: i.e. the arrows of $\mathbf{B}$ are the smallest set of arrows containing the projections, identities, and the sorted function symbols f, closed under composition and pairing. We may then define *subsititution and existential quantification along term* f , as follows (using lower case letters as variables): $f^*\Psi(v) = \Psi[f(u)/v]$, and $\exists_f\Phi(u) = \exists u(f(u) = v \wedge \Phi(u))$. Again, rules of first-order intuitionistic logic with equality show that $\exists_f$ is left adjoint to f^*. It is easily verified that this definition includes the former, when f is a projection p.

Defined this way, the functor enjoys all the properties one normally expects of existential quantification. The only change that one needs to make is to insert p^* in appropriate places in order to take into account the stratification induced by the arities of formulas. For example, in the traditional presentations of first-order predicate calculus, one has $\phi \leq \exists_X.\phi$ where X is a variable. In the present framework, in order to make sense of $\phi \leq \exists_p\phi$ we really need to write $\phi \leq p^*(\exists_p\phi)$, since we need to make the formulas live in the same fibre before we can sensibly compare them. The fact that $\phi \leq p^*(\exists_p\phi)$, is of course just the unit of the adjunction. We collect together the basic facts about existential quantification in the present framework. For convenience, we write down the hyperdoctrinal version as well as the version with existential quantification defined in terms of free variables and without the stratification in terms of the number of free variables. Proofs, are omitted here; in any case they are all trivial.

Fact 3.1 $\phi \leq p^*(\exists_p\phi)$ *[*$\phi \leq \exists_X.\phi$*].*

The proof is immediate from the definition of $\exists_p$ as the left adjoint of p^*; this is the unit of the adjunction.

Fact 3.2 $\exists_p(p^*\phi) \leq \phi$ *[*$\exists_X.\phi \leq \phi$ *if* X *does not occur in* ϕ*.]*

This is just the counit of the adjunction.

Fact 3.3 $\exists_p(p^*(\exists_p(\phi))) = \exists_p(\phi)$ *[* $\exists_X.\exists_X.\phi = \exists_X.\phi$ *], where the equality can mean either two way entailment or equality of subsets of* V^N*.*

This is just one of the two "triangle equalities" of adjunctions. (In the posetal context, this is an equality; generally one could only expect maps going each way, with equality of one composition.) This equation says that existential quantification is idempotent.

Fact 3.4 $p^*(\exists_p(p^*(\phi))) = p^*(\phi)$ *[*$\exists_X.\phi = \phi$ *if* X *does not occur in* ϕ*.]*

This is the other triangle equality.

Fact 3.5 *If* $V \neq \emptyset$ *(*i.e. *if there is an arrow* $0 \to 1$ *in* $\mathbf{B}$*), then* $\exists_p(p^*(\phi)) = \phi$ *[*$\exists_X.\phi = \phi$ *if* X *does not occur in* ϕ*.]*

The proof follows from applying f^* to the previous equation, for the map $f\colon 0 \longrightarrow 1$ mentioned in the hypothesis. Thus, this says we have a reflection rather than just an adjunction. For the following we just use colimit preservation.

Fact 3.6 *Suppose that the fibres are equipped with a least element, generically written $false_N$. Then $\exists_p.false_{N+1} = false_N$.*

This is immediate because left adjoints preserve colimits, or initial objects in this case. In view of this we will skip the subscripts on $false$ henceforth.

Fact 3.7 *If joins, written $\vee$, exist then*

$$\exists_p(\phi \vee \psi) = \exists_p(\phi) \vee \exists_p(\psi)$$

.

$$[\exists_X.(\phi \vee \psi) = \exists_X.\phi \vee \exists_X.\psi]$$

Again this is immediate because left adjoints preserve joins.

Fact 3.8 *If joins exist then,*

$$\exists_p(\phi \vee p^*(\exists_p(\psi))) = \exists_p(\phi) \vee \exists_p(\psi).$$

$$[\exists_X.(\phi \vee \exists_X.\psi) = \exists_X.\phi \vee \exists_X.\psi]$$

This follows from facts 3.5 and 3.3.

The Beck condition that we discussed above needs to be checked for only a few simple classes of diagrams. Again when the constraint system is derived from syntax these are easily checkable formulas. Frobenius reciprocity, when one has implication, is equivalent to saying that the maps of the form f^* preserve implication. The following calculation demonstrates part of this claim.

$$\begin{array}{rcl}
f^*(\psi) \wedge \phi & \vdash & f^*(\exists_f(f^*(\psi) \wedge \phi)) \\
\phi & \vdash & f^*(\psi) \Rightarrow f^*(\exists_f(f^*(\psi) \wedge \phi)) \\
\phi & \vdash & f^*(\psi \Rightarrow \exists_f(f^*(\psi) \wedge \phi)) \\
\exists_f \phi & \vdash & \psi \Rightarrow \exists_f(f^*(\psi) \wedge \phi) \\
\psi \wedge \exists_f \phi & \vdash & \exists_f(f^*(\psi) \wedge \phi)
\end{array}$$

where the first line is the unit of the adfunction, the second, is the adjunction between implication and conjunction, the third is preservation of implication and the last two are the same adjunctions used in the reverse direction. In the absence of an implication in the fibres we demand Frobenius reciprocity as a condition. When the hyperdoctrine arises from a concrete presentation of logic, we can easily prove Frobenius reciprocity. In fact Seely [23] shows how one can go back and forth between the hyperdoctrinal presentation and first order logic with equality.

We conclude this section with a discussion of equality. Suppose that we have equality in our syntax. Again, for simplicity, we consider the case where the relevant base objects are $\mathcal{P}(\mathbf{1})$ and $\mathcal{P}(\mathbf{2})$. In a cartesian category, we have, for any object A, an arrow, the diagonal arrow, Δ from A to $A \times A$ given by $\langle I_A, I_A \rangle$. Thus in the base category we have an arrow from $\mathcal{P}(\mathbf{1})$ to $\mathcal{P}(\mathbf{2})$. In the same was as we defined p^* we can define Δ^*, which now goes from $\mathcal{P}(\mathbf{2})$ to $\mathcal{P}(\mathbf{1})$. If our logic has equality then Δ^* has a left adjoint. The left adjoint to Δ^* is written $\exists_\Delta$. The definitions can be understood as follows. Let ϕ be a formula with one free variable, then $\exists_\Delta \phi$ is a formula with two free variables obeying $\exists_\Delta \phi[X, Y] \equiv (X = Y) \wedge \phi[X]$. In particular, if we choose ϕ to be $true_1$ we get $\exists_\Delta true_1$ is the same as $X = Y$. We can use a quick categorical argument to show that $\exists_X . X = Y$ is equal to $true_1$. We have that $\exists_X . X = Y$ is, in categorical form, $\exists_p \exists_\Delta true_1$. Since the existential quantifier acts functorially we can rewrite this as $\exists_{p \circ \Delta} true_1$ but $p \circ \Delta = Id$ so we have $\exists_{Id} true_1 = true_1$ where we have again used the functoriality of $\exists$ in the last step. Finally we remark that checking the Beck conditions is easy because one has to check them in the case that the hyperdoctrine expresses simple properties of first order logic or first order logic with equality as has already been done by Seely [23].

4 A Hyperdoctrine of Closure Operators

In this section we show how one can build a new hyperdoctrine, actually two new hyperdoctrines, by defining a suitable functor from $\wedge -$ **Preord** to **CAL**, the category of complete algebraic lattices and then an endofunctor on **CAL**. Roughly speaking the first functor takes us from logic to "information structure" in a familiar way and the second takes us from an information structure to a collection of closure operators. The second construction is not familiar and, though, *in retrospect*, it is clear and easy to describe it makes precise an intuition for which it is not, a priori, clear that one can have a formalized statement.

The category $\wedge -$ **Preord** has preordered sets equipped with binary meets as objects and monotone functions as the morphisms. The category we actually use is not quite **CAL** but instead $\mathbf{CAL}^{adj}$ which has complete algebraic lattices as objects and adjunction pairs as morphisms.

Definition 4.1 *The objects of the category* $\mathbf{CAL}^{adj}$ *are complete algebraic lattices. A morphism m from L to M is a* adjunction pair *between L and M. In more detail, a morphism m from L to M is a pair of monotone functions,* $\langle f : L \to M,\ g : M \to L \rangle$ *with f left adjoint to g (i.e. $f \dashv g$). The composition of morphisms* $\langle f,\ g \rangle$ *and* $\langle h,\ k \rangle$ *is* $\langle h \circ f,\ g \circ k \rangle$.

By the adjoint functor theorem, f preserves arbitrary sups.

The lattices in question are obtained by taking entailment closed sets of formulas or "theories". The programming significance of this is that these theories embody the information present in the store of a constraint programming system. The fact that we have a hyperdoctrine structure here means that one can pass from the logical concepts, conjunction and existential quantification to the information theoretic concepts of combining information and hiding information. The fact that the hyperdoctrine is the functorial image of the previous hyperdoctrine under a very obvious functor means that there is a very small shift in viewpoint taking place here. Since we are getting this hyperdoctrine by composing the previous hyperdoctrine with a functor the Beck conditions and Frobenius reciprocity hold automatically. This is a great simplification over having to check these conditions explicitly.

We can define a functor, $\mathcal{F}()$, from $\wedge -$ **Preord** to **CAL** as follows. In the next few paragraphs let A and B be meet preorders and $m : A \to B$ a monotone function between them.

Definition 4.2 *A* filter *in A is an upwards closed set that is also closed under the formation of binary (and hence all finitary) meets.*

Definition 4.3 *The lattice $\mathcal{F}(A)$ is defined to be the set of filters of A ordered by inclusion.*

It is easy to check that $\mathcal{F}(A)$ is a complete algebraic lattice. We could have equally well used the reverse inclusion order on filters. The present choice of order is the so called "information ordering", traditionally used in programming language semantics. It has the unfortunate effect of reversing the sense of a few adjunctions, but the intuitions associated with the notion of information are too useful to give up. The reason that we have meet closure in the definition of filters is because we want filters to be entailment closed with the notion of entailment introduced in our preliminary discussion of constraint systems. Our notion of entailment there was that a finite set of formulas could entail a formula. Thus if ϕ and ψ are in an entailment closed set and we have meets then $\phi \wedge \psi$ will have to be included as well.

Convention 4.1 *We use letters like u, v, w to stand for elements of the complete algebraic lattices generated in this way.*

Notation 4.1 *Given ϕ an element of A, we write $(\phi)\uparrow$ for the principal filter generated by ϕ. For any subset A' of A we write $(A')\uparrow$ for $\bigcup_{\phi \in A}(\phi)\uparrow$.*

Note that with our choice of ordering we have $\phi \leq \psi \Rightarrow (\psi)\uparrow \sqsubseteq (\phi)\uparrow$.

Notation 4.2 *Given any set X, a function f from X to X' and a subset Y of X we write $f(Y')$ for the direct image $\{f(y) | y \in Y\}$.*

Definition 4.4 *The arrow part of the functor $\mathcal{F}()$, is given by mapping the monotone function m to the pair, $\langle \mathbf{F}(m),\ m^{-1}\rangle$, where $\mathbf{F}(m)$ from $\mathcal{F}(A)$ to $\mathcal{F}(B)$ is given by $u \mapsto (m(u))\uparrow$ where $u \in \mathcal{F}(A)$ and m^{-1} is inverse image.*

It is easy to see that $\mathcal{F}()$ really is a functor between the categories defined above.

Proposition 4.5 *If A is a meet-preorder then $\mathcal{F}(A)$ is a complete algebraic lattice with joins given by upward closures of unions and meets given by intersections. If m is a monotone function from A to B then $\langle \mathbf{F}(m),\ m^{-1}\rangle$ is an adjunction pair from $\mathcal{F}(A)$ to $\mathcal{F}(B)$. $\mathcal{F}()$ defined in this way is a functor.*

Proof: The proof is straightforward. As an example we check that $\mathbf{F}(m) \dashv m^{-1}$. Suppose that $u \in \mathcal{F}(A)$ and $v \in \mathcal{F}(B)$. We want to show that $\mathbf{F}(m)(u) \subseteq v \iff u \subseteq m^{-1}(v)$. The forward direction is trivial. For the reverse direction, let $x \in \mathbf{F}(m)u$, then, for some $z \in u$, $m(z) \leq x$. Now by assumption, $m(z) \in v$ and, since v is upwards closed, $x \in v$. ∎
In fact we can think of $\mathcal{F}()$ as going from adjunction pairs to adjunction pairs. In view of this we have the following theorem.

Theorem 4.6 *The composition of the functors $\mathcal{F}()$ and $\mathcal{P}()$ produces another hyperdoctrinal structure on $\mathbf{B}$ with $(p^*)^{-1}$* **right** *adjoint to $\mathbf{F}(p^*)$.*

Note that if we had chosen the reverse ordering we would have had existential quantification as a left adjoint as before. The following calculation shows that $(p^*)^{-1}$ is in fact $\mathbf{F}(\exists_p)$. Suppose that $u \in \mathcal{F}(A)$, we show that $(p^*)^{-1}(u) = \mathbf{F}(\exists_p)(u)$. Suppose that $\phi \in (p^*)^{-1}(u)$, i.e. $p^*(\phi) \in u$ or, in other words, $\exists_p p^*(\phi) \in \exists_p(u)$. Now using the fact that $\exists_p p^*(\phi) \leq \phi$ we conclude that $\phi \in \mathbf{F}(\exists_p)(u)$. For the other direction, we note that if ϕ in $\mathbf{F}(\exists_p)(u)$ then, for some $\psi \in u$, $\exists_p \psi \leq \phi$. Using the adjunction, $\psi \leq p^*(\phi)$. Since u is a filter, and hence upwards closed, we have $p^*(\phi) \in u$ so $\phi \in (p^*)^{-1}(u)$. We will often write $\langle \mathbf{F}(p^*), \mathbf{F}(\exists_p)\rangle$ rather than $\langle \mathbf{F}(p^*), (p^*)^{-1}\rangle$, to emphasize the role of existential quantification.

Now we consider closure operators over the lattices produced in the last hyperdoctrine. We have already explained the significance of closure operators for constraint programming. The remarkable property of closure operators is that one can think of them either as functions or in terms of their sets of fixed points. We will pass back and forth between these two ways of viewing closure operators.

We recall the formal definition of closure operators and some basic facts about them [21, 5]. In order to avoid confusion in comparing our statements with those in the compendium we note that we say "left adjoint" where the compendium would say "lower adjoint" and similarly we say "right adjoint" where the compendium would say "upper adjoint".

Definition 4.7 *Given a lattice L a function c from L to L is a* closure operator *if c is monotone, idempotent, i.e. $c = c \circ c$, and increasing (extensive), i.e. $\forall x \in L. x \leq c(x)$.*

The following proposition, taken from pages 21 and 22 of [5], gives an equivalent characterization of closure operators.

Proposition 4.8 *Let f be a monotone function from L to L. Let f° be the corestriction to the image $f^\circ : L \to f(L)$ and let $f_\circ$ be the inclusion of the image of f in L, $f_\circ : f(L) \to L$. Then f is a closure operator iff f° is right adjoint to $f_\circ$.*

The next two propositions relate a closure operator with its set of fixed points.

Proposition 4.9 *The set of fixed points of a closure operator are closed under the formation of meets (infs) to the extent that they exist.*

Proposition 4.10 *Let L be a complete lattice. Let $\mathbf{Cl}(L)$ be the set of closure operators on L ordered extensionally. Let $\mathcal{C}(L)$ be the set of meet-closed subsets of L ordered by* reverse *inclusion. Then the map $fix : \mathbf{Cl}(L) \to \mathcal{C}(L)$ that takes a closure operator to its image (which is its set of fixed points) is an order isomorphism. The inverse of fix is the map $clo : \mathcal{C}(L) \to \mathbf{Cl}(L)$ given by $clo(S)(x) = min((x)\uparrow \cap S) = inf((x)\uparrow \cap S)$.*

Proposition 4.11 *The collection of closure operators on a complete algebraic lattice ordered extensionally, form a complete algebraic lattice.*

The proof is sketched in Data Types as Lattices [21].

Notation 4.3 *We write $\mathbf{Cl}(L)$ for the closure operators on a lattice L.*

Now we define a hyperdoctrine of lattices by taking the closure operators on the lattces that form the fibres in the previous hyperdoctrine. Once again we will define the new hyperdoctrine as the functorial image of the preceding hyperdoctrine. It turns out, however, that in order to do this we need to use adjoint pairs in a fashion reminiscent of the use of embedding-projection pairs in the construction of D_∞. It does not seem possible to map directly the hyperdoctrinal structure above onto a hyperdoctrine of closure operators.

Now we note the following technique for lifting adjunctions between posets to adjunctions between their endofunction spaces.

Proposition 4.12 *Suppose that $\langle f, g\rangle$ is a morphism*[3] *in $\mathbf{CAL}^{adj}$ from L to M. Then $\langle f', g'\rangle$ is a morphism from $[L \to L]$ to $[M \to M]$; where $f' = \lambda h : [L \to L].f \circ h \circ g$ $g' = \lambda k : [M \to M].g \circ k \circ f$*

Proof: We need to establish that if $k \in [M \to M]$ and $h \in [L \to L]$ that $f \circ h \circ g \leq k$ iff $h \leq g \circ k \circ f$. Suppose that $f \circ h \circ g \leq k$, composing on the left with g and on the right with f we get $g \circ f \circ h \circ g \circ f \leq g \circ k \circ f$. The unit of the adjunction, $f \dashv g$, gives $g \circ f \geq Id_M$. Thus $h \leq g \circ f \circ h \circ g \circ f$ and hence $h \leq g \circ k \circ f$. Similarly for the other direction. ∎

The basic idea is to try to define an endofunctor from $\mathbf{CAL}^{adj}$ to $\mathbf{CAL}^{adj}$ as follows. We map an object L (a complete algebraic lattice) to the set of all closure operators on L ordered pointwise. For the arrow part of the functor, we define it by saying that it maps a morphism $\langle f, g\rangle$ from L to M to $\langle f', g'\rangle$ as defined in the last proposition. Since the closure operators on L are a sublattice of $[L \to L]$ we should have another adjoint pair and hence a morphism of $\mathbf{CAL}^{adj}$. Unfortunately, however, given h, a closure operator on L, $f \circ h \circ g$ need not be a closure operator on L.

In order to get around the above difficulty we use a remarkable function, V, discussed by Scott [21]. The function V maps functions on $[D \to D]$ to closure operators on D. It turns out that V is itself a closure operator on $[[D \to D] \to [D \to D]]$ whose fixed points are exactly the closure operators. In our discussion we use the characterization of closure operators in terms of their sets of fixed points.

Lemma 4.13 *If f is any monotone function in $[D \to D]$, where D is any poset, then $\{x \in D | f(x) \leq x\}$ is closed under the formation of meets insofar as they exist.*

Proof: Let S be any subset of $\{x \in D | f(x) \leq x\}$. Let u be $\sqcap S$. Note that $f(u)$ is a lower bound for S and hence $f(u) \leq u$. Thus $u = \sqcap S$ is in $\{x \in D | f(x) \leq x\}$. ∎

Definition 4.14 *Suppose that f is a function in $[D \to D]$ where D is any poset. We define the function V in $[[D \to D] \to [D \to D]]$ by $V(f) = clo(\{x \in D | f(x) \leq x\})$.*

The lemma tells us that this definition makes sense. In terms of functions, V is defined as $\lambda f.\lambda x.\mathbf{Y}(\lambda y.x \sqcup f(y))$. Intuitively this can be thought of as follows. In order to make f into a closure operator, extensionally greater than f, one has to guarantee that the result

[3]We only use the fact that they form an adjunction.

is bigger than the input; thus one is led to define it as $\lambda x.x \sqcup f(x))$ but now in order to guarantee idempotence one has to iterate this.

Thus, using V, we may take any monotone endofunction and convert it into a closure operator by applying V to it.

Notation 4.4 *Suppose that f is a monotone function in $[D \to D]$ we write $\overline{f}$ for $V(f)$.*

Comment 4.1 *Note that $\overline{f}$ is the least closure operator extensionally greater than f.*

Lemma 4.15 *If $\langle f,\, g\rangle$ is a morphism in $\mathbf{CAL}^{adj}$ from L to M then $\langle f',\, g'\rangle$ is a morphism in $\mathbf{CAL}^{adj}$ from $\mathbf{Cl}(L)$ to $\mathbf{Cl}(M)$ where $f' = \lambda h : \mathbf{Cl}(L).\overline{f \circ h \circ g}$ and $g' = \lambda k : \mathbf{Cl}(M).\overline{g \circ k \circ f}$.*

Proof: This can be proved directly as follows. We need to show that for $h : \mathbf{Cl}(L)$ and $k : \mathbf{Cl}(M)$ we have $f'(h) \leq k$ iff $h \leq g'(k)$.
Suppose that $f'(h) \leq k$, i.e. $\overline{f \circ h \circ g} \leq k$. Since $\overline{(.)}$ is a closure operator itself, $f \circ h \circ g \leq k$. Now using the argument from the proposition above, we get $h \leq g \circ k \circ f$ and so $h \leq \overline{g \circ k \circ f}$. Now suppose that $h \leq g'(k)$. We use the fact that closure operators can be represented by their sets of fixed points. In this paragraph, we will not notationaly differentiate between the closure operator as a function and the closure operator as a set of fixed points, the resulting confusion is easily resolved by context. The payoff is that the proofs are easy to read (and invent!). In terms of sets of fixed points; $\overline{g \circ k \circ f} \subseteq h$. We will show that $k \subseteq \overline{f \circ h \circ g}$. Assume that $u \in k$, i.e. that $k(u) = u$. According to the counit of the adjunction $f \dashv g$, $f \circ g(u) \leq u$. Thus, since g is monotone $g \circ k \circ f \circ g(u) \leq g \circ k(u) = g(u)$. This means that $g(u) \in \overline{g \circ k \circ f} \subseteq h$. Thus $h(g(u)) = g(u)$, hence $h(g(u)) \leq g(u)$. Applying f to both sides and using the counit of $f \dashv g$, we get $f \circ h \circ g(u) \leq u$ i.e. $u \in \overline{f \circ h \circ g}$, in other words, $k \subseteq \overline{f \circ h \circ g}$. ∎

We can now define the promised endofunctor on $\mathbf{CAL}^{adj}$.

Definition 4.16 *The functor $\mathbf{Cl}(-)$ from $\mathbf{CAL}^{adj}$ to $\mathbf{CAL}^{adj}$ is defined as follows. It takes a complete algebraic lattice L to the complete algebraic lattice of closure operators on L ordered extensionally (reverse inclusion of the sets of fixed points). It takes a morphism $\langle f,\, g\rangle$ from L to M to the morphism $\langle f',\, g'\rangle$ where $f' = \lambda h : [L \to L].\overline{f \circ h \circ g}$ $g' = \lambda k : [M \to M].\overline{g \circ k \circ f}$.*

We need, of course, to check that we really have a functor, i.e. that identities and compositions are preserved. In order to do this, however, it is much easier to describe the arrow part of the functor in terms of the effects on sets of fixed points.

Lemma 4.17 *Suppose that $\langle f,\, g\rangle$ is a morhism from L to M with $h \in \mathbf{Cl}(L)$ and $k \in \mathbf{Cl}(M)$. Then $\overline{g \circ k \circ f} = g(fix(k))$ and $\overline{f \circ h \circ g} = g^{-1}(fix(h))$.*

Proof: We use the same ambiguity between a closure operator and its set of fixed points. Suppose that $u \in L$. Now $u \in \overline{g \circ k \circ f} \Leftrightarrow g \circ k \circ f(u) \leq u$ by definition. From the unit of $f \dashv g$ we have $u \leq g \circ f(u)$, now using monotonicity of g the fact that k is a closure operator and transitivity we have $u \leq g \circ k \circ f(u)$ for any $u \in L$. Thus $u \in \overline{g \circ k \circ f} \Leftrightarrow u = g \circ k \circ f(u)$. The latter is of course in $g(k)$. The reverse inclusion is equally easy.

Suppose that $v \in M$. Now $v \in \overline{f \circ h \circ g} \Leftrightarrow (f \circ h \circ g(v) \leq v) \Leftrightarrow h \circ g(v) \leq g(v) \Leftrightarrow h(g(v)) = g(v) \Leftrightarrow v \in g^{-1}(h)$. ∎

With this in hand the proof of the folowing is trivial.

Lemma 4.18 $\mathbf{Cl}(\langle f_1, g_1\rangle) \circ \mathbf{Cl}(\langle f_2, g_2\rangle) = \mathbf{Cl}(\langle f_1 \circ f_2, g_1 \circ g_2\rangle)$.

This now completes the description of the endofunctor on $\mathbf{CAL}^{adj}$. Since we have defined it as acting on adjoint pairs it is evident that the hyperdoctrinal structure on $\mathbf{CAL}^{adj}$ is carried over to the closure operators.

One can now inspect the table for the denotational semantics of the concurrent constraint language and see that the definintion of the construct new X in A uses the concrete version of the existential quantification operation in the last hyperdoctrine. Parallel composition is modelled by the sup operation in the lattices which is, of course comes ultimately from the meet operation in the first hyperdoctrine.

5 Properties of the Closure Operator Functor

The functor $\mathbf{Cl}(-)$ defined in the last section turns out to define an adjunction between the category **Frm** of frames and the category $\mathbf{Frm}-\mathbf{I}$ of frames equipped with an "inversion" operation. This adjunction helps clarify the role of ask and tell in the categorical framework. For general constraint systems, which may not be distributive, one has only a lax adjunction. The adjunction also suggests some, as yet poorly understood, connections to locale theory.

We first recapitulate some basic definitions [12] about frames. Frames are complete lattices obeying the following infinite distributivity law $x \sqcap (\bigsqcup_i x_i) = \bigsqcup_i (x \sqcap x_i)$. Morphisms preserve finite meets and arbitrary joins. In the category that we consider, morphisms preserve arbitrary joins but not finite meets. Furthermore, we do not have any distributivity laws. Thus we are in a situation not as general as that of the last section. The category $\mathbf{Frm}-\mathbf{I}$ has, as objects, frames equipped with an additional unary operation written x^{-1}. This obeys the following additional axioms.

$$\begin{array}{ll}
f \leq g \Rightarrow g^{-1} \leq f^{-1} & 1 \\
f \leq (f^{-1})^{-1} & 2 \\
(f \sqcup g)^{-1} = f^{-1} \sqcap g^{-1} & 3 \\
f^{-1} \sqcup g^{-1} \leq (f \sqcap g)^{-1} & 4 \\
f \sqcap f^{-1} = \bot & 5 \\
\bot^{-1} = \top, \top^{-1} = \bot & 6
\end{array}$$

Any Heyting algebra has such a negation and the objects themselves are in fact Heyting algebras qua objects. The reason that we do not refer to this category as the category of Heyting algebras is because the morphisms are not required to preserve implication, only inversion. There is an obvious forgetful functor, U from **Frm** to $\mathbf{Frm}-\mathbf{I}$. We claim that $\mathbf{Cl}()$ is left adjoint to U.

In order to establish this claim we need to define the inversion operation on lattices of closure operators.

Definition 5.1 *Let f be a closure operator on L. Consider the set $L \setminus f$ of nonfixed points of f. We define f^{-1} to be the smallest meet closed subset of L that contains $L \setminus f$.*

Checking the following proposition is routine.

Proposition 5.2 *The inversion operation on closure operators obeys all the axioms required of a* $\mathbf{Frm} - \mathbf{I}$ *object.*

Proposition 5.3 *If L is a frame then* $\mathbf{Cl}(L)$ *is an object of* $\mathbf{Frm} - \mathbf{I}$.

Proof We already know that $\mathbf{Cl}(L)$ is a complete lattice. The preceding proposition shows that the inversion structure exists. The infinite distributivity law is a trivial calculation; viz.,

$$\begin{aligned}
&(f \sqcap \textstyle\bigsqcup_i f_i)(x) &&= \\
&(f(x) \sqcap \textstyle\bigsqcup_i f_i(x)) &&= \\
&\textstyle\bigsqcup_i (f(x) \sqcap f_i(x)) &&= \\
&\textstyle\bigsqcup_i (f \sqcap f_i)(x).
\end{aligned}$$

Given this structure on $\mathbf{Cl}(L)$ we have the following "representation theorem" for closure operators.

Theorem 5.4 *Let c be a closure operator on L. The set of fixed points of c is given by $\bigsqcup_{x \in L}[(x)\uparrow^{-1} \sqcap (c(x))\uparrow]$.*

Proof: Recall that the lub of a family of closure operators is given by the intersection of their set of fixed points. Suppose $u \in c$ and let x be an arbitrary member of L. Suppose that $x \leq u$, it follows that $c(x) \leq u$ so $u \in (c(x))\uparrow$. If $x \not\leq u$ then $u \in (x)\uparrow^{-1}$. In either case $u \in [(x)\uparrow^{-1} \sqcap (c(x))\uparrow]$. If $u \notin c$ then $u \notin [(u)\uparrow^{-1} \sqcap (c(u))\uparrow]$ and hence not in the intersection. ∎

This shows that arbitrary closure operators can be expressed in terms of the uparrow embedding, which defines tell, and its inverse and the basic lattice operations. The combination $[(x)\uparrow^{-1} \sqcap (c(x))\uparrow]$ is the set of fixed points of the closure operator defined by the ask/tell combination $ask(x) \longrightarrow tell(c(x))$. Thus any closure operator is a sup of such ask/tell combinations. This fact is reminiscient of the fact that every sublocale can be expressed as a sup of meets of open and closed sublocales [12]. Note, however, that the nuclei used in the definition of sublocales are closure operators that preserve meets, whereas our closure operators are not required to preserve meets.

With this in mind we can state the basic adjunction.

Theorem 5.5 $\mathbf{Cl}() \dashv U$.

Proof: We establish this by showing that given any $\mathbf{Frm}$ object L we can canonically embed L in $\mathbf{Cl}(L)$ in such a way that given any $\mathbf{Frm} - \mathbf{I}$ object K and any $\mathbf{Frm}$ map f from L to K we can factor this map through the embedding. The embedding of L into $\mathbf{Cl}(L)$ is achieved by $(.)\uparrow$. This will turn out to be the unit of the adjunction. Though this is not necessary, it is illuminating to check that this is a nautral transformation from $\mathbf{Cl}(.)$ to the identity functor. In order to do this, we need to show that $\mathbf{Cl}(f)((x)\uparrow) = (f(x))\uparrow$. Recall, however, that an arrow $f : L \longrightarrow L'$ is really a pair of arrows $\langle f,\ g\rangle$ and that by lemma 4.17, $\mathbf{Cl}(f)$, viewed in terms of its action on sets of fixed points, is just the inverse image under g. Thus, $\mathbf{Cl}(f)((x)\uparrow) = \{u \in L' | x \leq g(u)\} = \{u | f(x) \leq u\}$ since f is left adjoint to g. But this last expression is just $(f(x))\uparrow$.

Now we check the universal mapping property, which establises the adjunction. Given $f : L \longrightarrow U(K)$ we define $f^+ : \mathbf{Cl}(L) \longrightarrow K$ by $f^+(c) = \bigsqcup_{u\in L}[f(x)^{-1} \sqcap f(c(x))]$. Now we verify that $f^+((x)\uparrow) = f(x)$ by the following calculation. By definition we have $f^+((x)\uparrow) = \bigsqcup_{u\in L}[f(u)^{-1} \sqcap f(u \sqcup x)]$. Using the fact that f preserves sups and distributivity we get $\bigsqcup_{u\in L}[(f(u)^{-1}\sqcap f(u))\sqcup(f(u)^{-1}\sqcap f(x))]$. Now using property 5 of inversion we get $\bigsqcup_{u\in L}[f(u)^{-1}\sqcap f(x)]$ and using big distributivity we get $f(x) \sqcap [\bigsqcup_{u\in L} f(u)^{-1}]$. Now since f preserves $\perp$ we have $f(\perp)^{-1} = \top$, hence $[\bigsqcup_{u\in L} f(u)^{-1}] = \top$. Thus we get $f(x)$.

Finally, we need to show that f^+ is the unique arrow that satisfies $f^+((x)\uparrow) = f(x)$. We show that $(.)\uparrow$ is epic. Suppose that $h \circ (.)\uparrow = k \circ (.)\uparrow$. Let x be an arbitrary element of L. Now we have $h \circ (x)\uparrow = h(\bigsqcup_{u\in L}[(u)\uparrow^{-1} \sqcap (x \sqcup u)\uparrow])$, using the representation theorem for closure operators and the fact that the closure operator $(x)\uparrow$ is just $\lambda u.u \sqcup x$. Now since h is preserves all the structure, i.e. finite meets, arbitrary joins and inversion, and that $(u \sqcup x)\uparrow = (x)\uparrow \sqcup (u)\uparrow$, we conclude that the last expression can be rewritten as $\bigsqcup_{u\in L}[(h((u)\uparrow))^{-1} \sqcap [h((x)\uparrow) \sqcup h((u)\uparrow)]]$. Using the assumed equality, $h \circ (.)\uparrow = k \circ (.)\uparrow$, we can replace all hs with ks to get $\bigsqcup_{u\in L}[(k((u)\uparrow))^{-1} \sqcap [k((x)\uparrow) \sqcup k((u)\uparrow)]]$ from which the required equality immediately follows.

∎

The counit of the adjunction is the map $\epsilon = \lambda c : \mathbf{Cl}(L). \bigsqcup_{x\in L} x^{-1} \sqcap c(x)$.

6 The Fibrational Version

It is often easier to study indexed structures in what is called the fibrational form. Instead of presenting an indexed structure as a functor F from a base category $\mathcal{B}$ to another category $\mathcal{C}$ one can present it as a "projection" functor P from $\mathcal{G}$ to $\mathcal{B}$, where $\mathcal{G}$ is a suitably constructed category. Roughly speaking, the category $\mathcal{G}$ "collects" all the fibres of the indexed structure into a single category.

It is well known that one can go from a hyperdoctrine to a fibration by a standard construction called the "Grothendieck" fibration. If the fibres are posets the Grothendieck construction still yeilds a non-posetal category. In this section we discuss the fibrational version of constraint systems. The main reason for doing this will be to develop a suitable notion of map between constraint systems. A good elementary reference for fibrations is the recent book by Barr and Wells [2]. Our discussion is taken from chapter 11 of that book, simplified where possible to take into account that our fibres are posets.

Suppose that we have a base category $\mathbf{B}$ and a functor $\mathcal{P}$ from $\mathbf{B}$ to a category of posets $\mathcal{C}$. For example, $\mathcal{C}$ could be $\wedge - \mathbf{Preord}$. We construct a category $\mathcal{G}(\mathcal{P})$ and a functor $\mathbf{P}(\mathcal{P})$ from $\mathcal{G}(\mathcal{P})$ to $\mathbf{B}$. The inverse images of the functor $\mathbf{P}(\mathcal{P})$ are the fibres in the original indexed presentation. The construction is itself functorial in the sense that given arrows between indexed categories, i.e. natural transformations between functors from $\mathbf{B}$ to $\mathcal{C}$, one can define an arrow, i.e. a functor, between the corresponding fibrations.

The objects of $\mathcal{G}(\mathcal{P})$ are pairs $\langle \phi,\ n \rangle$ where n is an object of $\mathbf{B}$, which we are thinking of as an integer, and ϕ is an object in the fibre over n. This in our first hyperdoctrine the Grothendieck construction yields a category where the objects are formulas tagged by an integer which represents the number of free variables. The arrows of $\mathcal{G}(\mathcal{P})$ are pairs $\langle u,\ f \rangle : \langle \phi,\ n \rangle \longrightarrow \langle \phi',\ n' \rangle$ where f is an arrow in $\mathbf{B}$ from n' to n and u is an arrow in n' from $\mathcal{P}(f)(\phi)$ to ϕ'. An arrow like u is just an instance of the order relation. The definition of composition of arrows is more or less inevitable given the definition of arrows.

Note that the category $\mathcal{G}(\mathcal{P})$ is not posetal in general even when the fibres of the original indexed category are. In the case where the hyperdoctrine that we start with is just the one corresponding to a first order logic, the arrows in the resulting category are entailment instances between formulas and substitution instances of other formulas. Of course given two formulas with different numbers of free variables there may be several different substitution instances of one that entail the other.

7 Conclusions

We view the results of this paper as the start of a larger investigation in the same spirit. The most important direction is to formulate a notion of higher order constraint programming. Higher order process calculi are starting to be studied in earnest especially those related to the lambda calculus [4, 9]. In recent work, Jagadeesan and Pingali [11], and independently, Saraswat [18] have developed a higher order concurrent constraint process calculus. In order to understand models for these calculi we are using our framework to develop a category of constraint systems in analogy with the category of information systems. One can define such a category by applying the Grothendieck fibration construction to the hyperdoctrines that we have. The technical questions that we are studying are how one defines exponents in this category and how such exponential objects would be related to the models of higher order concurrent constraint calculi.

The other important direction to pursue is the study of other logical combinators and process combinators. For example, what happens when disjunction, negation, implication or universal quantification are added to the logic? One can also ask the reverse type of question, what is the logical significance of the meet operation on constraint systems? We have studied some of these issues already and are preparing additional investigations.

Date: Wed, 17 Jun 92 09:55:49 -0400 From: Prakash ¡prakash@trichur.cs.mcgill.ca¿

References

[1] K.R Apt and M.H. van Emden. Contributions to the theory of logic programming. *JACM*, 29(3):841–862, 1982.

[2] M. Barr and C. Wells. *Category Theory for Computing Science.* prentice-Hall, 1990.

[3] G. Berry and G. Boudol. The chemical abstract machine. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 81–94. ACM, 1990.

[4] G. Boudol. Towards a lambda-calculus for concurrent and communicating systems. In J. Diaz, editor, *TAPSOFT 89, Lecture Notes in Computer Science 351*, pages 149–161. Springer-Verlag, 1989.

[5] G.Gierz, K.H.Hoffman, K.Keimel, J.D.Lawson, M.Mislove, and D.S.Scott, editors. *A compendium of continuous lattices.* Springer-Verlag Berlin Heidelberg New York, 1980.

[6] Leon Henkin, J. Donald Monk, and Alfred Tarski. *Cylindric Algebras (Part I).* North Holland Publishing Company, 1971.

[7] J. M. E. Hyland and A. M. Pitts. The theory of constructions: Categorical semantics and topos-theoretic models. In *Categories in Computer Science and Logic*, pages 137–199. AMS, 1987. AMS Contemporory Mathematics Series 92.

[8] B. Jacobs. Fibrations. Submitted to Mathematical Structures in Computer Science, 1991.

[9] R. Jagadeesan and P. Panangaden. A domain-theoretic model of a higher-order process calculus. In M. Paterson, editor, *Proceedings of the 17th International Colloquium on Automata Languages and Programming*, pages 181–194. Springer-Verlag, 1990. Lecture Notes in Computer Science 443.

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

[11] R. Jagadeesan and K. Pingali. A higher order functional language with logic variables. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, 1992.

[12] Peter Johnstone. *Stone Spaces*, volume 3 of *Cambridge Studies in Advanced Mathematics.* Cambridge University Press, 1982.

[13] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*, pages 993–998. North-Holland, 1977.

[14] R.A. Kowalski and M.H. van Emden. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[15] F. W. Lawvere. Functorial semantics of algebraic theories. *Proc. Nat. Acad. Sci. U.S.A.*, 50:869–872, 1963.

[16] D. Pavlović. *Predicates and Fibrations.* PhD thesis, University of Amsterdam, 1991.

[17] V. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, 1989. To appear Doctoral Dissertation Award and Logic Programming Series, MIT Press.

[18] V. Saraswat. The category of constraint systems is cartesian closed. In *Seventh Annual IEEE Symposium On Logic In Computer Science*, 1992.

[19] Vijay Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, 1990.

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

[21] D. Scott. Data types as lattices. *SIAM Journal of Computing*, 5(3):522–587, 1976.

[22] D. S. Scott. Domains for denotational semantics. In *Ninth International Colloquium On Automata Languages And Programming*. Springer-Verlag, 1982. Lecture Notes In Computer Science 140.

[23] R. A. G. Seely. Hyperdoctrines, natural deduction and the beck conditions. *Zeitschr. f. math. Logik und Grundlagen d. Math.*, 29:505–542, 1983.

[24] R. A. G. Seely. Categorical semantics for higher-order polymorphic lambda calculus. *J. Symb. Logic*, 52(4):969–989, 1987.