ATMS-based constraint programming

Vijay A. Saraswat

Johan de Kleer

Brian C. Williams

Xerox PARC,
3333 Coyote Hill Road
Palo Alto, Ca 94304

September 25, 1991

Abstract

We are developing a broad and comprehensive theory of constraint programming, subsuming the paradigms of functional and logic programming [Sar89, SR90, SRP91, JSS91, SRH91]. The theory is based on a view of computation organized around the notions of constraint systems as systems of *partial information*, of *store-as-constraint* and *process-as-information-transducer*, and emphasizes the use of constraints for communication and control between concurrently executing processes.

Such languages offer a very simple and abstract level for specifying computations for combinatorial exploration. In prior work [JSS91, SRH91] we have shown how to smoothly integrate **Prolog**-style backtracking and various local consistency techniques within this framework. In this paper we develop the integration of this framework with the alternate approach to combinatorial exploration predominant in AI, *truth-maintenance systems* (TMSs) [Doy79, McA90, deK86a]. In particular, we give a generic mechanism for enriching *any* constraint system B with the notion of assumptions to obtain another constraint system $\mathcal{A}(B)$. To implement $\mathcal{A}(B)$, ATMS-implementation techniques are integrated with the constraint-solver for B. All the usual cc combinators — such as ask, tell, conjunction (parallel composition) and hiding, all closed under recursion — are immediately available on top of $\mathcal{A}(B)$, providing a simple and powerful constraint language. In addition, several new combinators can now be defined which take advantage of the extra structure of assumptions built into the constraint system. As before, each of these new combinators can be given a very simple mathematical characterization as closure operators over the underlying constraint system. They enable a simple and elegant expression of various previously developed programming notions on top of ATMSs, such as (various kinds of) consumers [deK86b].

1 Introduction

We take it as being self-evident that a large class of computations in AI (and knowledge representation, and natural language understanding) centrally involves the representation and manipulation of constraints. Our task is to develop the conceptual, mathematical and practical foundations of a general theory of computation based on constraints.

The *concurrent constraint* (cc) programming approach [Sar89, SR90, SRP91] is based on treating a constraint system as just a system of partial information *a la* [Sco82]: essentially, a set of objects (with unanalyzed internal structure) equipped with an entailment relation. Arising from earlier work in constraint logic programming [JL87] and concurrent logic programming [Mah87, Sar87], the basic move in this paradigm is to replace the notion of store-as-valuation central to von Neumann computing with the notion that the store is a constraint, that is, a set of pieces of partial information about the values that variables can take. The usual notions of "read" and "write" no longer make sense; instead computation progresses via the execution of *ask* and *tell* actions. An ask operations 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 with the constraints already in the store. (Thus computation progresses via the monotonic accumulation of information.) Concurrency arises because any number of agents may simultaneously interact with the store. Synchronization is achieved via a blocking ask — an agent blocks on a constraint c if the store is not strong enough to entail c; 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.

The very wide applicability of the underlying notion of a constraint system means that the notion of constraint programming based on it is very general (see [Sar89]) — indeed it encompasses logic programming, concurrent logic programming, data-flow and (at least first-order) functional programming. Hand-in-hand with this generality goes mathematical simplicity — (determinate) programs in this framework were shown to correspond to *closure operators* over the underlying constraint system E, and non-determinate programs were shown to correspond to closure operators over the Smyth power-domain of E [SRP91, JSS91]. Closure operators — operators over a partial order that are monotone, idempotent and extensive — have a very beautiful algebra summarized in Appendix A; in particular they are closed under a very natural notion of parallel composition. [Sar89, SRH91] showed how this paradigm accounts for the pre-nascent notions of constraint programming evident in the work of [dKS80] (constraint propagation techniques) and in [Ste80] (constraint posting), and how these ideas can be extended to accomodate the notion of local consistency techniques for solving networks of constraints.

1.1 Control of search.

The predominant style for combinatorial computation, embodied within Prolog, is to cast the computation as a search in an And-Or tree. Various techniques for searching such trees efficiently (perhaps in parallel) have been devised. For example, the Andorra principle, an embodiment of the least commitment principle, governs the selection of nodes to expand next at an And-phase: "determinate" nodes — those that have at most one Or-branch below them which is not known to be inconsistent — are expanded first.

A major drawback of such approaches is that there is no possibility for communication between different Or-parallel branches. For several applications, such as diagnosis, it is necessary for the problem-solver to be able to move back and forth between different Or-parallel worlds with facility. (For example, each consistent Or-parallel world corresponds to a diagnosis). Also, the techniques for controlling search available when exploring And-Or trees top-down tend to be based on controlling the *forward phase* of the search: the phase that determines what nodes to expand next. (Indeed, the integration of these techniques with the cc framework has been a topic of recent research and is now reasonably well-understood [Hen89, SRH91].) In general, such techniques offer a significant computational advantage over chronological backtracking (by detecting inconsistency earlier). However, they are of less use in many AI applications in which not all decisions depend on each other. For such applications, the methods and techniques of *truth maintenance systems* have been developed over the last decade [Doy79, deK86a, McA90].

Truth Maintenance Systems offer an attractive and conceptually very different computation model from the And-Or trees underlying Prolog-like languages. Within the standard view of constraint programming, non-determinism is reflected by *splitting*: at a branch-point, conceptually, copies of the state of the world (the store, other conjunctive agents) are made, and these copies are kept strictly insulated from each other. In the TMS viewpoint, no splitting is done, even conceptually: instead an *assumption* is associated with each branch, and all the information on that branch is made conditional on that assumption. To give a simple example, Prolog-like languages treat a formula $\phi \wedge (\psi_1 \vee \psi_2)$ as the (equivalent) formula $(\phi \wedge \psi_1) \vee (\phi \wedge \psi_2)$; TMSs treat it as the (equivalent) formula $\phi \wedge (A \rightarrow \psi_1) \wedge (\neg A \rightarrow \psi_2)$ (for A some new Boolean variable). In addition, in *assumption-based* TMSs [deK86a, deK86b, deK86c], each formula is associated with a *label* which records the set of conditions (conjunctions of assumptions) under which the formula has been asserted.

The fundamental idea in this paper is to extend the structure of tokens underlying the constraint programming language by building in the notion of assumptions and a restricted form of implication. Assume given a constraint system with vocabulary D. In the new constraint system, tokens of the form $(\{A_0, \ldots, A_{n-1}\}, c)$ are allowed, where the A_i are propositional variables, and c is a token in D or (a conjunction of) other propositional variables. Intuitively, such a token expresses the idea that the piece of information c is known to hold — but only if the assumptions $A_0, \ldots, A_{n-1}$ are in force.

We should emphasize that our approach yields not just *a* programming language on top of an ATMS, but an *infinite* family of languages, depending upon the particular constraint system employed. Most ATMS-based languages in the past can be thought of as having a very simple built-in constraint system — we shall present such a constraint system in detail later in this paper.

2 Assumptive constraint systems

In this section we briefly review the notion of a simple constraint system, and discuss how to generate from an arbitrary constraint system B (the "base" constraint system), and assumptive constraint system $\mathcal{A}(B)$, given some underlying set of "nodes" (Definition 2.3). A simple characterization theorem for such constraint systems is also given.

In the following, for any set D, we shall let $P(D)$ ($p(D)$) stand for the set of all (finite) subsets of D. Also, we use the symbol $\subseteq_f$ for the "is a finite subset of" relationship between two sets.

Definition 2.1 *A* simple constraint system *A is a structure $\langle D_A, \vdash_A\rangle$ where $\vdash_A$ is an* entailment relation on *D_A, that is, a relation $\vdash_A \subseteq p(D_A) \times D_A$ satisfying:*

A1 *$u \vdash_A c$ for every $c \in u$.*

A2 *if $u \vdash_A c$ for every $c \in v$, and $v \vdash_A d$, then $u \vdash_A d$.*

Together these properties imply monotonicity: if $u \vdash_A d$ and $v \supseteq u$ then $v \vdash_A d$.

In the following, we shall overload the entailment symbol, letting $u \vdash u'$ (for u, u' finite sets of tokens) stand for the conjunction of assertions $u \vdash d'$ for all $d' \in u'$. Finally, we take $\approx$ to be the equivalence relation on $p(D)$ induced by the pre-order $\vdash$.

Example 2.1 *For any set of tokens D, the* generic constraint system over *D, $G(D)$ is given by $\langle D, \{(u,d) \mid d \in u\}\rangle$, It is easy to check that (A1) and (A2) are satisfied and that $u \vdash_{G(D)} v$ iff $u \supseteq v$.*

Let $\perp \in D$ be a special pre-designated token. Then the generic constraint system over *D* with falsehood *$\perp$, $G_\perp(D)$ is $\langle D, \{(u,d) \mid \perp \in u$ or $d \in u\}\rangle$. Again, it is easy to check that (A1) and (A2) are satisfied.*

As another example, note that $\langle\emptyset, \emptyset\rangle$ specifies a constraint system with no tokens — called the vacuous *constraint system.*

In the above example, a set of tokens in $G(D)$ is nothing more than the "sum of its parts", that is a "free" conjunction. $G(D)$ is essentially the "strongest" constraint system on top of a token-set D — any other constraint system on D

must have an entailment relation that is weaker (i.e., is larger as a set) than $\vdash_G (D)$. Alternatively, $\vdash_G (D)$ is the intersection of all possible entailment relations on D. $G_\perp(D)$ is just like $G(D)$ except that it has the notion of falsehood built into it: $\perp$ satisfies the property that for every token $d \in D$, $\{\perp\} \vdash_{G_\perp(D)} d$. Shortly we shall see that for a set N of "nodes" $G_\perp(N)$ can be thought of as the constraint system underlying TMSs.

We now show how to generate an assumptive constraint systems $\mathcal{A}(B)$ from a given base constraint system B. We first fix a (possibly infinite) set N of propositional variables (called *nodes*). For simplicity, we shall assume that N is chosen to be disjoint from D_B. We identify a (possibly infinite) subset $A \subset N$ of *assumptions*. We require that there be a special symbol, conventionally written $\perp$ to signify falsehood, in N but not in A. Now we shall obtain the assumptive constraint system $\mathcal{A}(B) = \langle D_{\mathcal{A}(B)}, \vdash_{\mathcal{A}(B)} \rangle$ by combining B with $G_\perp(N)$ in a particular way.

Defining $D_{\mathcal{A}(B)}$. Let $D = D_B \cup D_{G_\perp(N)}$. A $d \in D$ is called a *datum*; a datum is a node (if it comes from $D_{G_\perp(N)}$, i.e., N) xor a *base constraint* (if it comes from D_B).[1]

Now we can define $D_{\mathcal{A}(B)}$: it is just the set of pairs (e, d), where e is a finite subset of N, and $d \in D$. Intuitively, a token (e, d) asserts that d must hold whenever each of the nodes in e hold. In such a pair, we shall call e the *antecedent* and d the *consequent*. Conventionally, tokens whose consequents are nodes (rather than base constraints) are called *justifications*. Each justification $(\{n_1, \ldots, n_k\}, n)$ can be thought of as the propositional Horn clause $n_1 \wedge \ldots \wedge n_k \rightarrow n$. Also, a token of the form $(e, \perp)$ is called a *nogood* (since it represents the assertion that not all of a set of nodes can hold simultaneously).

Note that we do not allow basic constraints $d \in D_B$ to occur in antecedents of tokens, since the ATMS is just a propositional engine. This effect can be obtained by means of a user program as shown in Section 3.

In the following, whenever a datum d occurs in a context where an element of $D_{\mathcal{A}(B)}$ is expected, we should think of d as shorthand for $(\emptyset, d)$. Similarly, for $u \subseteq D$, whenever (e, u) appears in a context where a subset of D is expected, we should take it to be the set $\{(e, d) \mid d \in u\}$, and for $z \subseteq D_{\mathcal{A}(B)}$ whenever (e, z) appears in a context where a subset of $D_{\mathcal{A}(B)}$ is expected, we should take it to be the set $\{(e \cup e', d') \mid (e', d') \in z\}$. Finally, for any set $Q \subseteq D_{\mathcal{A}(B)}$, we let $J(Q)$ denote its subset of justifications. (Thus, the set of justifications is just $J(D_{\mathcal{A}(B)})$.)

Defining $\vdash_{\mathcal{A}(B)}$. We define $\vdash_{\mathcal{A}(B)}$ in three steps. First we define an entailment relation on justifications, then we define an entailment relation on the data-set D, and finally we use these two relations to define $\vdash_{\mathcal{A}(B)}$.

[1] Just a word of warning: those familiar with the TMS literature will find this terminology slightly non-standard.

The relation $\vdash_J$ on the set of justifications is defined by: $u \vdash_J r$ iff the conjunction of each token in u (viewed as a propositional formula) propositionally entails r (viewed as a propositional formula). Note that for a conjunction of nodes e and node n, $e \vdash_{G_\perp(N)} n$ iff $e \vdash_J n$.[2]

The relation $\vdash_D$ is obtained by extending $\vdash_B$ and $\vdash_{G_\perp(N)}$ to $D = D_B \cup D_{G_\perp(N)})$ in the obvious way: roughly, $u \vdash_D d$ if u entails d in the base constraint system B or u entails d in the "TMS constraint system" $G_\perp(N)$. The only interesting aspect of the definition is that inconsistency must be handled correctly.

Definition 2.2 *Define $\vdash_D \subseteq p(D) \times D$ by: $u \vdash_D d$ iff one of the following conditions holds:*

- *u is inconsistent; that is, $u \cap D_{G_\perp(N)} \vdash_{G_\perp(N)} \perp$ or $u \cap D_B \vdash d'$ for all $d' \in D_B$,*
- *u entails d considering only the tokens in u of the same type as d; that is, $d \in D_B$ and $u \cap D_B \vdash_B d$, or $d \in G_\perp(N)$ and $u \cap D_{G_\perp(N)} \vdash_{G_\perp(N)} d$).*

It is easy to check that this relation satisfies (A1) and (A2).

Given a conjunction of nodes e and a finite set of tokens u, the *e-context of u*, denoted u_e, is just the set

$$\{d \mid (e', d) \in u, J(u) \cup e \vdash_J e'\}$$

of all data that "hold" in e. We are now ready to give the main definition:

Definition 2.3 (Assumptive constraint systems) *For a constraint system B, and an underlying set of nodes N and assumptions A, the assumptive constraint system $\mathcal{A}(B)$ is the structure $\langle D_{\mathcal{A}(B)}, \vdash_{\mathcal{A}(B)} \rangle$ where $\vdash_{\mathcal{A}(B)}$ is given by:*

$$u \vdash_{\mathcal{A}(B)} (e, d) \textit{ iff } u_e \vdash_D d$$

As usual, the elements of the constraint system, denoted $|\mathcal{A}(B)|$ are the $\vdash_{\mathcal{A}(B)}$-closed subsets of $D_{\mathcal{A}(B)}$. For every set $u \subseteq D_{\mathcal{A}(B)}$ we denote by $\overline{u}$ the smallest $\vdash_{\mathcal{A}(B)}$-closed set containing u. We define a conjunction of nodes e to be *inconsistent* in u iff $u \vdash_{\mathcal{A}(B)} (e, \perp)$.

Example 2.2 *The simplest example of an assumptive constraint system is obtained by taking B to be the vacuous constraint system. In such a case, the tokens of $\mathcal{A}(B)$ are just of the form (e, n), and $\vdash_{\mathcal{A}(B)}$ is propositional entailment. Such an assumptive constraint system can be implemented directly with an ATMS.*

[2]Note that by our notaitonal conventions, $e \vdash_J n$ should be read as: $\{(\emptyset, x) \mid x \in e\} \vdash_J (\emptyset, a)$.

The central property of the implication inherent in tokens (e, z) is that it can be regarded as adjoint to conjunction in the following precise sense familiar from logic:

Theorem 2.1 *For any set $u, z \subseteq D_{\mathcal{A}(B)}$, and $e \subseteq_f A$:*

$$u \vdash_{\mathcal{A}(B)} (e, z) \text{ iff } u \cup e \vdash_{\mathcal{A}(B)} z$$

Several observations can readily be made. Since $u \cup e \vdash_{\mathcal{A}(B)} u$, it follows that $u \vdash_{\mathcal{A}(B)} (e, u)$. Also, we get $e \cup (e, z) \vdash_{\mathcal{A}(B)} z$, and $e \cup (e, z) \vdash_{\mathcal{A}(B)} e \cup z$, since $(e, z) \vdash_{\mathcal{A}(B)} (e, z)$. From the last statement, employing the adjunction again we get $(e, z) \approx_{\mathcal{A}(B)} (e, e \cup z)$.

Note also that $e \cup u_e = (e \cup u)_\emptyset$ — that is, given a store u, a datum is in the context labelled by e iff it is in the context labelled by $\emptyset$, provided that the nodes e are asserted as being true.

2.1 Canonical representations of conditional constraints

In ATMS terminology, an *environment* is a conjunction of assumptions, a *label* for a node n is the set of environments s such that (s, n) is derivable from the given justifications. An environment s for n is said to be *sound* if (s, n) is entailed by the store, and consistent if $(s, \perp)$ is not entailed by the store. A label $l = \{s_0, \ldots, s_{k-1}\}$ for n is said to be *sound* if each s_i is sound for n, *consistent* if each s_i is consistent for n, *complete* if for any sound environment s for n, there is an i such that $s \supseteq s_i$ and *minimal* if for all $i < k$, if $s_i \subseteq s$ and s is a sound environment for n, then $s \subseteq s_i$. The basic ATMS algorithm maintains sound, consistent, complete and minimal labels for each node. We note below some consequences of our definitions which justify the label propagation algorithm in the ATMS.

For any two conjunctions of nodes e, e', recall that $e \vdash_{G_\perp(N)} e'$ iff $e \supseteq e'$ or $\perp \in e$.

Proposition 2.2 *For $e, e' \in p(A)$ and $d \in (D + A)$, $\{(e, d)\} \vdash_{\mathcal{A}(B)} (e', d)$ whenever $e' \vdash_{G_\perp(N)} e$. Further, $e \supseteq e'$ implies $u_e \supseteq u_{e'}$.*

As a special case, this proposition says that if a datum is in (the context of) a given environment, then it is also in (the context of) every superset environment.

Proposition 2.3 *For any $e \in p(D_{G_\perp(N)})$ $\{(e, \perp)\} \vdash_{\mathcal{A}(B)} (e, d)$ for all $d \in D$.*

As a special case, this proposition says that every datum holds in an inconsistent environment (an environment in the label of $\perp$).

These two propositions are used internally in the ATMS implementation of the generic constraint system over an arbitrary token-set D. We can think of the canonical form maintained internally by the ATMS as being obtained by deleting from (the closure of) any finite set of tokens u:

- tokens (s, d) (for $d \neq \perp$) if $(s, \perp) \in u$ (elimination of nogood environments from labels).
- tokens (s', d) if $(s, d) \in u$, for $s \subset s'$ (eliminating non-minimal environments).

If these two rules are followed, the label for each token $d \in D$ will be consistent, sound, complete and minimal.

3 A simple programming language

The notion of computation on top of an assumptive constraint system is no different from that on top of any constraint system. The store consists of a set of tokens, and computation progresses via a number of (possibly recursively defined) agents running in parallel, checking for the presence of certain tokens in the store (ask), and adding some other tokens to the store (tell). The role of the "problem-solver" in the usual TMS-architecture is played here by the user program, and the role of the ATMS is played by the constraint-solver implementing the entailment relation on which the given constraint system is based. In the case of an assumptive constraint system, this constraint solver can conceptually be thought of as consisting of two parts — a part which deals with managing environments (the TMS) and the part which deals with performing the entailment operations in the underlying constraint system. (In usual TMS/problem-solver architectures, the latter is thought of as part of the problem-solver: thus the "usual" TMS architecture is obtained by taking as built-in the vacuous constraint system.)

In addition to ask, tell and parallel composition operations — which were first defined in the form below in [SRP91] — a number of other combinators can be defined which exploit the extra structure of tokens and provide extra control over execution. In the following we shall follow the strategy of defining the new combinators by giving their mathematical denotations and then describing informally how such combinators can be implemented. This allows a far simpler and more precise description than is possible if one were to rely purely on implementational notions. As before, the denotation of each agent will continue to be a closure operator over the set of elements of the underlying constraint system, in this case $|\mathcal{A}(B)|$. We shall represent each closure operator by the set of its fixed-points.

Tell actions Given any finite set of tokens c, the agent $c\star$[3] adds the tokens in c to the current store. Using lambda-notation, the tell combinator $c\star$ is $\lambda z.\overline{z \cup c}$. The set of fixed-points of such an operator are exactly those elements of $|D_a|$

[3] When there is no risk of confusion, we write a $c\star$ as c.

Syntax.

P::=	$D.A$	*Programs*
D::=	$\epsilon \mid p(X) :: A \mid D.D$	*Declarations*
A::=		*Agents*
	$c\star$	*Tell*
	$\mid c \rightarrow A$	*Ask*
	$\mid A \wedge A$	*Conjunction (parallel composition)*
	$\mid \texttt{in}\ e\ \texttt{do}\ A$	*Focussing*
	$\mid \texttt{all}\ A$	*??*
	$\mid \exists X.A$	*Hiding*
	$\mid p(X)$	*Procedure Call*

Semantic Equations.

$$\begin{aligned}
&\mathcal{A}(c)e = \{d \in |D_a| \mid d \geq c\}\\
&\mathcal{A}(c \ \rightarrow\ A)e = \{d \in |D_a| \mid d \supseteq c \Rightarrow d \in \mathcal{A}(A)e\}\\
&\mathcal{A}(A \wedge B)e = \mathcal{A}(A)e \cap \mathcal{A}(B)e\\
&\mathcal{A}(\exists X.A)e = \{c \in |D_a| \mid \exists_X c = \exists_X d \text{ for some } d \in \mathcal{A}(A)e\}\\
&\mathcal{A}(\texttt{in}\ e'\ \texttt{do}\ A)e = \{d \in |D_a| \mid \overline{d \cup e} \in \mathcal{A}(A)e\}\\
&\mathcal{A}(\texttt{all}\ A')e = \cap_{e \in p(A)} \mathcal{A}(\texttt{in}\ e\ \texttt{do}\ A')(e)\\
&\mathcal{A}(p(X))e = \mathbf{S}_{[X/\alpha]}e(p))\\
&\mathcal{D}(\epsilon)e = e\\
&\mathcal{D}(p(X) :: A.D) = \mathcal{D}(D)e[p \mapsto \mathbf{S}_{[\alpha/X]}\mathcal{A}(A)e)]\\
&\mathcal{P}(D.A) = \mathcal{A}(A)\mathit{fix}(\mathcal{D}(D))
\end{aligned}$$

Above, 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.)

Table 1: Denotational semantics for the ATMS-based `cc` languages

which contain all the tokens in c:

$$c\star = \{d \in |D_a| \mid c \subseteq d\}$$

Example 3.1 *Execution of the tell agent* $(\{A_1, A_2\}, X = 1)$ *causes the token* $(\{A_1, A_2\}, X = 1)$ *to be added to the store. In operational terms, this causes the label for* $X = 1$ *to be updated to reflect the fact that the conjunctions of assumptions* A_1 *and* A_2 *is sufficient to cause* $X = 1$ *to hold.*

More generally, telling the store the (justification) token $(\{n_1, \ldots, n_k\}, n)$ corresponds to the TMS notion of "installing" the justification $n_1 \wedge \ldots \wedge n_k \rightarrow n$. In particular, telling the token $(\{n_1, \ldots, n_k\}, \perp)$ is asserting that the set $\{n_1, \ldots, n_k\}$ is a nogood.

Ask For any finite set of tokens c and agent A, the "ask" agent $c \rightarrow A$ (read: "c then A") is defined as follows: on any input z, examine if z has enough information to entail (each token in) c. If so, then return the result of running A in z; else return z unchanged. Using lambda-notation, the function is: $c \rightarrow A = \lambda z.\ \mathit{if}\ c \subseteq z\ \mathit{then}\ A(z)\ \mathit{else}\ z$. It is easy to see that this defines a closure operator (if A is a closure operator). In terms of fixed-points:

$$c \rightarrow A = \{z \in |D_a| \mid c \subseteq z \Rightarrow z \in A\}$$

Example 3.2 *The agent* $((\{A_1\}, X \geq 3), (\{A_2\}, Y = 4)) \rightarrow (\{B\}, Z = 1)$ *suspends until there are enough constraints in the contexts* u_e *(for environments* e *that entail* $\{A_1\}$*) to entail* $X \geq 3$*, (this is what it means for the store* u *to entail the token* $(A_1, X \geq 3)$*), and similarly for* $(\{A_2\}, Y = 4)$*; and if and when these conditions are satisfied the agent fires and adds the token* $(\{B\}, Z = 1)$ *to the store. (Note that the antecedent of the token added does* not *have to be related to the antecedents of the tokens in the ask-part of the agent.*

Relationship with consumers. Given a set of tokens u, the ask action (e, e') succeeds iff the conjunction of the nodes in e implies each of the nodes in e', given u. (In ATMS-terminology, a node a is said to imply a node b if every environment in the label for a is a subset of an environment in the label for b.) This notion is the basic building block of the *inclusive implication consumers* of [deK86c], which can therefore be expressed as particular kinds of such ask-and-tell agents.

Similarly, the *contradiction consumers* of [FdK88] are also a special case of such ask-and-tell agents. Suppose we want an agent A to be executed whenever it is discovered that a particular environment e is inconsistent. This is expressed by running the agent

$$(e, \perp) \rightarrow A$$

which asks whether the token $(e, \perp)$ can be entailed in the current store, and if so, runs A.

More generally, ask and tell combinators provide capabilities that are not available in the usual consumer architecture. Consumers do not explicitly reference node labels, whereas both ask and tell operations work on tokens of the form (e, d), for e an environment and d a basic token or an assumption.

Parallel composition For any agents A_1 and A_2, their parallel composition, written (A_1, A_2), is the agent which on an input store z alternately applies A_1 and A_2 (in any order), until no progress can be made (that is, a value is reached which is a fixed-point of both). It is not hard, though a bit tedious, to show that as long as both A_1 and A_2 are executed "fairly" the resulting store *does not depend* on the order of application of the agents.[4] In terms of fixed-points, parallel composition has a startlingly simple denotation:

$$(A_1, A_2) = A_1 \cap A_2$$

That is, the set of fixed-points of (A_1, A_2) is exactly the intersection of the set of fixed-points of A_1 and A_2.

Parallel composition is the fundamental combinator for treating a collection of agents as a single agent; in TMS-terminology, for treating a collection of consumers as a single consumer.

With and All combinators We now introduce a combinator which exploit the structure of tokens in $D_{\mathcal{A}(B)}$. The `in` combinator is of the form:

$$\texttt{in}\ e\ \texttt{do}\ A$$

Intuitively, the idea is to isolate just the constraints in the store which are in force given the assumptions e, and to execute A in such a store. More operationally, to run `in` e `do` A in the store u, the system isolates just the constraints u' that hold in the environment e, (logically, e' is just the ($\vdash_{\mathcal{A}(B)}$-closure) of the set $e \cup u$), runs A in u', and then adds to u the constraint $(e, A(u'))$. More precisely, the definition is:[5]

$$\texttt{in}\ e\ \texttt{do}\ A = \lambda z.\overline{z \cup (e, A(\overline{e \cup z}))}$$

Theorem 3.1 *In terms of fixed-points:*

$$\texttt{in}\ e\ \texttt{do}\ A = \{z \in |D_a| \mid \overline{z \cup e} \in A\}$$

The calculation above is routine and easy.

[4]An execution sequence is fair if there is no agent which is infinitely often enabled in the sequence but never executed. The possibility of infinite execution sequences arises in the presence of recursion.

[5]Recall that for any $u \subseteq D_a$, $\overline{u}$ is the smallest set containing u closed under the entailment relation $\vdash_a$.

Example 3.3 *The agent:*

$$\texttt{in}\ \{A_1, A_2\}\ \texttt{do}\ (X = 1 \ \rightarrow\ Y > 3)$$

is identical to the simple ask-tell agent:

$$(\{A_1, A_2\}, X = 1)\ \rightarrow\ (\{A_1, A_2\}, Y > 3)$$

In fact, as the next theorem demonstrates, for every finite (recursion-free) program containing the `in` combinator, there is an equivalent program without the `in` combinator. But the whole point of the ($\texttt{in}\ e\ \texttt{do}\ A$) construction is that A can in general be a very complex (perhaps even recursively defined agent) and the translation may not be as straightforward as above. Besides this gives a conceptually simple way of separating the context from the agent which it is desired to run in that context.

Theorem 3.2 *This combinator satisfies the following laws:*

$$\begin{array}{l}
\texttt{in}\ e\ \texttt{do}\ c = (e, c)\\
\texttt{in}\ e\ \texttt{do}\ (\texttt{in}\ e'\ \texttt{do}\ A) = \texttt{in}\ (e \cup e')\ \texttt{do}\ A\\
\texttt{in}\ e\ \texttt{do}\ (c\ \rightarrow\ A) = (e, c)\ \rightarrow\ (\texttt{in}\ e\ \texttt{do}\ A)\\
\texttt{in}\ e\ \texttt{do}\ (A_1, A_2) = (\texttt{in}\ e\ \texttt{do}\ A_1, \texttt{in}\ e\ \texttt{do}\ A_2)\\
\texttt{in}\ \emptyset\ \texttt{do}\ A = A
\end{array}$$

The proofs of all these assertions are straightforward from their definitions.

Relationship with consumers. The "`in` " combinator corresponds to the "implied-by" strategy for executing consumers [FdK88], as illustrated by the following example.

Consider the agent $\texttt{in}\ \{A, B\}\ \texttt{do}\ (N\ \rightarrow\ M)$. Suppose the current store has the label $\{\{A\}, \{C\}\}$ for N and the label $\{\}$ for M. To execute the agent, the environments of N in the "focus" $\{A, B\}$ are selected; others are ignored. (In this case, only the environment $\{A\}$ is selected.) For each such environment the corresponding tell action is performed. Thus, the net result of executing the above agent is to update the label for M to be $\{\{A\}\}$. Note that the same result would be obtained by expanding the agent, using the above theorem, to: $(\{A, B\}, N)\ \rightarrow\ (\{A, B\}, M)$.

`all` combinator. Using `in` , we can define the "`all` " combinator: the agent

$$\texttt{all}\ A$$

is equivalent to the (infinite) conjunction of the agents ($\texttt{in}\ e\ \texttt{do}\ A$) for all sets of assumptions e. It should be clear that the same result is obtained if the conjunction above is limited to environments that are consistent in the current store. (There is no point in firing an agent in an inconsistent environment since no more information can be added in that environment.)

Example 3.4 *The agent*

$$\texttt{all}\,(N \rightarrow M)$$

is the agent which checks for every *context whether N holds in that context, and, if it does, adds M to that context.*

Relationship with consumers. `all` A behaves very much like a basic consumer in the ATMS-architecture. In fact, the agent `all` $(N \rightarrow M)$ can be executed by installing a justification with antecedent N and consequent M: the usual ATMS-propagation algorithm will ensure that in each minimal, consistent environment in which N holds, M will also hold, thus providing a simple, finitary implementation.

Conditional constraint consumers can now readily be translated, as illustrated by the following example:

Example 3.5 *Consider the following model of an AND-gate:*

```
and(X, A, B) ::
    (A = 0 → X = 0), (B = 0 → X = 0),
    (A = 1 → X = B), (B = 1 → X = A),
    (X = 1 → (A = 1, B = 1)).
and_gate(M, X, A, B) ::
    M = ok → and(X, A, B),
    M = stuck_at_zero → X = 0,
    M = stuck_at_one → X = 1.
```

A typical invocation of such a device would be:

$$\texttt{all and\_gate(M,X,A,B)}$$

Operationally, such an agent checks for every environment e whether one of the constraints `M = ok`, `M = stuck_at_zero`, `M = stuck_at_one` *holds in e. If so, the corresponding action is taken — in the first case, the agent* `and(X,A,B)` *is executed (under the assumption e), and in the latter two cases, the corresponding constraint is added to the store, contingent on e.*

As discussed earlier, antecedents to justifications can only consist of nodes, not basic constraints. We can obtain the effect of basic constraints in antecedents, however, using ask operations. Suppose it is desired to use the basic constraint d in an antecedent. The effect can be achieved by introducing a node n_d corresponding to d, running the agent `all` $(d \rightarrow n_d)$ (recall that this is shorthand for `all` $((\emptyset, d) \rightarrow (\emptyset, n_d))$ and using n_d wherever it is desired to use d in an antecedent. As a result, in any context in which d can be shown to hold, the node n_d will be labelled “in”, as desired.

4 Conclusion

Above, we have discussed the notion of a general determinate `cc` language on top of an ATMS, given a precise description of the combinators, and shown how various "consumer-based" programming idioms in ATMS-based languages can be expressed in this set-up.

The addition of disjunctive agents to the above set-up does not present any new problems — it is straightforward to consider the denotation of every agent to be a closure operator over the Smyth-powerdomain and extend the two new combinators (**`all`** and **`in`**) to that setting. Indeed, disjunctive agents are needed in order to capture the "choose" construct of ATMS-based languages, and hence account for "interpretation-construction" in this framework.

Similarly, the addition of existential quantification is straightforward. Semantically, this requires us to move from simple constraint systems to *first-order* constraint systems (which are sets of tokens equipped with an entailment relation and also a notion of substitution and existential quantifiers). Operationally, existential quantifiers provide a simple and elegant way to create new nodes.

Several tasks remain to be done. Conceptually, there does not seem to be any difficulty in generalizing to a *first-order ATMS*, that is, to a setting where assumptions are first-order constraints rather than just propositional variables. For example, we may want to consider each of $\texttt{Mode} = \texttt{ok}$, $\texttt{Mode} = \texttt{stuck\_at\_zero}$, $\texttt{Mode} = \texttt{stuck\_at\_one}$ as different assumptions, and allow them to be appear in labels. Implementationally, we will have to allow the possibility that an environment may be inconsistent not because it subsumes a no-good but because it is *internally* inconsistent. For example, the environment $\{\texttt{Mode} = \texttt{stuck\_at\_zero}, \texttt{Mode} = \texttt{ok}\}$ is internally inconsistent (because it is an unsatifiable constraint). As another example, we could allow assumptions to be positive or negative propositional literals, thus requiring the implementation to have a full-scale Clause Management System (CMS) [RdK87]. Further practical exploration of these possibilities is warranted.

Practical implementations of such languages — over a non-trivial constraint system — need to be developed. Perhaps the simplest non-trivial example would be to consider a first-order constraint language with constants and equality; the next step would be to allow constructors, thus getting the constraint system **Herbrand**. We hope to report on such implementations in future work.

Acknowledgements. Conversations with Philippe Codognet have been very helpful.

References

[deK86a] J. deKleer. An assumption based TMS. *Artifical Intelligence*, 28:127–162, 1986.

[deK86b] J. deKleer. Extending the ATMS. *Artifical Intelligence*, 28:163–196, 1986.

[deK86c] J. deKleer. Problem solving with the ATMS. *Artifical Intelligence*, 28:197–224, 1986.

[dKS80] Johan de Kleer and Gerald Jay Sussman. Propagation of constraints applied to circuit synthesis. *Circuit Theory and Applications*, 8:127 – 144, 1980.

[Doy79] Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[FdK88] Kenneth D. Forbus and Johan de Kleer. Focusing the ATMS. In *AAAI 88*, 1988.

[GKK+80] 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.

[Hen89] Pascal Van Hentenryck. *Constraint satisfaction in logic programming*. Logic Programming Series. MIT Press, 1989.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 111–119. ACM, January 1987.

[JSS91] R. Jagadeesan, V. Shanbhogue, and V. Saraswat. Angelic non-determinism in concurrent constraint programming. Technical report, System Sciences Laboratory, Xerox PARC, January 1991.

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

[McA90] David McAllester. Truth maintenance. In *Proceedings of AAAI*, pages 1109–1116, July 1990.

[RdK87] Raymond Reiter and Johan de Kleer. Foundations of assumption-based truth maintenance systems: a preliminary report. In *Proceedings of AAAI-87*, pages 183 – 188, August 1987.

[Sar87] Vijay A. Saraswat. The concurrent logic programming language **cp**: definition and operational semantics. In *Proceedings of the SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 49–62. ACM, January 1987.

[Sar89] Vijay A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, January 1989. To appear, Doctoral Dissertation Award and Logic Programming Series, MIT Press, 1990.

[Sco76] Dana S. Scott. Data types as lattices. *SIAM*, 5(3):522–587, 1976.

[Sco82] Dana S. Scott. Domains for denotational semantics. In *Proceedings of ICALP*, 1982.

[SR90] Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of Seventeenth ACM Symposium on Principles of Programming Languages, San Fransisco*, January 1990.

[SRH91] Vijay A. Saraswat, Fransesca Rossi, and Pascal Van Hentenryck. Towards a general framework for constraint programming. Technical report, System Sciences Laboratory, Xerox PARC, February 1991.

[SRP91] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of Eighteenth ACM Symposium on Principles of Programming Languages, Orlando*, January 1991.

[Ste80] Mark Jeffrey Stefik. *Planning with constraints.* PhD thesis, Stanford University, January 1980.

[Tar56] A. Tarski. *Logics, semantics and meta-mathematics.* Oxford University Press, 1956. Translated by J.H. Woodger.

A Closure operators

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]; continuous closure operators have been used in [Sco76] to characterize data-types.

We list here some basic properties. Every closure operator can be uniquely represented by its range (which is the same as its set of fixed points); the function can be recovered by mapping each input to the least element in the range above it. (In the following, we shall often confuse a closure operator with its range, writing $x \in f$ to mean $f(x) = x$.) In fact, a subset of $|D|$ is the range of a closure operator iff it is closed under glbs of arbitrary subsets. (Thus, the range of every closure operator is non-empty, since it must contain $\top_{|D|} = \sqcup\emptyset$.) This condition — which we shall call the *determinacy condition* in what follows — corresponds to the requirement that for any input there should be a *unique* least fixed-point of the given function above that input.

Also, it can be shown that a closure operator is continuous iff its range is closed under the lubs of directed subsets. The extensional partial order on closure operators ($f \leq g$ iff for all x, $f(x) \leq g(x)$) translates into reverse-inclusion on the set of fixed points: $f \leq g$ iff $f \supseteq g$. The identity operator I (with range $|D|$) is the bottom element; the operator which maps every element to $\top$ (and hence has range $\{\top\}$) is the top element. In fact it is easy to see that the set of closure operators over $|D|$ ordered by $\supseteq$ is a complete (algebraic) lattice, with meets given by $P \sqcap Q = \{c \sqcap d \mid c, d \in P \cup Q\}$ and joins by $P \sqcup Q = P \cap Q$. As we shall see, joins correspond to parallel composition, and one of the contributions of this paper is to point out that meets too have a very interesting computational interpretation.

In the presence of angelic non-determinism (disjunction), it becomes necessary to require the domain of denotations to be closed under a disjunction operation. When initiated in a store, such an operation can return multiple solutions — hence it becomes necessary to take the denotation to be some kind of mapping from $|D|$ to sets of $|D|$. As discussed in [JSS91], the technically simplest solution is to take the domain of denotations to be the set of linear closure operators on the disjunctive power-domain $\mathcal{P}(|D|)$ (also called the *Smyth power-domain*) of the underlying constraint system. Roughly, this power-domain has as elements upwards-closed subsets of $|D|$, ordered by reverse set-inclusion.[6] A *linear* (also called *distributive*) closure operator over such a domain is a closure operator satisfying the property $f(S) = \cup_{s \in S} f(\{s\})$[7]. Intuitively, $f(S)$ is the set of outputs of a process obtained by taking the union of the outputs produced by the process when started on any store $s \in S$.

For the present discussion, the crucial property of such closure operators that they too can be characterized by their "resting points" — those inputs $\uparrow c$ (for $c \in |D|$) which are fixed-points. Given such a set f_S, the function f can be recovered as the unique linear extension of the function $\lambda c. \uparrow f_S \cap \uparrow c$. In fact, [JSS91] shows that the set of such closure operators is in one-to-one correspondence with the set of all Scott-compact subsets of $|D|$. The important point here is that no condition analogous to the determinacy condition (for closure operators over $|D|$) is required — -for instead of requiring that above any input there should be a *least* fixed-point, we can now take the result to be the (upwards-closure of) the set of *all* fixed-points above the input.

The set of linear closure operators over $\mathcal{P}(|D|)$ forms a distributive algebraic lattice with lubs given by intersection (of singleton fixed-point sets), and glbs by union.

[6] More precisely, the elements of the power-domain are required to be Scott-compact, that is, compact in the Scott-topology.

[7] Thus, linear functions do not combine the information present in two or more elements in the input set, but treat them separately.