Confluence in Concurrent Constraint Programming

Moreno Falaschi [a,1,2], Maurizio Gabbrielli [b,1], Kim Marriott [c] and Catuscia Palamidessi [d,1,3]

[a] *Dipartimento di Matematica e Informatica, Università di Udine, Via Delle Scienze 206, Udine, Italy. E.mail: falaschi@dimi.uniud.it, WWW: http://www.dimi.uniud.it/~falaschi*

[b] *Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy. E.mail: gabbri@di.unipi.it, WWW: http://www.di.unipi.it/~gabbri/gabbri.html*

[c] *Department of Computer Science, Monash University, Wellington Rd, Clayton 3168, Victoria, Australia. E.mail: marriott@cs.monash.edu.au, WWW: http://www.cs.monash.edu.au/people/profiles/marriott.html*

[d] *DISI, Università di Genova, Via Dodecaneso 35, 16146 Genova, Italy. E.mail: catuscia@disi.unige.it, WWW: http://www.disi.unige.it/person/PalamidessiC*

Abstract

Concurrent constraint programming (ccp), like most of the concurrent paradigms, has a mechanism of global choice which makes computations dependent on the scheduling of processes. This is one of the main reasons why the formal semantics of ccp is more complicated than the one of its deterministic and local-choice sublanguages. In this paper we study various subsets of ccp obtained by adding some restriction on the notion of choice, or by requiring confluency, i.e. independency from the scheduling strategy. We show that it is possible to define simple denotational semantics for these subsets, for various notions of observables. Finally, as an application of our results we develop a framework for the compositional analysis of full ccp. The basic idea is to approximate an arbitrary ccp program by a program in the restricted language, and then analyze the latter, by applying the standard techniques of abstract interpretation to its denotational semantics.

[1] Partially supported by the ESPRIT BRA project PARFORCE
[2] Partially supported by the HCM project CONSOLE
[3] Partially supported by the HCM project EXPRESS.

1 Introduction

Concurrent constraint programming (ccp) [14–16] is a programming paradigm which elegantly combines logical concepts and concurrency mechanisms. The computational model of ccp is based on the notion of *constraint system*, which consists of a set of constraints and an *entailment* relation. Processes interact through a common *store*. Communication is achieved by *telling* (adding) a given constraint to the store, and by *asking* (checking whether the store entails) a given constraint. Non-determinacy arises in two ways: because of a guarded choice construct and because of different process schedulings.

Like for most of concurrent languages, the presence of guarded nondeterminism causes the denotational semantics of ccp to be rather complicated (see [3] and [16]), and therefore programs are difficult to analyze and to reason about. Various subsets of ccp, which admit a simpler semantics, have been investigated. In particular, determinate ccp [16], where no form of choice is allowed, and local-choice ccp [8], where choice is allowed, but it does not depend on the external environment (i.e., the choice of the branch does not depend on the present store: the guard is checked after the branch is selected). For both these two languages a denotational semantics based on closure operators has been given; in the first case for the notion of input-output observables, in the second case for the upward closure of such observables (which is a less refined notion).

In this paper, we investigate other subsets of ccp which allow for a simple semantics. The first one is what we call *confluent ccp*. The concept of confluence arises in various areas of computation theory, in different forms depending on the contexts. In Lambda Calculus and in Rewriting Theory confluence means that all (terminating) computations give the same results, even if they might follow different paths. This is a very strong notion of confluence, in fact it ensures determinacy, which is what is needed in order to regard these frameworks as foundations of Functional Programming. There is however also another aspect attached to this notion: confluence allows us to capture the meaning of a program by considering only one computation instead of all the possibly many different ones. It is worthwhile to explore in what forms this second aspect can be preserved when we consider languages that, for their nature, are intrinsically non-deterministic. Logic Programming, for instance, is a language for computing relations, hence it is natural to have different computations with different results. Yet, it is important to know that this indeterminacy arises only from the presence of alternative clauses for the same predicate, not from the choice of the atom to reduce (selection rule). This property of independence from the selection rule is what we could call confluence in the context of an intrinsically nondeterministic language. Also in Concurrency Theory confluence has been explored [11,12] as a notion which does not imply determinacy (in the sense that a process can be confluent and still have the choice of performing two different actions which lead to two unrelated continuations. Note that the word "determinacy" is used in [11,12] with a different meaning). It is difficult to compare this latter notion of confluence to Logic Programming, since it is based on the labeled transition system for CCS (where labels represent synchronous communication actions), however also in this case one of the features of this notion is the independence from the selection rule.

For ccp, which is based on asynchronous communication, and is more similar to (the process interpretation of) Logic Programming rather than to CCS, we define the notion of confluence as independence from the selection rule. We show that also for (structurally) confluent programs it is possible to define a denotational semantics along the lines of [8], allowing us to retrieve the same class of observables. This is an extension of the results in [8], because local-choice ccp programs are clearly confluent (in the same way that logic programs are), but the converse is not always true.

One of the drawbacks of the above notion of confluence, however, is that it is a co-semidecidable notion. That is, there might be programs which are confluent, but we are not able to prove it. It would be interesting, then, to determine syntactic restrictions which would ensure confluence. The only operator which is problematic for confluence in ccp is the guarded choice. Intuitively, the problem is that the choice depends on the store, and the content of the store is determined by the relative speed of the processes in the network. We study then a restricted form of guarded choice, which is still more expressive than local choice and still depends on the environment, but not at the price of confluence. For this second subset of ccp, which we call *admissible*, we prove that in the finite (non recursive) case, confluence is ensured. We conjecture that it would be ensured also in the infinite case, if we would allow the limit results of infinite computations to be taken into account (in this paper we consider finite computations only). However, the interest for admissible ccp comes also from the fact that we can develop a denotational model correct (in the sense of program analysis) wrt a notion of observables more refined than the one considered in [8]: the input-output relation. Moreover, we show that for a subset of admissible programs (*mutually-exclusive ccp*, which essentially coincides with local-choice ccp), this model is correct (in the standard sense) wrt the input-output relation.

Our interest for a simple denotational model is motivated by the possibility of using such model for the compositional analysis of ccp programs, following standard techniques from the theory of Abstract Interpretation [6]. Our results can also be exploited for full ccp: we can in fact approximate an arbitrary ccp program by a confluent or admissible program (in the sense that every result of the original program is also a result of the latter), and then analyze the latter. A similar transformation was independently proposed in [17].

The idea of using confluence for more efficient program analysis was investigated also in [4], in the context of concurrent logic programming (with atomic tell). In that paper, the focus was on the analysis based on operational semantics, where confluence is a convenient property because it ensures that it is sufficient to analyze only the computations which come from a particular selection rule.

Compositional analyses for ccp based on the denotational semantics have also been investigated in [5]. Also in that work programs are approximated so to simplify the semantic description. However our approach is orthogonal to the one followed there, in fact the approximation introduced in that paper corresponds to allowing the parallel components of a system to "restart" from scratch, whereas here the approximation consists on generating more possibilities in the choice points.

The next section contains preliminaries about ccp. In Section 3 we investigate confluent ccp and develop for it a denotational semantics which is correct and fully abstract with respect to the upward-closed observables. In Section 4 we investigate the admissible ccp programs and study for this class a denotational semantics which is correct (in the program analysis sense) with respect to the input-output observables. In Section 5 we consider a subset of confluent and admissible ccp, the mutually-exclusive ccp, and we show that previous semantics is correct with respect to it in the standard sense. Section 6 shows applications to program analysis.

2 Concurrent constraint programming

In this section we recall the definition of concurrent constraint programming, its operational semantics and observational behavior. We refer to [16] for more details.

2.1 Cylindric constraint systems

Concurrent constraint programming is based on the notion of constraint system. Here we consider an abstract definition of such systems as lattices, following [16] [4]. Other notions of constraint systems, like the one based on First

[4] We actually simplify the definition of [16] because we are interested only on finite observables. We don't require, for instance, the lattice to be complete.

Order Logic [9] can be seen as instances of this definition. All results of this paper still hold, of course, when more concrete systems are considered.

A *cylindric constraint system* is a structure $\mathbf{C} = \langle \mathcal{C}, \leq, \sqcup, \mathit{true}, \mathit{false}, \mathit{Var}, \exists, d\rangle$ such that

(i) $\langle \mathcal{C}, \leq, \sqcup, \mathit{true}, \mathit{false}\rangle$ is a lattice, where $\sqcup$ is the lub operation (representing the logical and), and *true*, *false* are the least and the greatest elements of $\mathcal{C}$, respectively [5]. The elements of $\mathcal{C}$ are called *constraints*.

(ii) *Var* is a denumerable set of *variables* (or *names*), and for each $x \in \mathit{Var}$ the function $\exists_x : \mathcal{C} \to \mathcal{C}$ is a *cylindrification operator* [7], i.e. it satisfies the following properties:
   (a) $\exists_x c \leq c$,
   (b) if $c \leq d$ then $\exists_x c \leq \exists_x d$,
   (c) $\exists_x(c \sqcup \exists_x d) = \exists_x c \sqcup \exists_x d$,
   (d) $\exists_x \exists_y c = \exists_y \exists_x c$.

(iii) For each $x, y \in \mathit{Var}$, $d_{xy} \in \mathcal{C}$ is a *diagonal element* [7], i.e. it satisfies the following properties:
   (a) $d_{xx} = \mathit{true}$,
   (b) if z is different from x, y then $d_{xy} = \exists_z(d_{xz} \sqcup d_{zy})$,
   (c) if x is different from y then $c \leq d_{xy} \sqcup \exists_x(c \sqcup d_{xy})$.

The cylindrification operators model a sort of existential quantification and are helpful for defining a hiding operator in the language. The diagonal elements are useful to model parameter passing. If $\mathbf{C}$ contains an equality theory, then the elements d_{xy} can be thought of as the formulas $x = y$.

2.2 The language

The syntax and semantics of ccp is parametric with respect to an underlying cylindric constraint system. Agents A and declarations D are described by the following abstract syntax, where c, c_i represent constraints.

$$A ::= \mathbf{Stop} \mid \mathbf{tell}(c) \mid \textstyle\sum_{i=1}^{n} \mathbf{ask}(c_i) \to A_i \mid A_1 \parallel A_2 \mid \exists_x A' \mid p(x)$$

$$D ::= \epsilon \mid p(x)\text{:-}A \mid D_1, D_2$$

We will also use $+$ as a shorthand for $\sum_{i=1}^{2}$.

The agent **Stop** represents inaction. The $\mathbf{ask}(c)$ and $\mathbf{tell}(c)$ operations act on a common *store* which ranges over $\mathcal{C}$. If d is the current store, then the

[5] The entailment relation $\vdash$, which is commonly used in the literature, is the reverse of $\leq$. Formally: for $c, d \in C$, $c \vdash d$ iff $d \leq c$.

execution of $\mathbf{tell}(c)$ adds c to the store, that is, it sets the store to be $c \sqcup d$. The $\mathbf{ask}(c)$ operation is a *guard* and its execution does not modify the store: it just tests the current store. We say that $\mathbf{ask}(c)$ is *enabled* in d if $c \leq d$. The *guarded choice* agent $\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i$ selects nondeterministically one $\mathbf{ask}(c_i)$ which is enabled in the current store, and then behaves like A_i. If no guards are enabled, then it *suspends*, waiting for other (parallel) agents to add information (constraints) to the store. Parallel composition of agents is represented by the symbol $\|$. We use $\exists_x$ (with a slight abuse of notation) also to indicate an *hiding operator* on agents: the intended meaning of $\exists_x A$ is that of an agent which behaves like A, but where x is considered *local* or *private* in A. Finally, the agent $p(x)$ is a procedure call, where p is the name of the procedure and x is the actual parameter. The meaning of $p(x)$ is given by a procedure declaration of the form $p(y)$:-A, where y is the formal parameter. In the following, we assume that for every procedure name there exists one and only one declaration in D.

2.3 The operational model

The operational model of ccp, informally introduced above, is described in terms of a transition system $T_D = (Conf, \longrightarrow)$ which is specified with respect to a given set of declarations (or *program*) D. The configurations in *Conf* are pairs consisting of an agent, and a constraint representing the store. Table 1 describes the rules of T_D.

Rule *R1* describes the behavior of an agent of the form $\mathbf{tell}(c)$: it adds c to the store and then stops. Rule *R2* describes the fact that a choice agent selects one of the branches whose guard is enabled. Note that this choice operator models global non-determinism: it depends on the current store whether or not a guard is enabled, and the current store is subject to modifications by the external environment. Rule *R3* describes parallelism as an interleaving of the steps performed by single agents. Rule *R4* describes locality. Here we found it convenient to extend the syntax by introducing agents of the form $\exists_x^d A$, which represent an agent A where x is local and d is the information that has been produced locally on x. The local store is assumed to be initially empty, which amounts to regarding $\exists_x A$ as equivalent to $\exists_x^{true} A$. The execution of a procedure call is modeled by *R5*. The symbol Δ_y^x stands for $\exists_\alpha^{d_{x\alpha}} \exists_y^{d_{\alpha y}}$, where α is assumed to occur neither in the declaration nor in the agent, and it is used to establish the link between the formal and the actual parameters.

Given an agent A and an initial store c, a *computation* from $\langle A, c\rangle$ is a sequence of transitions which starts from $\langle A, c\rangle$ and leads to a final configuration $\langle B, d\rangle$, final in the sense that no transitions can take place from $\langle B, d\rangle$. Note that all the basic sub-agents of a final configuration must either be suspended or

Table 1
The transition system T_D

R1	$\langle \mathbf{tell}(c), d\rangle \longrightarrow \langle \mathbf{Stop}, c \sqcup d\rangle$	
R2	$\langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i, d\rangle \longrightarrow \langle A_j, d\rangle$	$j \in [1, n]$ *and* $c_j \leq d$
R3	$\dfrac{\langle A, c\rangle \longrightarrow \langle A', c'\rangle}{\begin{array}{l}\langle A \parallel B, c\rangle \longrightarrow \langle A' \parallel B, c'\rangle \\ \langle B \parallel A, c\rangle \longrightarrow \langle B \parallel A', c'\rangle\end{array}}$	
R4	$\dfrac{\langle A, d \sqcup \exists_x c\rangle \longrightarrow \langle B, d' \sqcup \exists_x c\rangle}{\langle \exists_x^d A, c\rangle \longrightarrow \langle \exists_x^{d'} B, c \sqcup \exists_x d'\rangle}$	
R5	$\langle p(x), c\rangle \longrightarrow \langle \Delta_y^x A, c\rangle$	$p(y)$:-A is the declaration for p in D

inaction. By "basic agent" we mean all the agents whose rules in Table 1 do not have premises, i.e. the tell agent, the choice agent, the procedure call, and **Stop**. The collection of stores of such final configurations we call *observables* of A with respect to c:

Definition 1 *The observables of an agent A with respect to an initial store c are:*

$$\mathcal{O}(A, c) = \{d \mid \textit{there exists } B \textit{ s.t. } \langle A, c\rangle \longrightarrow^* \langle B, d\rangle \not\longrightarrow\}$$

where $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$.

Note that $\mathcal{O}(A, c)$ may be empty. This happens when all maximal sequences of transitions from $\langle A, c\rangle$ are infinite.

Example 2 *Consider a Herbrand constraint system, that is, a system whose constraints are equations, possibly existentially quantified, among terms on a given set of variables and data constructors. Note that the diagonal element d_{xy} corresponds to the equation $x = y$.*

The following declarations specify a procedure p which produces a list of a's of *arbitrary length, and a procedure c which consumes a list of a's.*

$$
\begin{aligned}
p(x) \;\text{:-}\; & (\mathbf{ask}(true) \rightarrow \exists_{x'}\, (\mathbf{tell}(x = [a|x']) \parallel p(x'))) \\
& + \\
& (\mathbf{ask}(true) \rightarrow \mathbf{tell}(x = [\,])),
\end{aligned}
$$

$$
\begin{aligned}
c(x) \;\text{:-}\; & (\mathbf{ask}(\exists_{x'}\; x = [a|x']) \rightarrow \exists_{x'}\, (\mathbf{tell}(x = [a|x']) \parallel c(x'))) \\
& + \\
& (\mathbf{ask}(x = [\,]) \rightarrow \mathbf{Stop})).
\end{aligned}
$$

Consider the network composed by the agents $p(x)$ and $c(x)$ in parallel, i.e. $p(x) \parallel c(x)$. The agent $p(x)$ will produce a list of a's on x and then stop. The agent $c(x)$ will consume such list and then stop. The computations of $p(x)$ and $c(x)$ may interleave in an arbitrary way; the only restriction is that c cannot be quicker than p, i.e. it cannot consume an item not yet produced by p. The observables are: $\mathcal{O}(p(x) \parallel c(x), true) = \{[\,], [a], [a, a], [a, a, a], \ldots\}$

2.4 Process scheduling and computation trees

Since we are interested in formalizing confluence, we need to introduce the notion of *process scheduling*, or *selection strategy*. Intuitively, a process scheduling is a particular selection of a basic sub-agent for each transition in a computation. This concept generalizes the notion of the *computation rule* of Logic Programming [10].

Definition 3 *Let $C \longrightarrow C'$ be a transition in T_D. The* selected *agent in C is the (basic) agent of C if the transition is obtained by Rule R1, R2, or R5. In case the transition is the consequence of Rule R4 or R3, then the selected agent in C is the selected agent of the initial configuration in the premise of the rule.*

We say that an agent A is enabled *in a configuration C if there is a transition from C with A as the selected agent.*

Once we fix a particular scheduling, still the transitions from an initial configuration form a branching structure, due to the different alternatives which may be possible for a choice agent. This branching structure we call *computation tree*. In general different process schedulings give rise to different computation trees.

Definition 4 *A* computation tree *for a configuration C and a set of declarations D is a maximal tree which has root C and in which each node C' is*

(i) a leaf node, if no agents in C' are enabled, or
(ii) a non-leaf node, with children $C''_1, \dots C''_n$, if for $i = 1, \dots, n$, $C' \longrightarrow C''_i$, are transitions in T_D with a fixed selected agent A which is enabled.

Example 5 *Consider the declarations*

$$p(x) \text{ :- } A$$
$$q(x) \text{ :- } B$$

where

$$A = (\mathbf{ask}(true) \rightarrow \mathbf{tell}(a)) + (\mathbf{ask}(true) \rightarrow \mathbf{tell}(b))$$

and

$$B = (\mathbf{ask}(a) \rightarrow \mathbf{tell}(c)) + (\mathbf{ask}(b) \rightarrow \mathbf{tell}(d))$$

with the assumption $true < a < b$, where the symbol "$<$" means "strictly smaller than". Figure 1 shows a computation tree for the configuration $\langle p(x) \parallel q(x), true\rangle$. We underline the occurrences of the selected agent and, for the sake of simplicity, we omit the existential quantifications due to the procedure calls. Note that the agent B cannot be selected before a or b are added to the store, because it is not enabled.

The finite branches of a computation tree represent computations in the sense previously defined. Note that a computation tree can have arbitrary combination of finite and infinite branches. For instance, all the computation trees for $\langle p(x) \parallel q(x), true\rangle$ in Example 2 will have an infinite branch corresponding to repeatedly selecting the first alternative in the definition of p.

$$\begin{tikzcd}[column sep=small]
 & \langle \underline{p(x)} \parallel q(x), true\rangle \arrow[d] & & & \\
 & \langle \underline{A} \parallel q(x), true\rangle \arrow[dl] \arrow[drr] & & & \\
\langle \underline{\mathbf{tell}(a)} \parallel q(x), true\rangle \arrow[d] & & & \langle \mathbf{tell}(b) \parallel \underline{q(x)}, true\rangle \arrow[d] & \\
\langle \mathbf{Stop} \parallel \underline{q(x)}, a\rangle \arrow[d] & & & \langle \underline{\mathbf{tell}(b)} \parallel B, true\rangle \arrow[d] & \\
\langle \mathbf{Stop} \parallel \underline{B}, a\rangle \arrow[d] & & & \langle \mathbf{Stop} \parallel \underline{B}, b\rangle \arrow[dl] \arrow[dr] & \\
\langle \mathbf{Stop} \parallel \underline{\mathbf{tell}(c)}, a\rangle \arrow[d] & & \langle \mathbf{Stop} \parallel \underline{\mathbf{tell}(c)}, b\rangle \arrow[d] & & \langle \mathbf{Stop} \parallel \underline{\mathbf{tell}(d)}, b\rangle \arrow[d] \\
\langle \mathbf{Stop} \parallel \mathbf{Stop}, a \sqcup c\rangle & & \langle \mathbf{Stop} \parallel \mathbf{Stop}, b \sqcup c\rangle & & \langle \mathbf{Stop} \parallel \mathbf{Stop}, b \sqcup d\rangle
\end{tikzcd}$$

Fig. 1. A computation tree

Let us denote by $Ctree_D(A, c)$ the set of all computation trees for a configuration $\langle A, c\rangle$ and a set of declarations D. By the above observation we have:

Remark 6 *Given a set of declarations D, an agent A, and an initial store c, we have*

$$\begin{aligned}\mathcal{O}(A, c) = \{d \mid & \textit{ there exists } T \in Ctree_D(A, c) \textit{ s.t.} \\ & \langle B, d\rangle \textit{ is a leaf of } T \textit{ for some agent } B\ \}\end{aligned}$$

3 Confluent ccp

One of the main aims of this paper is to study ccp programs which are *confluent*, in the sense that process scheduling does not affect their observed behavior. That is, for every process scheduling the set of results is exactly the same. In this section we investigate the class of structurally confluent ccp programs and we develop for this class a denotational semantics which is fully abstract with respect to the upward closure of the observables.

Definition 7 *Given a computation tree T, we denote by $Stores(T)$ the set of stores of the leaves of T, that is*

$$Stores(T) = \{d \mid \textit{there exists } B \textit{ s.t. } \langle B, d\rangle \textit{ is a leaf of } T\}$$

An agent A is confluent*, with respect to a program D, if for all constraints c, and for all $T \in Ctree_D(A, c)$, we have $\mathcal{O}(A, c) = Stores(T)$.*

Not all ccp agents are confluent, since different process schedulings may enable different guards in the same choice.

Example 8 *Assume $a < b < c$ and consider the following declaration D*

$$q(x) \text{ :- } (\ (\mathbf{ask}(true) \rightarrow \mathbf{tell}(a)) \ + \ (\mathbf{ask}(b) \rightarrow \mathbf{tell}(c))\ ) \ \| \ \mathbf{tell}(b)$$

Then $\mathcal{O}(q(x), true) = \{b, c\}$ while, for $T \in Ctree_D(q(x), true)$ obtained by considering a leftmost selection strategy, we have $Stores(T) = \{b\}$. Therefore the agent $q(x)$ is not confluent.

Another typical example of a non-confluent agent is the merge process given in Section 6.2.

Confluent agents do not have many structural properties; for instance, the parallel composition of two confluent agents is not necessarily confluent, and conversely the sub-agents of a confluent agent are not necessarily confluent. It turns out that these points are problematic for the development of a denotational semantics. We therefore consider a more restricted class, which we call *structurally confluent*. These are agents whose sub-agents are confluent, also when composed in parallel with $\mathbf{tell}(c)$, for arbitrary constraint c.

Definition 9 *We say that an agent A is* strongly confluent*, with respect to a program D, if for all constraints c the agent $A \parallel \mathbf{tell}(c)$ is confluent. We say that A (with respect to D) is* structurally confluent *if all of its sub-agents are strongly confluent. We say that D is structurally confluent if for each definition $p(x)$:-A in D, A is structurally confluent with respect to D.*

Note that by definition a procedure call is structurally confluent iff it is strongly confluent, thus for instance the declaration $p(x)$:-$p(x)$ is structurally confluent. On the contrary, the declaration $p(x)$:- $(\mathbf{ask}(a) \rightarrow \mathbf{tell}(a)) + (\mathbf{ask}(true) \rightarrow p(x))$ is not structurally confluent, because $p(x)$ is not strongly confluent: $\langle p(x) \parallel \mathbf{tell}(a), true\rangle$ gives no result if we keep on selecting $p(x)$, while $\mathcal{O}(p(x) \parallel \mathbf{tell}(a), true) = \{a\}$.

3.1 Denotational Semantics for structurally confluent agents

We now show that structurally confluent programs have a simple denotational model which is fully abstract with respect to the upward closure of the observables. Our interest in this notion is justified by the so-called *declarative*

Table 2
The denotational semantics for structurally confluent agents and programs

E1	$\mathcal{D}^u[\![\mathbf{Stop}]\!] = \mathcal{C}$
E2	$\mathcal{D}^u[\![\mathbf{tell}(c)]\!] = \uparrow c$
E3	$\mathcal{D}^u[\![\sum_{i=1}^n \mathbf{ask}(c_i) \rightarrow A_i]\!] = \bigcup_{i=1}^n (\uparrow c_i \cap \mathcal{D}^u[\![A_i]\!]) \cup \bigcap_{i=1}^n \overline{\uparrow c_i}$
E4	$\mathcal{D}^u[\![A \parallel B]\!] = \mathcal{D}^u[\![A]\!] \bigcap \mathcal{D}^u[\![B]\!]$
E5	$\mathcal{D}^u[\![\exists_x^c A]\!] = \exists_x^{-1} \exists_x (\mathcal{D}^u[\![A]\!] \cap \uparrow c)$
E6	$\mathcal{D}^u[\![p(x)]\!] = \mathcal{D}^u[\![\Delta_y^x A]\!]$ where $p(y)$:-A is the declaration of p in D

interpretation of ccp: In such an interpretation, a set of declarations is regarded as a logical theory. The "logical consequences" of this theory, for a given agent A and an initial store c, correspond to the constraints entailing the final results, namely to the upward closure of $\mathcal{O}(A, c)$.

Given a set of constraints $X \subseteq \mathcal{C}$, we denote by $\uparrow X$ (upward closure of X) the set $\{d \in \mathcal{C} \mid \text{ there exists } c \in X \ s.t. \ c \leq d\}$, and by $\overline{X}$ (complement of X) the set $\mathcal{C} \setminus X$. The upward closure of the observables is defined as $\mathcal{O}^u(A, c) = \uparrow \mathcal{O}(A, c)$. Furthermore, we extend the function $\exists_x$ to sets of constraints, that is, $\exists_x X = \{\exists_x c \mid c \in X\}$,and we define its inverse relation as $\exists_x^{-1} X = \{c \mid \exists_x c \in X\}$.

We now define the denotational model.

Definition 10 *Let $\mathcal{P}(\mathcal{C})$ denote the set of subsets of $\mathcal{C}$. The denotational semantics $\mathcal{D}^u$: Agents $\rightarrow \mathcal{P}(\mathcal{C})$ is the least function, with respect to the ordering induced by $\subseteq$, which satisfies the equations in Table 2.*

Intuitively, $\mathcal{D}^u[\![A]\!]$ represents the *possible resting points* (or *quiescent points*) of the process A, that is, those stores for which there exists a computation of A which does not affect them (more precisely, a computation from $\langle A, c\rangle$ which ends in a configuration with the same final store).

In the rest of this section we show that $\mathcal{D}^u$ is well-defined, that is, that the least function which satisfy the equations actually exists, and that $\mathcal{O}^u$ and $\mathcal{D}^u$

are equivalent for structurally confluent agents, in the sense that the one can be retrieved from the other. This implies the correctness of $\mathcal{D}^u$ and, trivially, the so-called "full abstraction" of $\mathcal{D}^u$.

In order to show these results, it is convenient to define a monotonic (and continuous) function associated to the program. This will allow us to exploit some standard results of fixed point theory. This function corresponds to the one-step inference operator of Logic Programming, and works on *interpretations*, namely functions which assign a meaning to agents. It will turn out that $\mathcal{D}^u$ is the least fixed point of this function.

Definition 11 *An interpretation is any function $I : Agents \to \mathcal{P}(\mathcal{C})$.*

We denote by $\mathcal{I}$ the set of all interpretations. With slight abuse of notation, we will also denote by $\subseteq$ the ordering on $\mathcal{I}$ induced by the ordering $\subseteq$ on $\mathcal{P}(\mathcal{C})$, and by $\cup$ the least upper bound operation.

Definition 12 *The mapping $\mathcal{F} : \mathcal{I} \to \mathcal{I}$, associated to the set of declarations D, is defined as follows:*

(i) $\mathcal{F}(I)(\mathbf{Stop}) = \mathcal{C}$,
(ii) $\mathcal{F}(I)(\mathbf{tell}(c)) = \uparrow c$,
(iii) $\mathcal{F}(I)(\sum_{i=1}^{n} \mathbf{ask}(c_i) \to A_i) = \bigcap_{i=1}^{n} \overline{\uparrow c_i} \,\cup\, \bigcup_{i=1}^{n}(\uparrow c_i \cap \mathcal{F}(I)(A_i))$,
(iv) $\mathcal{F}(I)(A \parallel B) = \mathcal{F}(I)(A) \bigcap \mathcal{F}(I)(B)$,
(v) $\mathcal{F}(I)(\exists_x^c A) = \exists_x^{-1}\exists_x(\mathcal{F}(I)(A) \cap \uparrow c)$,
(vi) $\mathcal{F}(I)(p(x)) = I(\Delta_y^x A)$, *where $p(y)$:-A is the declaration of p in D.*

Note the correspondence with Table 2. We have in fact the following characterization of the solutions of Equations *E1–E6*.

Proposition 13 *Let I be an interpretation. Then I is a solution of Equations E1–E6 iff I is a fixed point of $\mathcal{F}$.*

Proof. For the procedure call we have: $I(p(x)) = I(\Delta_y^x A)$ iff $I(p(x)) = \mathcal{F}(I)(p(x))$. The rest of the proof follows by an easy structural induction. Let us analyze the parallel operator; the other cases are similar.

$$\begin{aligned} I(A \parallel B) = I(A) \bigcap I(B) &\Leftrightarrow I(A \parallel B) = \mathcal{F}(I)(A) \bigcap \mathcal{F}(I)(B) \\ &\Leftrightarrow I(A \parallel B) = \mathcal{F}(I)(A \parallel B). \end{aligned}$$

□

It is easy to see that $\mathcal{F}$ is monotonic, hence by the theorem of Knaster-Tarski, it admits a least fixed-point. We thus can rephrase Definition 10 as follows.

Remark 14 *Let $[\mathcal{I} \to \mathcal{I}]$ denote the set of monotonic functions from $\mathcal{I}$ to $\mathcal{I}$, and lfp : $[\mathcal{I} \to \mathcal{I}] \to \mathcal{I}$ denote the least fixed point operator. Then $\mathcal{D}^u = \mathit{lfp}(\mathcal{F})$.*

This shows also that $\mathcal{D}^u$ is well-defined. For the correspondence with $\mathcal{O}^u$, it will be useful to introduce the *bottom-up iterations* (or *powers*) of $\mathcal{F}$:

Definition 15 *Let $I_\perp$ be the least interpretation, that is, the interpretation which maps every agent into $\emptyset$. Define the bottom-up iterations of $\mathcal{F}$ as:*

(i) $\mathcal{F} \Uparrow 0 = I_\perp$,
(ii) $\mathcal{F} \Uparrow n+1 = \mathcal{F}(\mathcal{F} \Uparrow n)$,
(iii) $\mathcal{F} \Uparrow \omega = \bigcup_n \mathcal{F} \Uparrow n$.

From the monotonicity of $\mathcal{F}$, it follows that

Remark 16 $\mathcal{F} \Uparrow \omega \subseteq \mathit{lfp}(\mathcal{F})$.

In order to show the correspondence of $\mathit{lfp}(\mathcal{F})$ with the upward-closed observables, it is convenient to formalize the notion of *quiescent points*, informally introduced before. We will show that the *quiescent points* actually coincide with $\mathit{lfp}(\mathcal{F})$ (hence with $\mathcal{D}^u$), and that from them we can retrieve the upward-closed observables, and vice-versa.

Definition 17 *The quiescent points of an agent A are the set*

$$\mathit{Quie}(A) = \{c \in \mathcal{C} \mid c \in \mathcal{O}(A, c)\}.$$

Note that the function *Quie* has type $\mathit{Agents} \to \mathcal{P}(\mathcal{C})$, hence it is an interpretation. The next lemma states a property of structurally confluent ccp agents which is fundamental both for the equality between *Quie* and $\mathit{lfp}(\mathcal{F})$ and for the correspondence between *Quie* and the observables. Essentially it says that, in structurally confluent ccp, if d is the result of a computation which starts with an input c, then when we start with an input smaller than c it is always possible to construct a computation whose result approximates d. Note that this is not the case for full ccp, because a smaller input can force the activation of a branch which brings to a greater or incomparable result (see Example 19).

Lemma 18 *Let A be a structurally confluent agent, let $c \in \mathcal{C}$, and let ξ be a computation $\langle A, c\rangle \longrightarrow^* \langle B, d\rangle \not\longrightarrow$. Let $c' \in \mathcal{C}$ with $c' \le c$. Then there exists a computation ξ': $\langle A, c'\rangle \longrightarrow^* \langle B', d'\rangle \not\longrightarrow$, such that $d' \le d$.*

Proof. Consider the configuration $\langle A \parallel \mathbf{tell}(c), c'\rangle$. By selecting the agent $\mathbf{tell}(c)$, we have the transition $\langle A \parallel \mathbf{tell}(c), c'\rangle \longrightarrow \langle A \parallel \mathbf{Stop}, c\rangle$. From the last configuration, by mimicking ξ, we obtain a computation $\langle A \parallel \mathbf{Stop}, c\rangle \longrightarrow^*$

$\langle B \parallel \mathbf{Stop}, d\rangle \not\longrightarrow$. Hence we conclude that $d \in \mathcal{O}(A \parallel \mathbf{tell}(c), c')$. Consider now a selection strategy which delays the selection of $\mathbf{tell}(c)$ as much as possible, i.e. until A has reduced to some suspended agent B'. Since A is strongly confluent, there must exist a sequence of transitions ξ'' of the form: $\langle A \parallel \mathbf{tell}(c), c'\rangle \longrightarrow^* \langle B' \parallel \mathbf{tell}(c), d'\rangle$, with $d \in \mathcal{O}(B' \parallel \mathbf{tell}(c), d')$. Given that transitions can only increase or leave unchanged the store, we derive $d' \leq d$. The intended ξ' can now be obtained by mimicking ξ'' in the obvious way. □

The previous lemma does not hold for full ccp. The following is a counterexample.

Example 19 *Let $a, b, c \in \mathcal{C}$ with $a < b < c$. Consider the ccp agent*

$$
\begin{aligned}
A = \; & \mathbf{ask}(b) \rightarrow \mathbf{tell}(b) \\
& + \\
& \mathbf{ask}(a) \rightarrow \mathbf{tell}(c).
\end{aligned}
$$

Then $\langle A, b\rangle$ has a computation with final store b whereas $\langle A, a\rangle$ has only one computation with final store c.

Next we show that $Quie$ is $\mathcal{F}$-closed.

Proposition 20 *For every structurally confluent program and every structurally confluent agent A, $Quie(A)$ is $\mathcal{F}$-closed, that is, $\mathcal{F}(Quie)(A) \subseteq Quie(A)$ holds.*

Proof. By structural induction on A.

$A = \mathbf{Stop}$. Obvious.
$A = \mathbf{tell}(c)$. Obvious.
$A = \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i$. Assume $c \in \mathcal{F}(Quie)(A)$. Then either $c \in \bigcap_{i=1}^{n} \overline{\uparrow c_i}$ or there exists $i \in [1, n]$ such that $c \in \uparrow c_i \cap \mathcal{F}(Quie)(A_i)$, that is, $c_i \leq c$ and $c \in \mathcal{F}(Quie)(A_i)$. In the first case $\langle A, c\rangle$ suspends and therefore c is a resting point. In the second case, we have a transition $\langle A, c\rangle \longrightarrow \langle A_i, c\rangle$. By the inductive hypothesis, $c \in Quie(A_i)$. Therefore $c \in Quie(A)$.
$A = A_1 \parallel A_2$. Assume $c \in \mathcal{F}(Quie)(A)$. Then $c \in \mathcal{F}(Quie)(A_1)$ and $c \in \mathcal{F}(Quie)(A_2)$. By the inductive hypothesis, $c \in Quie(A_1)$ and $c \in Quie(A_2)$. By executing first $\langle A_1, c\rangle$ until it arrests, and then $\langle A_2, c\rangle$, we obtain a computation for $\langle A, c\rangle$ which has c as a resting point.
$A = \exists_x^e B$. This is the only case in which we need the structural confluence hypothesis. Let $c \in \mathcal{F}(Quie)(\exists_x^e B)$. Then there exists $d \in \mathcal{F}(Quie)(B) \cap \uparrow e$ such that $\exists_x c = \exists_x d$. Observe that by the inductive hypothesis $d \in Quie(B)$ holds. Hence there exists a computation $\langle B, d\rangle \longrightarrow^* \langle B', d\rangle \not\longrightarrow$. Since $e \sqcup$

$\exists_x c \leq d$, we can apply Lemma 18: There exists a computation $\langle B, e \sqcup \exists_x c\rangle \longrightarrow^* \langle B'', d' \sqcup \exists_x c\rangle \not\longrightarrow$, for some B'' and d', with $d' \leq d$. It follows that $\langle \exists_x^e B, c\rangle \longrightarrow^* \langle \exists_x^{d'} B'', c \sqcup \exists_x d'\rangle \not\longrightarrow$ is also a computation. Finally observe that, since $d' \leq d$, we have $\exists_x d' \leq \exists_x d = \exists_x c \leq c$ (by properties (a) and (b) of cylindrification operators), from which we obtain $c \sqcup \exists_x d' = c$.

$A = p(x)$. Assume $c \in \mathcal{F}(Quie)(p(x))$. Then $c \in Quie(\Delta_y^x B)$ holds, where $p(y)$:-B is the declaration for p in D. Hence $c \in Quie(p(x))$. $\square$

The Theorem of Knaster-Tarski ensures that the least fixed point of $\mathcal{F}$ coincides with the least $\mathcal{F}$-closed interpretation, that is, $lfp(\mathcal{F})(A) = min\{I \mid \mathcal{F}(I) \subseteq I\}$. Hence from Proposition 20 we have:

Corollary 21 *For every structurally confluent program and every structurally confluent agent A, $lfp(\mathcal{F})(A) \subseteq Quie(A)$ holds.*

We show now that $Quie(A) \subseteq \mathcal{F} \Uparrow \omega(A)$. This result actually holds for full ccp.

Proposition 22 *For every ccp program and every ccp agent A, $Quie(A) \subseteq \mathcal{F} \Uparrow \omega(A)$ holds.*

Proof. Assume $c \in Quie(A)$. Let ξ be a computation

$$\gamma_0 \longrightarrow \gamma_1 \longrightarrow \ldots \gamma_i \not\longrightarrow$$

with $i \geq 0$, such that $\gamma_0 = \langle A, c\rangle$ and $\gamma_i = \langle B, c\rangle$ for some B. We proceed by simultaneous induction on the length of the computation i, and on the structure of A.

$A = \mathbf{Stop}$. Immediate since $c \in \mathcal{F} \Uparrow 1(A)$.

$A = \mathbf{tell}(c)$. Immediate since $c \in \mathcal{F} \Uparrow 1(A)$.

$A = \sum_{i=1}^n \mathbf{ask}(c_i) \rightarrow A_i$. If $\langle A, c\rangle$ suspends then $c \in \bigcap_{i=1}^n \overline{\uparrow c_i}$ and therefore $c \in \mathcal{F} \Uparrow 1(A)$. Otherwise, there exists $i \in [1, n]$ such that $c_i \leq c$, $\langle A, c\rangle \longrightarrow \langle A_i, c\rangle$ is the first step of ξ, and $c \in Quie(A_i)$. By the (structural) inductive hypothesis, there exists n such that $c \in \mathcal{F} \Uparrow n(A_i)$, and therefore $c \in \mathcal{F} \Uparrow n(A)$.

$A = A_1 \parallel A_2$. Since $c \in Quie(A_1 \parallel A_2)$, there is a computation for $\langle A_1 \parallel A_2, c\rangle$ in which A_1 and A_2 do some steps, without changing the store, and then they both arrest. Hence both $c \in Quie(A_1)$ and $c \in Quie(A_2)$ hold. By the (structural) inductive hypothesis, there exist n_1, n_2 such that $c \in \mathcal{F} \Uparrow n_1(A_1)$ and $c \in \mathcal{F} \Uparrow n_2(A_2)$. Since $\{\mathcal{F} \Uparrow i\}_i$ is an increasing chain, we have $c \in \mathcal{F} \Uparrow n(A_1)$ and $c \in \mathcal{F} \Uparrow n(A_2)$, where $n = max\{n_1, n_2\}$. Therefore $c \in \mathcal{F} \Uparrow n(A_1 \parallel A_2)$.

$A = \exists_x^e B$. Since $c \in Quie(\exists_x^e B)$, there exists a computation $\langle B, e \sqcup \exists_x c\rangle \longrightarrow^* \langle B', d \sqcup \exists_x c\rangle \not\longrightarrow$, for some d such that $\exists_x d \leq c$. Hence $d \sqcup \exists_x c \in Quie(B)$. By the (structural) inductive hypothesis, there exists n such that $d \sqcup \exists_x c \in \mathcal{F} \Uparrow n(B)$. Furthermore, note that $e \leq d \leq d \sqcup \exists_x c$. Finally observe that $\exists_x(d \sqcup \exists_x c) = \exists_x d \sqcup \exists_x c = \exists_x(c \sqcup \exists_x d) = \exists_x c$.

$A = p(x)$. Let ξ' be the postfix of ξ starting from γ_1, which will be of the form $\langle \Delta_y^x B, c\rangle$, where $p(x)$:-B is the declaration for p in D. Then ξ' is a computation for γ_1 ending with store c, hence $c \in Quie(\Delta_y^x B)$. Furthermore the length of ξ' is $i-1$, so by the inductive hypothesis there exists n such that $c \in \mathcal{F} \Uparrow n(\Delta_y^x B)$. Therefore $c \in \mathcal{F} \Uparrow n+1(A)$. $\square$

By Corollary 21, Proposition 22 and Remark 16 we have $Quie(A) \subseteq \mathcal{F} \Uparrow \omega(A) = \mathcal{D}^u(A) \subseteq lfp(\mathcal{F}) \subseteq Quie(A)$. Hence we can conclude:

Theorem 23 *For every structurally confluent ccp agent A, $Quie(A) = \mathcal{D}^u(A)$.*

This proves also the continuity of $\mathcal{F}$. We show now the relation between $\mathcal{O}^u$ and $Quie$. In general, $Quie$ can be retrieved from $\mathcal{O}^u$. For structurally confluent ccp also the reverse holds, that is, $\mathcal{O}^u$ can be retrieved from $Quie$.

Proposition 24

(i) For every ccp agent A, for every $d \in \mathcal{C}$, we have $\mathcal{O}^u(A, d) \subseteq \uparrow (\uparrow d \cap Quie(A))$.

(ii) For every structurally confluent program and every structurally confluent agent A, and for every $d \in \mathcal{C}$, we have $\uparrow (\uparrow d \cap Quie(A)) \subseteq \mathcal{O}^u(A, d)$.

Proof.

(i) If $d' \in \mathcal{O}^u(A, d)$ then there exists $d'' \in \mathcal{O}(A, d)$ such that $d \leq d'' \leq d'$. Let ξ be the computation for $\langle A, d\rangle$ which results in the store d''. By starting from $\langle A, d''\rangle$, we can mimic ξ and obtain the same final configuration. Hence $d'' \in Quie(A)$ holds. Therefore $d' \in \uparrow (\uparrow d \cap Quie(A))$ holds.

(ii) Let $c \in \uparrow (\uparrow d \cap Quie(A))$. Then there exists $c' \leq c$ such that $c' \in \uparrow d \cap Quie(A)$. By Lemma 18, there exists $d' \in \mathcal{O}(A, d)$ such that $d' \leq c'$. Therefore $c' \in \mathcal{O}^u(A, d)$, hence, a fortiori, $c \in \mathcal{O}^u(A, d)$. $\square$

From Proposition 24 and Theorem 23 we finally obtain:

Theorem 25 (Correctness) *For every structurally confluent program and every structurally confluent agent A, and for every constraint c, $\mathcal{O}^u(A, c) = \uparrow (\uparrow c \cap \mathcal{D}^u(A))$ holds.*

Observe that also $\mathcal{D}^u$ can be retrieved from $\mathcal{O}^u$, in fact for each ccp agent A and constraint c, $Quie(A) = \{c \in \mathcal{C} \mid c \in \mathcal{O}^u(A, c)\}$ holds. Therefore by Theorem 23 we have:

Theorem 26 (Completeness) *For every structurally confluent ccp agent A and constraint c, $\mathcal{D}^u(A) = \{c \in \mathcal{C} \mid c \in \mathcal{O}^u(A, c)\}$ holds.*

These two results imply that $\mathcal{D}^u$ and $\mathcal{O}^u$ are equivalent, that is, they induce the same equivalence relation on programs. This implies also the full abstraction of $\mathcal{D}^u$ with respect to $\mathcal{O}^u$, in a trivial sense.

We conclude this section with the observation that, for generic ccp agents, it is not possible to retrieve $\mathcal{O}^u$ from $\mathcal{D}^u$, that is, $\mathcal{D}^u$ is not correct with respect to $\mathcal{O}^u$:

Example 27 *Let $\mathcal{C} = \{true, a, b, false\}$ with $a \sqcup b = false$. Consider the agents A_1 and A_2:*

$$A_1 = (\mathbf{ask}(true) \rightarrow \mathbf{tell}(a)) + (\mathbf{ask}(b) \rightarrow \mathbf{tell}(b))$$

$$A_2 = (\mathbf{ask}(true) \rightarrow \mathbf{tell}(a)) + (\mathbf{ask}(true) \rightarrow \mathbf{tell}(b)).$$

Then $\mathcal{D}^u[\![A_1]\!] = \mathcal{D}^u[\![A_2]\!] = \{a, b, false\}$. However, we have $\mathcal{O}^u(A_1, true) = \{a, false\}$ and $\mathcal{O}^u(A_2, true) = \{a, b, false\}$.

4 Admissible agents

The main cause of non-confluency, in a ccp program, is the presence of the global choice operator. In this section we study a restricted form of global choice which (in the finite case) ensures structural confluence. We then identify a class of ccp programs, the *admissible* programs, which can be constructed in a "bottom-up" way by using this restricted choice and the other standard operators. The interest for this class comes also from the fact that it admits a simple denotational model for the input-output relation $\mathcal{O}$, which is a more refined notion of observables than the one considered in the previous section.

We first need to define the following equivalence.

Definition 28 *The agents A and A' are* equivalent*, written $A \approx A'$, if for any $c \in \mathcal{C}$, $\mathcal{O}(A, c) = \mathcal{O}(A', c)$ holds.*

Intuitively, a guarded choice G is admissible if either G has the same guard on any branch, or, whenever two guards are both satisfied, the two related branches are equivalent.

Definition 29 *Let G be the guarded choice $\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i$.*

(i) *G is* local *if for all $i, j \in \{1, .., n\}$, $c_i = c_j$.*
(ii) *G is* compatible *if G is not local and for all $i, j \in \{1, .., n\}$, $\mathbf{tell}(c_i \sqcup c_j) \parallel A_i \approx \mathbf{tell}(c_i \sqcup c_j) \parallel A_j$.*
(iii) *G is* admissible *if it is either local or compatible.*
(iv) *A program D is* admissible *if any choice in D is admissible. An agent A is* admissible *(with respect to a program D) if any choice in A is admissible, and D is admissible.*

We now show that finite admissible agents and programs are structurally confluent. To prove this, we will show that admissible choice and all the other operators of ccp (apart from recursion) preserve the structural confluence property.

Here and in the sequel we denote by $\langle A, c\rangle \longrightarrow_{i=1}^{n} T_i$ a computation tree whose root is $\langle A, c\rangle$ and such that the children of the root are the roots of the (sub-)trees $T_1, \ldots, T_n$. When $n = 1$ we omit the subscript $i = 1$ and the superscript 1. Moreover we denote by $Root(T)$ the root of the tree T.

Lemma 30 *Let A be a structurally confluent agent and c, d be constraints. Then, for each $T \in Ctree(A, c \sqcup d)$ and $T' \in Ctree(A \parallel tell(c), d)$, $Stores(T) = Stores(T')$ holds.*

Proof. Consider the tree $T_1 = \langle A \parallel tell(c), d\rangle \longrightarrow T_2$, where $Root(T_2) = \langle A \parallel \mathbf{Stop}, c \sqcup d\rangle$, and T_2 is isomorphic to T, in the sense that the only difference is the presence of the parallel agent **Stop** in the nodes. By definition we have $T_2 \in Ctree(A \parallel \mathbf{Stop}, c \sqcup d)$ and therefore $T_1 \in Ctree(A \parallel tell(c), d)$. Moreover T_1 and T_2 have the same leaves, hence $Stores(T_1) = Stores(T)$. Since A is structurally confluent we have, for each $T' \in Ctree(A \parallel tell(c), d)$, that $Stores(T') = Stores(T_1)$ holds. $\Box$

Lemma 31 *Let G be the guarded choice $\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i$. If, for each $i \in [1, n]$ the agent A_i is structurally confluent, and G is admissible, then G is structurally confluent.*

Proof. Consider two generic trees $T, T' \in Ctree(G \parallel tell(c), d)$ where $c, d \in \mathcal{C}$. We have the following two cases.

(i) There exist no $i \in [1, n]$ such that $c_i \leq d$. In this case the root of each tree in $Ctree(G \parallel tell(c), d)$ has only one outgoing arc of the form

$$\langle G \parallel tell(c), d\rangle \longrightarrow \langle G, d \sqcup c\rangle.$$

If $c_i \not\leq d \sqcup c$ for any $i \in [1, n]$, then $Ctree(G \parallel tell(c), d)$ contains only one tree and the thesis holds vacuously. Otherwise assume that $c_{j_1}, \ldots, c_{j_i} \leq d \sqcup c$ for some $\{j_1, \ldots, j_i\} \subseteq [1, n]$. Then T has the form

$$\langle G \parallel tell(c), d\rangle \longrightarrow \langle G, d \sqcup c\rangle \longrightarrow_{k=1}^{i} T_{j_k}$$

while T' has the form

$$\langle G \parallel tell(c), d\rangle \longrightarrow \langle G, d \sqcup c\rangle \longrightarrow_{k=1}^{i} T'_{j_k}$$

with $Root(T_{j_k}) = Root(T'_{j_k}) = \langle A_{j_k}, d \sqcup c\rangle$. Since A_{j_k} is structurally confluent, for each $k \in [1, i]$ we have $Stores(T_{j_k}) = Stores(T'_{j_k})$. Therefore $Stores(T) = Stores(T')$ holds.

(ii) There exists $\{j_1, \ldots, j_i\} \subseteq [1, n]$ such that $c_{j_1}, \ldots, c_{j_i} \leq d$. If both T and T' are obtained by rewriting the same parallel component of the root $\langle G \parallel tell(c), d\rangle$ then the proof is the same as the one given for the previous case. Therefore we can restrict, without loss of generality, to the case in which T has the form

$$\langle G \parallel tell(c), d\rangle \longrightarrow_{k=1}^{i} T_{j_k},$$

with $Root(T_{j_k}) = \langle A_{j_k} \parallel tell(c), d\rangle$, and T' has the form

$$\langle G \parallel tell(c), d\rangle \longrightarrow \langle G, d \sqcup c\rangle \longrightarrow_{k=1}^{m} T'_{j_k}$$

with $m \geq i$, $Root(T'_{j_k}) = \langle A_{j_k}, d \sqcup c\rangle$, and $c_{j_k} \leq c \sqcup d$ for each $k \in [1, m]$.

By Lemma 30 we have that, for each $k \in [1, i]$,

$$Stores(T_{j_k}) = Stores(T'_{j_k}). \tag{1}$$

Since the outermost choice in G is admissible, we have now the following two cases:

(a) If G is local then $i = m$ $(= n)$. Then from (1) it follows that $Stores(T) = Stores(T')$.

(b) If G is compatible, then for each $h, k \in [1, m]$ we have $Stores(T'_{j_h}) = Stores(T'_{j_k})$. Thus also in this case from (1) it follows that $Stores(T) = Stores(T')$ and this completes the proof. □

Lemma 32 *If the agents A and B are structurally confluent and the program is structurally confluent then also the agent $A \parallel B$ is structurally confluent.*

Proof. This result essentially follows from the observation that if A and B are structurally confluent, and the program is structurally confluent, then for every sequence of derivations $\langle A \parallel B, c\rangle \longrightarrow^* \langle A' \parallel B', c'\rangle$, A' and B' are also structurally confluent. Then apply induction on the height of the computation tree (which must be finite). $\Box$

Lemma 33 *If the agent A is structurally confluent then also the agent $\exists_x A$ is structurally confluent.*

Proof. Obvious: just observe that each tree $T \in \mathit{Ctree}(\exists_x A \parallel \mathbf{tell}(c), d)$ corresponds to a tree $T' \in \mathit{Ctree}(A \parallel \mathbf{tell}(\exists_x(c \sqcup d)), \exists_x d)$ and that for each constraint $e \in \mathit{Stores}(T)$ there exists a constraint $e' \in \mathit{Stores}(T')$ such that $e = c \sqcup \exists_x e'$. $\Box$

Since the basic agents **Stop** and $\mathbf{tell}(c)$ are structurally confluent, we can now prove that finite (i.e. non recursive) admissible ccp agents are structurally confluent:

Theorem 34 *If A is an admissible agent with respect to a non-recursive program D, then A is structurally confluent.*

Proof. We prove this result by structural induction on A. Since the program is non-recursive, A can be finitely unfolded into an agent which does not contain procedure calls, hence we do not need to consider the case of the procedure call.

$A = \mathbf{Stop}$. Obvious.
$A = \mathbf{tell}(c)$. Obvious.
$A = \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i$. Apply the inductive hypothesis and Lemma 31.
$A = A_1 \parallel A_2$. Apply the inductive hypothesis and Lemma 32.
$A = \exists_x^d B$. Apply the inductive hypothesis and Lemma 33. $\Box$

This theorem does not hold for recursive programs. For instance, the declaration $p(x)$:- $(\mathbf{ask}(a) \rightarrow \mathbf{tell}(a)) + (\mathbf{ask}(\mathit{true}) \rightarrow p(x))$ considered in the previous section is admissible but not structurally confluent. One might object that the selection rule for which $\langle p(x) \parallel \mathbf{tell}(a), \mathit{true}\rangle$ gives no result is an unfair one, and that the problem might be fixed if we restrict our attention to fair selection rules. Unfortunately, this is not the case. The following is a counterexample.

Example 35 *Consider the declaration*

$$\begin{aligned} p(x) :\!-\, & (\mathbf{ask}(x = a) \to \mathbf{tell}(ok)) \\ & + \\ & (\mathbf{ask}(true) \to (p(x)||\exists_y(p(y)||\mathbf{tell}(y = a)))) \end{aligned}$$

We have that $p(x)$ is admissible, in fact $\mathbf{tell}(a) \parallel p(x) \parallel \exists_y(p(y) \parallel \mathbf{tell}(y = a))$ is equivalent (gives the same results) as $\mathbf{tell}(a) \parallel \mathbf{tell}(ok)$. But $p(x)$ is not structurally confluent, in fact $\langle p(x) \parallel \mathbf{tell}(a), true\rangle$ gives $\{a\}$ with the leftmost selection rule, and gives no result when we consider a rule which always selects $p(z)$ before the corresponding $\mathbf{tell}(z = a)$. Notice that the latter selection rule is fair, although it gives rise to a tree without finite branches.

We think that if we would modify the definition of the observables so to include also the result of infinite computations (defined as the limit of the intermediate stores) then the admissibility condition would imply structural confluence, also in the case of recursive programs. This conjecture is however out of the scope of this paper since we are concerned here only with finite computations. We leave it as an open problem.

Another problem of recursive programs is that it is not decidable whether two agents are equivalent, therefore it is not decidable whether an agent is admissible. In Section 5 we will make a further restriction on the kind of choice, which leads to a decidable subclass of admissible agents, and which ensures structural confluence (up to the identification of divergence and inconsistency). For the next section, however, we still consider the larger class of admissible programs since, as we will see, the condition of admissibility is sufficient for obtaining a simple denotational characterization of the input-output observables.

4.1 Denotational semantics for admissible agents

The main reason why the previous semantics allows us to retrieve $\mathcal{O}^u$, but not $\mathcal{O}$, is that it associates only one set of resting points to an agent, by putting together the resting points which come from different branches, regardless of whether they represent "a contribution by the agent" or simply "no reaction". For instance, $\mathcal{D}^u$ identifies the agents $A = (\mathbf{ask}(true) \to \mathbf{tell}(true)) + (\mathbf{ask}(true) \to \mathbf{tell}(a))$ and $B = (\mathbf{ask}(true) \to \mathbf{tell}(true))$, whereas $\mathcal{O}(A, true) = \{true, a\}$ and $\mathcal{O}(B, true) = \{true\}$.

To get around this limitation, we consider here a richer denotational domain,

Table 3
The denotational semantics for admissible agents and for the input-output relation

$E1'$	$\mathcal{D}[\![\mathbf{Stop}]\!] = \{\mathcal{C}\}$
$E2'$	$\mathcal{D}[\![\mathbf{tell}(c)]\!] = \{\uparrow c\}$
$E3'$	$\mathcal{D}[\![\sum_{i=1}^{n} \mathbf{ask}(c_i) \to A_i]\!] = \{(\uparrow c_j \cap X_j) \cup \bigcap_{i=1}^{n} \overline{\uparrow c_i} \mid X_j \in \mathcal{D}[\![A_j]\!] \text{ for } j \in [1,n]\} \cup \{\bigcap_{i=1}^{n} \overline{\uparrow c_i}\}$
$E4'$	$\mathcal{D}[\![A \parallel B]\!] = \{X \cap Y \mid X \in \mathcal{D}[\![A]\!], Y \in \mathcal{D}[\![B]\!]\}$
$E5'$	$\mathcal{D}[\![\exists_x^c A]\!] = \{\exists_x^{-1} \exists_x (X \cap \uparrow c) \mid X \in \mathcal{D}[\![A]\!]\}$
$E6'$	$\mathcal{D}[\![p(x)]\!] = \mathcal{D}[\![\Delta_y^x A]\!]$ where $p(y)$:-A is the declaration of p in D

namely we associate to an agent a set of sets of constraints. Intuitively, each set of constraints represents the resting points associated to a specific computation branch. Here and in the sequel we assume the set of declarations D to be fixed.

Definition 36 $\mathcal{D}$: *Agents* $\to \mathcal{P}(\mathcal{P}(\mathcal{C}))$ *is the least function, with respect to the ordering induced by* $\subseteq$*, which satisfies the equations in Table 3.*

In order to prove that the least function satisfying the equations in Table 3 actually exists, and the relation of $\mathcal{D}$ with the observables, we use again fixed point theory. Let us first define the new notion of interpretation as a function I : *Agents* $\to \mathcal{P}(\mathcal{P}(\mathcal{C}))$, let us denote again by $\mathcal{I}$ the set of all these (new) interpretations, by $\subseteq$ the ordering induced on $\mathcal{I}$ by set inclusion on $\mathcal{P}(\mathcal{P}(\mathcal{C}))$, and by $\cup$ the least upper bound operation.

Like before, we consider a monotonic mapping $\mathcal{T}$ on interpretations, associated to the program D, and defined in such a way that its fixed points are the solutions of Equations $E1'$–$E6'$.

Definition 37 *The mapping* $\mathcal{T}$: $\mathcal{I} \to \mathcal{I}$ *is defined as follows:*

(i) $\mathcal{T}(I)(\mathbf{Stop}) = \{\mathcal{C}\}$
(ii) $\mathcal{T}(I)(\mathbf{tell}(c)) = \{\uparrow c\}$

(iii) $\mathcal{T}(I)(\sum_{i=1}^{n} \mathbf{ask}(c_i) \to A_i) = \{(\uparrow c_j \cap X_j) \cup \bigcap_{i=1}^{n} \overline{\uparrow c_i} \mid X_j \in \mathcal{T}(I)(A_j) \text{ for } j \in [1,n]\} \cup \bigcap_{i=1}^{n} \overline{\uparrow c_i}$

(iv) $\mathcal{T}(I)(A \parallel B) = \{X \cap Y \mid X \in \mathcal{T}(I)(A), Y \in \mathcal{T}(I)(B)\}$

(v) $\mathcal{T}(I)(\exists_x^c A) = \{\exists_x^{-1} \exists_x (X \cap \uparrow c) \mid X \in \mathcal{T}(I)(A)\}$

(vi) $\mathcal{T}(I)(p(x)) = I(\Delta_y^x A)$ *where* $p(y)$:-A *is the declaration of* p *in* D.

The following proposition shows that indeed the solutions of Equations $E1'$–$E6'$ are the fixed points of $\mathcal{T}$.

Proposition 38 *An interpretation* I *is a solution of the Equations* $E1'$–$E6'$ *iff* $\mathcal{T}(I) = I$.

Proof. For the procedure call we have that $I(p(x)) = I(\Delta_y^x A)$ iff $I(p(x)) = \mathcal{T}(I)(p(x))$. The rest of the proof is straightforward by structural induction. $\Box$

We define the powers of the operator $\mathcal{T}$ as before, namely

(i) $\mathcal{T} \Uparrow 0 = I_\perp$,
(ii) $\mathcal{T} \Uparrow n+1 = \mathcal{T}(\mathcal{T} \Uparrow n)$,
(iii) $\mathcal{T} \Uparrow \omega = \bigcup_n \mathcal{T} \Uparrow n$

where $I_\perp$ is the least interpretation, namely the interpretation which maps each agent into the empty set (the only difference with respect to the analogous definition for $\mathcal{F}$ is that $I_\perp(A)$ here has a different type: it is a empty set *of sets*).

To show the intended results, we follow an approach different from previous section. First we show the continuity of $\mathcal{T}$:

Proposition 39 $(\mathcal{I}, \subseteq)$ *is a complete lattice and* $\mathcal{T}$ *is continuous.*

Proof. The fact that $(\mathcal{I}, \subseteq)$ is a complete lattice follows just by standard Set Theory. The continuity of $\mathcal{T}$ can be easily shown by structural induction and using the observation that (by definition) $\mathcal{T}(\bigcup_{k\in\omega} I_k)(p(x)) = (\bigcup_{k\in\omega} I_k)(\Delta_y^x A) = \bigcup_{k\in\omega} I_k(\Delta_y^x A) = \bigcup_{k\in\omega} \mathcal{T}(I_k)(p(x))$. $\Box$

Thus, from standard results, we have that the least fixed point of $\mathcal{T}$ exists and it coincides with $\mathcal{T} \Uparrow \omega$. From Proposition 38 we then obtain that $\mathcal{D}$ is well-defined, and that:

Corollary 40 *For each agent* A *we have* $\mathcal{D}[\![A]\!] = \mathcal{T} \Uparrow \omega(A)$.

By using this characterization we show now the relation between $\mathcal{D}$ and the input-output observables.

Lemma 41 *If $\langle A, c\rangle$ is a configuration such that $\langle A, c\rangle \not\longrightarrow$, then, for any natural number n and for any $X \in \mathcal{T} \Uparrow n(A)$, $c \in X$ holds.*

Proof. By structural induction on the agent A. We have only the following cases.

$A = \textbf{Stop}$. Obvious, since $\mathcal{T} \Uparrow n(\textbf{Stop}) = \{\mathcal{C}\}$.
$A = \sum_{i=1}^{n} \textbf{ask}(c_i) \rightarrow A_i$. The hypothesis $\langle A, c\rangle \not\longrightarrow$ and Rule $R2$ in Table 1 imply that $c \in \overline{\bigcup_{i=1}^{n} \uparrow c_i}$. Then, by Definition 37, Case 3, for any n and for any $X \in \mathcal{T} \Uparrow n(A)$, $c \in X$ holds.
$A = A_1 \parallel A_2$. The hypothesis and Rule $R3$ imply both that $\langle A_1, c\rangle \not\longrightarrow$ and $\langle A_2, c\rangle \not\longrightarrow$. By inductive hypothesis, for any n, for any $Z \in \mathcal{T} \Uparrow n(A_1)$ and for any $Y \in \mathcal{T} \Uparrow n(A_2)$, $c \in Z$ and $c \in Y$. Then, by Definition 37 Case 4, for any n and for any $X \in \mathcal{T} \Uparrow n(A)$, $c \in X$ holds.
$A = \exists_x^d B$ **with** $\exists_x d \leq c$. The hypothesis and Rule $R4$ imply that $\langle A_1, d \sqcup \exists_x c\rangle \not\longrightarrow$. Then, by inductive hypothesis, for any n and for any $X \in \mathcal{T} \Uparrow n(B)$, $d \sqcup \exists_x c \in X$. Now observe that, since $\exists_x d \leq c$, from the axioms for $\exists$ follows that $\exists_x c = \exists_x c \sqcup \exists_x d = \exists_x(c \sqcup \exists_x d)$. This and Definition 37, Case 5, imply that, for any n and for any $X \in \mathcal{T} \Uparrow n(A)$, $c \in X$ holds. $\square$

Lemma 42 *Assume that A is an admissible agent. If $\langle A, c\rangle \longrightarrow \langle B, d\rangle$ and $Y \in \mathcal{T} \Uparrow n(B)$, then there exists $X \in \mathcal{T} \Uparrow n+1(A)$ such that $X \cap \uparrow c = Y \cap \uparrow d$.*

Proof. By structural induction on the agent A and by induction on n. First observe that, since $\mathcal{T}$ is monotonic, from the definition of $\mathcal{T} \Uparrow n$ it follows that

$$\mathcal{T} \Uparrow n \subseteq \mathcal{T} \Uparrow n+1 \tag{2}$$

We have then the following cases.

$A = \textbf{Stop}$. Obvious.
$A = \textbf{tell}(e)$. By Rule $R1$ of Table 1 it follows that $B = \mathit{Stop}$ and $d = c \sqcup e$. Moreover Cases 1 and 2 of Definition 37 imply that, for any n, $\mathcal{T} \Uparrow n(\textbf{Stop}) = \{\mathcal{C}\}$ and $\mathcal{T} \Uparrow n(\textbf{tell}(e)) = \{\uparrow e\}$ hold. Then it suffices to note that $\uparrow e \cap \uparrow c = \uparrow (e \sqcup c) = \uparrow d \cap \mathcal{C}$.
$A = \sum_{i=1}^{n} \textbf{ask}(c_i) \rightarrow A_i$. Rule $R2$ in Table 1 implies that there exists $j \in [1, n]$ such that $c_j \leq c$, $d = c$ and $B = A_j$. Let $Y_j \in \mathcal{T} \Uparrow n(A_j)$. From Definition

37, Case 3, it follows that

$$X = \left[(\uparrow c_j \cap Y_j) \cup \bigcap_{i=1}^{n} \overline{\uparrow c_i} \right] \in \mathcal{T} \Uparrow n(A)$$

We have then the following equalities

$$\begin{aligned} \uparrow c \cap X &= \uparrow c \cap \uparrow c_j \cap Y_j \quad (\text{since } c \in \uparrow c_j) \\ &= \uparrow d \cap Y_j \quad (\text{since } c \in \uparrow c_j \text{ and } d = c) \end{aligned}$$

and from (2) it follows that the thesis holds.

$A = A_1 \parallel A_2$. We can assume, without loss of generality, that there exists a transition $\langle A_1, c \rangle \longrightarrow \langle A_1', d' \rangle$. Then, by Rule $R3$, $B = A_1' \parallel A_2$ and $d = d'$. If $Y \in \mathcal{T} \Uparrow n(A_1' \parallel A_2)$ then, by Definition 37, Case 4, there exists $Y_1 \in \mathcal{T} \Uparrow n(A_1')$ and $Y_2 \in \mathcal{T} \Uparrow n(A_2)$ such that $Y = Y_1 \cap Y_2$. By inductive hypothesis there exists $X_1 \in \mathcal{T} \Uparrow n+1(A_1)$ such that

$$\uparrow c \cap X_1 = \uparrow d \cap Y_1. \tag{1}$$

From (2) it follows that $Y_2 \in \mathcal{T} \Uparrow n+1(A_2)$. Then we have $X = X_1 \cap Y_2 \in \mathcal{T} \Uparrow n+1(A_1 \parallel A_2)$. By definitions of X, Y and by (1) it follows that $\uparrow c \cap X = \uparrow c \cap X_1 \cap Y_2 = \uparrow d \cap Y_1 \cap Y_2 = \uparrow d \cap Y$.

$A = \exists_x^e A_1$ **with** $\exists_x e \leq c$**.** The hypothesis and Rule $R4$ imply that there exists a transition $\langle A_1, e \sqcup \exists_x c \rangle \longrightarrow \langle A_2, e' \rangle$ such that $B = \exists_x^{e'} A_2$ and $d = c \sqcup \exists_x e'$. Let $Y \in \mathcal{T} \Uparrow n(\exists_x^{e'} A_2)$. Then, by Definition 37, Case 5, there exists $Y' \in \mathcal{T} \Uparrow n(A_2)$ such that $\exists_x^{-1}(\exists_x(Y' \cap \uparrow e')) = Y$. Note that from the definitions of $\exists_x^{-1}$ and $\exists_x$ it follows that

$$\exists_x^{-1}(\exists_x(Y' \cap \uparrow e')) \subseteq \uparrow (\exists_x e'). \tag{2}$$

By inductive hypothesis, there exists $X' \in \mathcal{T} \Uparrow n+1(A_1)$ such that

$$\uparrow e \cap \uparrow \exists_x c \cap X' = \uparrow e' \cap Y'. \tag{3}$$

Consider now the set $X = \exists_x^{-1}(\exists_x(X' \cap \uparrow e))$ which, by Definition 37, Case 5, belongs to $\mathcal{T} \Uparrow n(\exists_x^e A_1)$. We have then the following equalities

$$\begin{aligned} \uparrow c \cap X &= \uparrow c \cap \exists_x^{-1}(\exists_x(X' \cap \uparrow e)) \\ &= \uparrow c \cap \exists_x^{-1}(\exists_x(X' \cap \uparrow e \cap \uparrow \exists_x c)) \\ &= \uparrow c \cap \exists_x^{-1}(\exists_x(Y' \cap \uparrow e')) \quad (\text{by } (3)) \\ &= \uparrow c \cap \uparrow (\exists_x e') \cap \exists_x^{-1}(\exists_x(Y' \cap \uparrow e')) \quad (\text{by } (2)) \\ &= \uparrow d \cap Y. \end{aligned}$$

Then the thesis follows from (2).

$A = p(X)$. Straightforward by induction on n. $\square$

Lemma 43 *Let A be an admissible agent. If $d \in \mathcal{O}(A, c)$, then there exist n and $X \in \mathcal{T} \Uparrow n(A)$ such that $d = min(X \cap \uparrow c)$.*

Proof. By induction on the length k of the computation $\langle A, c\rangle \longrightarrow^* \langle B, b\rangle \not\longrightarrow$.
($k = 0$) In this case $\langle A, c\rangle \not\longrightarrow$. Then apply Lemma 41.
($k > 0$) Assume that $\langle A, c\rangle \longrightarrow \langle A', c'\rangle$ and $d \in \mathcal{O}(A', c')$. By inductive hypothesis there exists n and $Y \in \mathcal{T} \Uparrow n(A')$ such that $d = min(Y \cap \uparrow c')$. By Lemma 42 there exists $X \in \mathcal{T} \Uparrow n(A)$ such that $X \cap \uparrow c = Y \cap \uparrow c'$ and this completes the proof. $\square$

Theorem 44 *Let A be an admissible agent. Then*

$$\mathcal{O}(A, c) \subseteq \{d \mid \textit{there exists } X \in \mathcal{D}[\![A]\!] \textit{ s.t. } d = min(\uparrow c \cap X)\}$$

Proof. Straightforward from Lemma 43 and Corollary 40. $\square$

For the purpose of defining a denotational semantics which is correct for program analysis, the result expressed by Theorem 44 is sufficient. However, if we are interested in retrieving exactly the observables, then we have a problem since the converse of the previous implication, in fact, does not always hold. The following is a counterexample.

Example 45 *Consider the following agent A*

$$(ask(c) \rightarrow B)$$
$$+$$
$$(ask(d) \rightarrow B)$$

where B is the agent

$$(ask(true) \rightarrow ask(e) \rightarrow tell(f))$$
$$+$$
$$(ask(true) \rightarrow tell(false))$$

and assume that the constraint system is $\{true, c, d, e, f, false\}$ with $e = c \sqcup d$ and $e < f$. Then we have $X = \{true, d, f, false\} \in \mathcal{D}[\![A]\!]$ and $f = min(\uparrow c \cap X)$. However, $f \notin \mathcal{O}(A, c)$.

The reason for this mismatch is that, intuitively, it is not possible to know from $\mathcal{D}$ which sets are associated to which guards. For instance, in the above example, the set $X = \{true, d, f, false\}$ is associated with the guard $ask(d)$, and when we "activate" it with the constraint c, we get the wrong result.

We conclude this section with an example which shows that confluent and admissible ccp is strictly more expressive that local-choice ccp.

Example 46 *Consider a constraint system* $\mathcal{C} = \{true, a, b, c, d, false\}$ *with* $b < c$ *and* $a \sqcup c = d$*. Consider the program*

$$\begin{aligned} A = {} & (\mathbf{ask}(a) \rightarrow \mathbf{tell}(d)) \\ & + \\ = {} & (\mathbf{ask}(b) \rightarrow \mathbf{tell}(c)) \end{aligned}$$

We have that A is admissible and structurally confluent, but it cannot be rewritten equivalently in local-choice ccp, not even if we admit in the guards disjunctions of constraints along the lines of Section 6.

5 Mutually exclusive ccp

In this section we consider a further restriction on the choice operator. Namely, we admit only *mutually exclusive* and local choices. We will show that the resulting language is decidable, and that, if we identify divergence and inconsistency, it is also a subset of both admissible ccp and structurally confluent ccp. Furthermore, for this language the semantics developed in previous section is correct in the standard sense. Our interest in this subset also comes from the fact that, as we show in the next section, the translation of ccp programs for analysis purposes gives rise to mutually exclusive programs.

Definition 47 *Define the guarded choice* $\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i$ *to be* mutually exclusive *if for all* $i, j \in \{1, .., n\}$*,* $i \neq j$ *implies* $c_i \sqcup c_j = false$ *(i.e. the guards are pairwise inconsistent). Define mutually exclusive ccp as the subset of ccp where agents and programs contain only mutually exclusive and local choices.*

Clearly this is a decidable subset of ccp, because (as a general assumption of constraint systems used in practice) it is decidable whether $c \sqcup d = false$ or not.

To identify divergence and inconsistency means to assign *false* to be the result of infinite computations, or, equivalently, to define the observables to be:

$$\mathcal{O}(A, c) = \{d \mid \text{there exists } B \text{ s.t. } \langle A, c\rangle \longrightarrow^* \langle B, d\rangle \not\longrightarrow\} \cup \{\mathit{false}\}.$$

and the results of a computation tree to be:

$$\mathit{Stores}(T) = \{d \mid \text{there exists } B \text{ s.t. } \langle B, d\rangle \text{ is a leaf of } T\} \cup \{\mathit{false}\}.$$

With these new notions, the definition of admissibility and confluency are slightly modified, and we have the following:

Proposition 48 *Mutually exclusive ccp programs and agents are both admissible and structurally confluent.*

Proof. The admissibility is straightforward. For the structural confluence, just observe that a mutually exclusive choice $(\mathbf{ask}(a) \to A) + (\mathbf{ask}(b) \to B)$ can be rewritten equivalently in $(\mathbf{ask}(a) \to A) \parallel (\mathbf{ask}(b) \to B)$, and therefore the global nondeterminism can be eliminated. $\Box$

All the results shown previously still hold, with respect to this new notion of observables, if we abstract from *false*. In addition, for mutually exclusive ccp we can show also the converse of Theorem 44.

We first need to show the following two properties on the denotations of agents, which can be easily proved by using the equality $\mathcal{D} = \bigcup_n \mathcal{T} \Uparrow n$ and induction on n.

Lemma 49 *For each agents A, for each $X \in \mathcal{D}[\![A]\!]$, for each $c \in \mathcal{C}$, either $\uparrow c \cap X$ is empty, or it has a minimum element.*

Lemma 50 *For each agents A, if $X \in \mathcal{D}[\![A]\!]$, then X is finitary, that is, there does not exists an infinite chain $c_1 < d_1 < c_2 < d_2 < c_3 < d_3 < \ldots$, such that $c_1, c_2, c_3, \ldots \in X$ and $d_1, d_2, d_3, \ldots \notin X$.*

Theorem 51 *Let D and A be a mutually exclusive ccp program and agent. Then*

$$\mathcal{O}(A, c) \supseteq \{d \mid \textit{there exists } X \in \mathcal{D}[\![A]\!] \textit{ s.t. } d = \mathit{min}(\uparrow c \cap X)\}.$$

Proof. We show by induction on n and by structural induction, that if $X \in \mathcal{T} \Uparrow n(A)$ and $d = \mathit{min}(\uparrow c \cap X)$ for a constraint c, then there exists B such that $\langle A, c\rangle \longrightarrow^* \langle B, d\rangle \not\longrightarrow$ and there exists $Y \in \mathcal{T} \Uparrow n - k(B)$ such that $\uparrow c \cap X = \uparrow d \cap Y$, where k is the number of times that Rule $R5$ (procedure call) has been applied at the top-level in the derivation $\langle A, c\rangle \longrightarrow^* \langle B, d\rangle$. The thesis then follows from Corollary 40. We have the following cases.

$A = \mathbf{Stop}$. Obvious.
$A = \mathbf{tell}(d)$. Obvious.
$A = \sum_{i=1}^n \mathbf{ask}(c_i) \rightarrow A_i$. Assume that $X \in \mathcal{T} \Uparrow n(A)$, $d = min(X \cap \uparrow c)$ and that A is a mutually exclusive choice (the case of local choice is similar). We have two possibilities.

(i) For any $i \in [1, n]$, $c_i \not\leq c$. Then the thesis holds with $B = A$.

(ii) There exists $j \in [1, n]$ such that $c_j \leq c$. From Definition 37, Case 3, and the fact that $c_j \leq c$, it follows that $\uparrow c \cap X = \uparrow c \cap Y_j$, where $Y_j \in \mathcal{T} \Uparrow n(A_j)$. From the inductive hypothesis it follows that, for some B, $\langle A_j, c\rangle \longrightarrow^* \langle B, d\rangle \not\longrightarrow$ holds, and there exists $Y \in \mathcal{T} \Uparrow n - k(B)$ such that $\uparrow c \cap Y_j = \uparrow d \cap Y$. From Rule $R2$ in Table 1 we obtain $\langle A, c\rangle \longrightarrow^* \langle B, d\rangle \not\longrightarrow$, and furthermore we observe that $\uparrow c \cap X = \uparrow c \cap Y_j = \uparrow d \cap Y$.

$A = A_1 \parallel A_2$. Let $X \in \mathcal{T} \Uparrow n(A_1 \parallel A_2)$ and assume that $d = min(X \cap \uparrow c)$. By Definition 37, Case 4, there exist $X_1 \in \mathcal{T} \Uparrow n(A_1)$ and $X_2 \in \mathcal{T} \Uparrow n(A_2)$ such that $X = X_1 \cap X_2$. Then by Lemma 49 there must be a constraint, d_1, such that $d_1 = min(X_1 \cap \uparrow c)$. By induction hypothesis there exists a computation of the form $\langle A_1, c\rangle \longrightarrow^* \langle A'_1, d_1\rangle \not\longrightarrow$ and there exists $X'_1 \in \mathcal{T} \Uparrow n - k'_1(A'_1)$ such that $X_1 \cap \uparrow c = X'_1 \cap \uparrow d_1$. Now, let $d_2 = min(X_2 \cap \uparrow d_1)$. Again by induction hypothesis, there exists a computation of the form $\langle A_2, d_1\rangle \longrightarrow^* \langle A'_2, d_2\rangle \not\longrightarrow$ and there exists $X'_2 \in \mathcal{T} \Uparrow n - k'_2(A'_2)$ such that $X_2 \cap \uparrow d_1 = X'_2 \cap \uparrow d_2$. Now we return to A'_1 and consider $d_3 = min(X'_1 \cap \uparrow d_2) = min(X_1 \cap \uparrow d_2)$. By induction we have that there exists A''_1 such that $\langle A'_1, d_1\rangle \longrightarrow^* \langle A''_1, d_3\rangle \not\longrightarrow$ and there exists $X''_1 \in \mathcal{T} \Uparrow n - k''_2(A''_1)$ such that $X'_1 \cap \uparrow d_2 = X''_1 \cap \uparrow d_3$. By proceeding in this way, we obtain a chain of constraints $d_1 < d_2 < d_3 < d_4 \ldots$ which represent the intermediate stores of a derivation from $\langle A \parallel B, c\rangle$, which satisfy $d_i \leq d$ for each i, and such that $d_1, d_3, \ldots \in X_1 \setminus X_2$, $d_2, d_4, \ldots \in X_2 \setminus X_1$. Since X_1 and X_2 are finitary, by Lemma 50 such chain must be finite, hence for some finite m we have $d_m = d_{m+1} = d$. From this we derive that, for some B_1, B_2, $\langle A_1 \parallel A_2, c\rangle \longrightarrow^* \langle B_1, \parallel B_2, d_{m+1}\rangle \not\longrightarrow$ and that for some $Y_1 \in \mathcal{T} \Uparrow n - h_1(B_1)$ and $Y_2 \in \mathcal{T} \Uparrow n - h_2(B_2)$, $X_1 \cap X_2 \cap \uparrow c = Y_1 \cap Y_2 \cap \uparrow d$ holds. Finally, observe that $\mathcal{T} \Uparrow n - h_1(B_1) \subseteq \mathcal{T} \Uparrow n(B_1)$ and $\mathcal{T} \Uparrow n - h_2(B_2) \subseteq \mathcal{T} \Uparrow n(B_2)$, therefore $Y_1 \cap Y_2 \in \mathcal{T} \Uparrow n(B_1 \parallel B_2)$, and that there cannot be procedure calls at the top-level in the computation $\langle A_1 \parallel A_2, c\rangle \longrightarrow^* \langle B_1, \parallel B_2, d_{m+1}\rangle \not\longrightarrow$.

$A = \exists^e_x B$. Let $X \in \mathcal{T} \Uparrow n(\exists^e_x B)$ and assume $d = min(X \cap \uparrow c)$, with $\exists_x e \leq c$. Then, by Definition 37, Case 5, there exists $Y \in \mathcal{T} \Uparrow n(B)$ such that $X = \exists_x^{-1} \exists_x (Y \cap \uparrow e)$. Consider now $d' = min(Y \cap \uparrow e \cap \uparrow \exists_x c)$. By inductive hypothesis there exists a computation of the form $\langle B, e \sqcup \exists_x c\rangle \longrightarrow^* \langle B', e' \sqcup \exists_x c\rangle \not\longrightarrow$, with $d' = e' \sqcup \exists_x c$, and there exists $Y' \in \mathcal{T} \Uparrow n - k(B')$

such that $Y \cap \uparrow (e \sqcup \exists_x c) = Y' \cap \uparrow (e' \sqcup \exists_x c)$. By Rule $R4$, there exists a computation of the form $\langle \exists_x^e B, c\rangle \longrightarrow^* \langle \exists_x^{e'} B', c \sqcup \exists_x e\rangle \not\longrightarrow$. Note now that

$$\begin{aligned}
d &= min(X \cap \uparrow c) \\
&= min(\exists_x^{-1}\exists_x(Y \cap \uparrow e) \cap \uparrow c) \\
&= min(\exists_x^{-1}\exists_x(Y \cap \uparrow e \cap \uparrow \exists_x c) \cap \uparrow c) \\
&= (\exists_x d') \sqcup c \\
&= (\exists_x(e' \sqcup \exists_x c)) \sqcup c \\
&= (\exists_x e') \sqcup c.
\end{aligned}$$

Finally, note that

$$\begin{aligned}
X \cap \uparrow c \cap \uparrow \exists_x e &= \exists_x^{-1}\exists_x(Y \cap \uparrow e)\ \cap c \cap \uparrow \exists_x e \\
&= \exists_x^{-1}\exists_x(Y \cap \uparrow e \cap \uparrow \exists_x c)\ \cap c \cap \uparrow \exists_x e \\
&= \exists_x^{-1}\exists_x(Y' \cap \uparrow e' \cap \uparrow \exists_x c)\ \cap c \cap \uparrow \exists_x e \\
&= \exists_x^{-1}\exists_x(Y' \cap \uparrow e')\ \cap c \cap \uparrow \exists_x e,
\end{aligned}$$

and that $X' = \exists_x^{-1}\exists_x(Y' \cap \uparrow e') \in \mathcal{T} \Uparrow n - k(\exists_x^{e'} B')$.

$A = p(X)$. Straightforward by induction on n. $\square$

We conclude this section with an example which illustrate that $\mathcal{D}$ is not fully abstract with respect to $\mathcal{O}$, that is, there are programs which are equivalent from the point of view of the observables, even when immersed in a context, but have different denotations.

Example 52 *Consider a constraint system* $\mathcal{C} = \{true, a, b, c, false\}$ *with* $true < a < b < c < false$*, and consider the agents:*

$$\begin{aligned}
A = {} & (\mathbf{ask}(true) \rightarrow \mathbf{Stop}) \\
& + \\
& (\mathbf{ask}(true) \rightarrow \mathbf{ask}(a) \rightarrow \mathbf{tell}(c))
\end{aligned}$$

$$\begin{aligned}
B = {} & (\mathbf{ask}(true) \rightarrow \mathbf{Stop}) \\
& + \\
& (\mathbf{ask}(true) \rightarrow \mathbf{ask}(a) \rightarrow \mathbf{tell}(c)) \\
& + \\
& (\mathbf{ask}(true) \rightarrow \mathbf{ask}(b) \rightarrow \mathbf{tell}(c))
\end{aligned}$$

Then we have $\{true, a, c, false\} \in \mathcal{D}[\![B]\!]$ *while* $\{true, a, c, false\} \notin \mathcal{D}[\![A]\!]$, *but* $\mathcal{O}(A, d) = \mathcal{O}(B, d)$ *for every constraint* d, *and even* $\mathcal{O}(C[A], d) = \mathcal{O}(C[B], d)$ *for every context (agent "with a hole")* $C[\,]$.

6 Application to Program Analysis

In this section we show an application of previous results to the analysis of general ccp programs. The idea is the following: we first approximate an arbitrary ccp program by a mutually exclusive one, and then analyze the latter program. The analysis can be carried out by using abstract interpretation techniques based either on the operational semantics or on a denotational one. In the first case, the advantage is that we need to consider only a single process scheduling. In the second one, a simple compositional analysis can be obtained by using the semantics presented in the previous sections as a basis. Both denotational semantics can be used, because both of them are correct for mutually exclusive programs.

Let A, D be a ccp agent and program. Define $ME(A)$, $ME(D)$ as the mutually exclusive agent and program obtained by replacing each non-mutually-exclusive choice $\sum_{i=1}^{n} \mathbf{ask}(c_i) \to A_i$ in A, D by the (local) choice

$$\mathbf{ask}(\bigvee_{1=1}^{n} c_i) \to \sum_{i=1}^{n} \mathbf{ask}(true) \to (A_i).$$

Here $\bigvee_{1=1}^{n} c_i$ represents the intuitionistic logical disjunction, namely a constraint which is entailed by a constraint d iff there exists $i \in \{1, ..., n\}$ such that d entails c_i. In order to understand better the meaning of such a disjunctive constraint, we summarize here the definitions and results of [1] (although we don't need all that machinery in this paper), which introduced a logic of constraints and formally described it in terms of a lifting of the constraint system $\mathbf{C}$ to the power set. A similar lifting was previously introduced in [2]; the difference is that in [2] a notion of (classical) negation was also present.

Definition 53 ([1]) *Given a set of constraints* $\mathcal{C}$ *with typical element* c, *a property* ϕ *is an expression described by the following grammar:* $\phi ::= c \mid \phi \wedge \psi \mid \phi \vee \psi \mid \exists_x \phi \mid D_{xy}$

Properties are interpreted as elements of $\mathcal{P}^u(\mathcal{C})$:

Definition 54 **([1])** *We define the following interpretation for properties:*

$$
\begin{aligned}
\llbracket c \rrbracket &= \uparrow c \\
\llbracket \phi \wedge \psi \rrbracket &= \llbracket \phi \rrbracket \cap \llbracket \psi \rrbracket \\
\llbracket \phi \vee \psi \rrbracket &= \llbracket \phi \rrbracket \cup \llbracket \psi \rrbracket \\
\llbracket \exists_x \phi \rrbracket &= \uparrow \exists_x \llbracket \phi \rrbracket \\
\llbracket D_{xy} \rrbracket &= \uparrow d_{xy}.
\end{aligned}
$$

It is possible to show that:

Proposition 55 **([1])** *Given a cylindric constraint system* $\mathbf{C}$*, the set of its properties, denoted by* $\mathbf{C}_\Phi$*, is a cylindric constraint system where for any* $\phi, \psi \in \mathbf{C}_\Phi$*, the entailment is defined by* $\phi \vdash \psi$ *iff* $\llbracket \phi \rrbracket \subseteq \llbracket \psi \rrbracket$ *(i.e.* $\psi \leq \phi$ *iff* $\llbracket \phi \rrbracket \subseteq \llbracket \psi \rrbracket$*).*

Note that $\wedge$ on properties corresponds to the *lub* ($\sqcup$) operator on the original constraints in $\mathbf{C}$. In fact, for $c, d \in \mathcal{C}$, $\llbracket c \sqcup d \rrbracket = \llbracket \phi \rrbracket \cap \llbracket \psi \rrbracket = \llbracket c \wedge d \rrbracket$. A similar correspondence does not hold between $\vee$ and the *glb* ($\sqcap$) of the constraint system since, in general, we only have $\llbracket c \vee d \rrbracket = \llbracket \phi \rrbracket \cup \llbracket \psi \rrbracket \subseteq \llbracket c \sqcap d \rrbracket$. Using $\sqcap$ instead of $\vee$ would give an incorrect analysis.

The operational semantics of an agent with properties in the guards is defined in the obvious way, namely

$$\langle \sum_{i=1}^{n} \mathbf{ask}(\phi_i) \rightarrow A_i, c \rangle \longrightarrow \langle A_j, c \rangle \quad \text{if } \phi_j \leq c.$$

In our case, we only consider properties of the form $\phi_i = \bigvee_{k=1}^{m} c_{ik}$, hence $\phi_j \leq c$ means that $c_{jk} \leq c$ for some k.

Observe that the result of computations for these agents are still constraints of the original constraint system $\mathbf{C}$ since the lifted (disjunctive) constraints appear only in the ask's.

The following result shows the correctness of the transformation ME.

Theorem 56 *Let* A*,* D *be a ccp agent and program. Then for any* $c \in \mathcal{C}$*,* $\mathcal{O}(A, c) \subseteq \mathcal{O}(ME(A), c)$ *(*$\mathcal{O}(ME(A), c)$ *is computed, of course, in* $ME(D)$*).*

Proof. Just note that for any agent A, B, and constraints c, d, if $\langle A, c \rangle \longrightarrow$

$\langle B, d\rangle$ then $\langle ME(A), c\rangle \longrightarrow \langle ME(B), d\rangle$, and if $\langle A, c\rangle \not\longrightarrow$ then $\langle ME(A), c\rangle \not\longrightarrow$. The rest follows by induction on the length of computations. $\square$

The converse of Theorem 56 of course does not hold. For instance, the agent $A = (\mathbf{ask}(a) \to \mathbf{Stop}) + (\mathbf{ask}(b) \to \mathbf{tell}(c))$ with input b gives only c, while the agent $ME(A) = \mathbf{ask}(a \vee b) \to (\mathbf{ask}(true) \to \mathbf{Stop}) + (\mathbf{ask}(true) \to \mathbf{tell}(c))$ can give also b.

As a consequence of Theorem 56, for any result delivered by $\langle A, c\rangle$ in D, $\langle ME(A), c\rangle$ gives the same result in some fixed computation tree for $ME(D)$. Note that, as observed in [17], the analysis based on the transformation ME can be improved by using $A_i \parallel \mathbf{tell}(c_i)$ rather than A_i in the transformed agent. In fact, in this way the information contained in c_i can improve the precision of the approximation related to the $i-th$ branch.

6.1 Compositional Analysis

The denotational semantics presented in the previous sections can be used as the basis for a compositional analysis of ccp programs. The idea is to get an abstract denotational semantics from the concrete one, following the techniques of abstract interpretation. We develop here the abstract version of the semantics $\mathcal{D}^u$ (described in Table 2); the case of the semantics $\mathcal{D}$ (described in Table 3) can be developed following the same lines.

Definition 57 *A description $\langle \mathbf{A}, \alpha, \mathbf{C}\rangle$ consists of an abstract domain (a poset) $\mathbf{A} = \langle \mathcal{A}, \leq^{\mathcal{A}}\rangle$, a concrete domain $\mathbf{C} = \langle \mathcal{C}, \leq^{\mathcal{C}}\rangle$, and a monotonic abstraction function $\alpha : \mathcal{C} \to \mathcal{A}$.*

Given $a \in \mathcal{A}$, $c \in \mathcal{C}$, we say that a approximates c, written $a \propto c$, iff $a \leq^{\mathcal{A}} \alpha(c)$. The approximation relation is lifted to functions and sets as follows:

- *Let $\langle \mathbf{A}_1, \alpha_1, \mathbf{C}_1\rangle$ and $\langle \mathbf{A}_2, \alpha_2, \mathbf{C}_2\rangle$ be descriptions, $F : \mathcal{A}_1 \to \mathcal{A}_2$ and $G : \mathcal{C}_1 \to \mathcal{C}_2$ be functions. Then $F \propto G$ iff for each $d \in \mathcal{A}_1$ and for each $e \in \mathcal{C}_1$, $d \propto_1 e$ implies $F(d) \propto_2 G(e)$.*
- *Let $\langle \mathbf{A}, \alpha, \mathbf{C}\rangle$ be a description and let $X \in \mathcal{P}(\mathcal{A})$ and $Y \in \mathcal{P}(\mathcal{C})$. Then $X \propto Y$ iff for each $e \in Y$ there exists $d \in X$ such that $d \propto e$.*

For cc languages, we are interested in descriptions of constraint systems. We give the following definition, which allows us to develop a compositional analysis based on $\mathcal{D}^u$.

Definition 58 *Consider the cylindric constraint systems $\mathbf{C}$ and $\mathbf{A}$ with $\mathbf{C} = \langle \mathcal{C}, \leq, \sqcup, true, false, Var, \exists\rangle$ and $\mathbf{A} = \langle \mathcal{A}, \leq^{\mathcal{A}}, \sqcup^{\mathcal{A}}, true^{\mathcal{A}}, false^{\mathcal{A}}, Var, \exists^{\mathcal{A}}\rangle$. A*

Table 4
The abstract denotational semantics $\mathcal{D}^{\mathcal{A}}$

AD1	$\mathcal{D}^{\mathcal{A}}[\![\mathbf{Stop}]\!] = \mathcal{A}$
AD2	$\mathcal{D}^{\mathcal{A}}[\![\mathbf{tell}(c)]\!] = \alpha(\uparrow c)$
AD3	$\mathcal{D}^{\mathcal{A}}[\![\sum_{i=1}^{n} \mathbf{ask}(\phi_i) \rightarrow A_i]\!] = \bigcup_{i=1}^{n}(\alpha(\uparrow \phi_i) \cap \mathcal{D}^{\mathcal{A}}[\![A_i]\!]) \cup \{a \mid \text{there exists } c \text{ s.t. } a \propto c \text{ and } (\bigvee_{i=1}^{n} \phi_i) \not\leq c\}$
AD4	$\mathcal{D}^{\mathcal{A}}[\![A \parallel B]\!] = \mathcal{D}^{\mathcal{A}}[\![A]\!] \cap \mathcal{D}^{\mathcal{A}}[\![B]\!]$
AD5	$\mathcal{D}^{\mathcal{A}}[\![\exists_x^c A]\!] = \{a \mid a \in \mathcal{A} \text{ and there exists } b \in \mathcal{D}^{\mathcal{A}}[\![A]\!] \cap \alpha(\uparrow c) \text{ s.t. } \exists_x^{\mathcal{A}} a = \exists_x^{\mathcal{A}} b\}$
AD6	$\mathcal{D}^{\mathcal{A}}[\![p(x)]\!] = \mathcal{D}^{\mathcal{A}}[\![\Delta_y^x A]\!]$, where $p(y)\text{:-}A$ is the declaration of $p(x)$ in D

constraint system description $\langle \mathbf{A}, \alpha, \mathbf{C}\rangle$ is a description such that

(i) $\sqcup^{\mathcal{A}} \propto \sqcup$
(ii) $\forall x \in Var.\ \exists_x^{\mathcal{A}} \propto \exists_x$.
(iii) $\forall c \in \mathcal{C}.\ \alpha(\exists_x c) = \exists_x^{\mathcal{A}} \alpha(c)$.
(iv) $\forall x, y \in Var.\ \alpha(d_{xy}) = d_{xy}^{\mathcal{A}}$.

For any constraint set X define $\alpha(X) = \{\alpha(d) \mid d \in X\}$.

Definition 59 *The semantics $\mathcal{D}^{\mathcal{A}} : Agents \rightarrow \mathcal{P}(\mathcal{A})$ is the least function which satisfies the equations in Table 4.*

The abstract semantics $\mathcal{D}^{\mathcal{A}}$ can be used to approximate the observables. Let us define the function $\mathcal{O}^{\mathcal{A}} : Agents \times \mathcal{A} \rightarrow \mathcal{P}(\mathcal{A})$ as:

$$\mathcal{O}^{\mathcal{A}}(A, a) = \uparrow (\uparrow a \ \cap \ \mathcal{D}^{\mathcal{A}}[\![A]\!])$$

Intuitively, $\mathcal{O}^{\mathcal{A}}$ represents the "abstract observables" retrieved from $\mathcal{D}^{\mathcal{A}}$. We now prove the safety of our approximation. First we need the following lemma:

Lemma 60 *Let A be an agent, and let $c \in \mathcal{D}^u[\![A]\!]$. Then $\alpha(c) \in \mathcal{D}^{\mathcal{A}}[\![A]\!]$.*

Proof. By structural induction on the agent A and by induction on the number n of iterations of the fixed point operator associated with the concrete and abstract equations. In the following, given $\phi = c_1 \vee c_2 \vee \ldots \vee c_n$, the notation $\uparrow \phi$ will stand for $\uparrow c_1 \cup \uparrow c_2 \cup \ldots \cup \uparrow c_n$. We have the following cases.

$A = \mathbf{Stop}$. Obvious, by definition of description.
$A = \mathbf{tell}(e)$. By Equation $E2$ of Table 2, $\mathcal{D}^u[\![\mathbf{tell}(e)]\!] = \uparrow e$. Let $c \in \uparrow e$. By definition of α on sets of constraints, we have $\alpha(c) \in \alpha(\uparrow e) = \mathcal{D}^{\mathcal{A}}[\![\mathbf{tell}(e)]\!]$.
$A = \sum_{i=1}^{n} \mathbf{ask}(\phi_i) \rightarrow A_i$. Let $c \in \mathcal{D}^u[\![\sum_{i=1}^{n} \mathbf{ask}(\phi_i) \rightarrow A_i]\!]$. Then, either $c \in \bigcup_{i=1}^{n}(\uparrow \phi_i \cap \mathcal{D}^u[\![A_i]\!])$ or $c \in \bigcap_{i=1}^{n} \overline{\uparrow \phi_i}$. In the first case, if $c \in \bigcup_{i=1}^{n}(\uparrow \phi_i \cap \mathcal{D}^u[\![A_i]\!])$, then there exists $1 \leq j \leq n$ such that $c \in (\uparrow \phi_j \cap \mathcal{D}^u[\![A_j]\!])$. Thus, by structural induction, $\alpha(c) \in \mathcal{D}^{\mathcal{A}}[\![A_j]\!]$ and, by definition of α over sets, and since α is monotonic and $\phi_j \leq c$, we can conclude that $\alpha(c) \in \mathcal{D}^{\mathcal{A}}[\![A_j]\!] \cap \alpha(\uparrow \phi_j)$. Let us consider the second case: if $c \in \bigcap_{i=1}^{n} \overline{\uparrow \phi_i}$ then $\bigvee_{i=1}^{n} \phi_i \not\leq c$. Since $\alpha(c) \propto c$, then $\alpha(c) \in \mathcal{D}^{\mathcal{A}}[\![\sum_{i=1}^{n} \mathbf{ask}(\phi_i) \rightarrow A_i]\!]$.
$A = A_1 \parallel A_2$. Let $c \in \mathcal{D}^u[\![A_1 \parallel A_2]\!] = \mathcal{D}^u[\![A_1]\!] \cap \mathcal{D}^u[\![A_2]\!]$. Then $c \in \mathcal{D}^u[\![A_1]\!]$ and $c \in \mathcal{D}^u[\![A_2]\!]$, and, by structural induction, $\alpha(c) \in \mathcal{D}^{\mathcal{A}}[\![A_1]\!]$ and $\alpha(c) \in \mathcal{D}^{\mathcal{A}}[\![A_2]\!]$. Hence, $\alpha(c) \in \mathcal{D}^{\mathcal{A}}[\![A_1]\!] \cap \mathcal{D}^{\mathcal{A}}[\![A_2]\!] = \mathcal{D}^{\mathcal{A}}[\![A_1 \parallel A_2]\!]$.
$A = \exists_x^c B$. Let $c' \in \mathcal{D}^u[\![\exists_x^c B]\!]$ then, by Equation $E5$, there exists $c'' \in \mathcal{D}^u[\![A]\!] \cap \uparrow c$, such that $\exists_x c' = \exists_x c''$. By structural induction, $\alpha(c'') \in \mathcal{D}^{\mathcal{A}}[\![A]\!]$. Since $c'' \in \uparrow c$ then $\alpha(c'') \in \alpha(\uparrow c)$. Thus, $\alpha(c'') \in \mathcal{D}^{\mathcal{A}}[\![A]\!] \cap \alpha(\uparrow c)$. Since $\exists_x c' = \exists_x c''$ and by definition of constraint system description, $\exists_x^{\mathcal{A}} \alpha(c'') = \alpha(\exists_x c'') = \alpha(\exists_x c') = \exists_x^{\mathcal{A}} \alpha(c')$. Thus, by Equation $AD5$, $\alpha(\exists_x c') \in \mathcal{D}^{\mathcal{A}}[\![\exists_x^c B]\!]$.
$A = p(x)$. Straightforward by induction on n. $\square$

Proposition 61 *For any structurally confluent agent A, and any constraint c, $\mathcal{O}^{\mathcal{A}}(A, \alpha(c)) \propto \mathcal{O}^u(A, c)$ holds.*

Proof. By Proposition 25, $\mathcal{O}^u(A, c) = \uparrow (\uparrow c \cap \mathcal{D}^u[\![A]\!])$. Let $d \in \uparrow (\uparrow c \cap \mathcal{D}^u[\![A]\!])$. Then there exists $d' \leq d$ such that $d' \in \uparrow c \cap \mathcal{D}^u[\![A]\!]$. Thus, $c \leq d'$. By Lemma 60, $\alpha(d') \in \mathcal{D}^{\mathcal{A}}[\![A]\!]$, and by monotonicity of α, $\alpha(c) \leq \alpha(d')$. Thus $\alpha(d') \in \mathcal{D}^{\mathcal{A}}[\![A]\!] \cap \uparrow \alpha(c)$. Again by monotonicity of α, we get $\alpha(d) \in \uparrow (\uparrow \alpha(c) \cap \mathcal{D}^u[\![A]\!])$. Finally, apply the definition $\mathcal{O}^{\mathcal{A}}(A, \alpha(c)) = \uparrow (\uparrow \alpha(c) \cap \mathcal{D}^u[\![A]\!])$. $\square$

6.2 An Example of Compositional Groundness Analysis

In this section we illustrate the use of the abstract denotational semantics $\mathcal{D}^{\mathcal{A}}$ by an example of groundness analysis for ccp programs over term equations.

Consider the terms $t_1, t_2 \ldots$ on a signature and a set Var of variables. Let $\mathcal{E}$ be the set of existentially quantified conjunctions of equations, that is, the least set $\mathcal{E}$ such that

- for any pair of terms t, t', $t = t' \in \mathcal{E}$,
- if $e \in \mathcal{E}$ then $\exists x.e \in \mathcal{E}$,
- if $e, e' \in \mathcal{E}$ then $e \sqcup e' \in \mathcal{E}$.

We denote by **Eqn** the Herbrand (cylindric) constraint system whose elements are those in $\mathcal{E}$ modulo logical equivalence with ordering $[e] \leq [e']$ iff $e' \models e$, and whose operations are the obvious ones ($\sqcup$ is logical conjunction and $\exists_x$ is the existential quantifier).

Definition 62 *An element $e \in \mathcal{E}$ is* solved *if e is of the form $\exists \vec{y}.x_1 = t_1 \sqcup \cdots \sqcup x_n = t_n$ where each x_i is a distinct variable not occurring in any of the terms t_i and each $y \in \vec{y}$ occurs in some t_j.*

It is well known that any satisfiable $e \in \mathcal{E}$ can be transformed into an equivalent one $Sol(e)$ which is solved. If e is not satisfiable we define $Sol(e) = \mathit{false}$.

The idea of groundness analysis is to infer statically which variables in the initial state are bound to ground terms in all possible successful computations.

A description of an element $e \in \mathcal{E}$ is a set of variables, with the intended meaning that any unifier of e binds these variables to ground terms. This description can be given by

$$\alpha(e) = \begin{cases} \{x \mid x = t \in Sol(e) \text{ and } t \text{ is } \mathit{ground}\} & \text{if } Sol(e) \neq \mathit{false} \\ Var & \text{if } Sol(e) = \mathit{false} \end{cases}$$

The abstract constraint system **A** has domain $\mathcal{P}(Var)$ (i.e. $\mathcal{A} = \mathcal{P}(Var)$) and operations defined as follows. For any $X, Y \in \mathcal{P}(Var)$:

(i) $X \leq^{\mathcal{A}} Y$ iff $X \subseteq Y$,
(ii) $X \sqcup^{\mathcal{A}} Y = X \cup Y$,
(iii) $\exists^{\mathcal{A}}_x X = X \setminus \{x\}$,

It is easy to verify that $\langle \mathbf{A}, \alpha, \mathbf{Eqn} \rangle$ is a constraint system description.

Consider the following declaration D defining two producers $p1$ and $p2$, which respectively produce a stream of a's and b's of arbitrary length (where a, b are constants), and an agent m (*angelic merge*, [13]), which non-deterministically merges its two input streams x and y into an output stream z.

$$
\begin{aligned}
p1(x) :- \ & (\mathbf{ask}(true) \rightarrow \exists_{x'} \ (\mathbf{tell}(x = [a|x']) \parallel p1(x'))) \\
& + \\
& (\mathbf{ask}(true) \rightarrow \ \mathbf{tell}(x = [\,]))
\end{aligned}
$$

$$
\begin{aligned}
p2(y) :- \ & (\mathbf{ask}(true) \rightarrow \exists_{y'} \ (\mathbf{tell}(y = [b|y']) \parallel p2(y'))) \\
& + \\
& (\mathbf{ask}(true) \rightarrow \ \mathbf{tell}(y = [\,]))
\end{aligned}
$$

$$
\begin{aligned}
m(x,y,z) :- \ & (\mathbf{ask}(\exists_v \exists_{x'} \ x = [v|x']) \rightarrow (\exists_v \exists_{x'} \exists_{z'} \ (\mathbf{tell}(x = [v|x']) \parallel \mathbf{tell}(z = [v|z']) \parallel m(x', y, z')))) \\
& + \\
& (\mathbf{ask}(\exists_w \exists_{y'} \ y = [w|y']) \rightarrow (\exists_w \exists_{y'} \exists_{z'} \ (\mathbf{tell}(y = [w|z']) \parallel \mathbf{tell}(z = [w|z']) \parallel m(x, y', z')))) \\
& + \\
& (\mathbf{ask}(x = [\,]) \rightarrow \ \mathbf{tell}(z = y)) \\
& + \\
& (\mathbf{ask}(y = [\,]) \rightarrow \ \mathbf{tell}(z = x))
\end{aligned}
$$

Assume that we wish to analyze the agent $p1(x) \parallel p2(y) \parallel m(x,y,z)$, which specifies that the streams produced by $p1$ and $p2$ are merged into z. Note that the agent $p1(x) \parallel p2(y) \parallel m(x,y,z)$ is not confluent.

In order to apply our techniques, since the agent m is not confluent, we have first to transform it into a mutually exclusive agent. The agents $p1$ and $p2$ have local choice only, hence we do not need to transform them. The resulting program for the agent $m' = ME(m)$ is:

$$
\begin{array}{ll}
m'(x,y,z) \text{ :- } & (\mathbf{ask}(\exists_v \exists_{x'}\ x = [v|x'] \vee \exists_w \exists_{y'}\ y = [w|y'] \vee x = [\,] \vee y = [\,]) \\
& \rightarrow \\
& (\mathbf{ask}(true) \rightarrow \exists_v \exists_{x'} \exists_{z'}\ (\ \mathbf{tell}(x = [v|x']) \\
& \qquad\qquad\qquad\qquad\quad \parallel \\
& \qquad\qquad\qquad\qquad\quad \mathbf{tell}(z = [v|z']) \\
& \qquad\qquad\qquad\qquad\quad \parallel \\
& \qquad\qquad\qquad\qquad\quad m'(x', y, z')\ ))) \\
& + \\
& (\mathbf{ask}(true) \rightarrow (\exists_w \exists_{y'} \exists_{z'}\ (\ \mathbf{tell}(y = [w|y']) \\
& \qquad\qquad\qquad\qquad\quad \parallel \\
& \qquad\qquad\qquad\qquad\quad \mathbf{tell}(z = [w|z']) \\
& \qquad\qquad\qquad\qquad\quad \parallel \\
& \qquad\qquad\qquad\qquad\quad m'(x, y', z')\ ))) \\
& + \\
& (\mathbf{ask}(true) \rightarrow \mathbf{tell}(z = y)) \\
& + \\
& (\mathbf{ask}(true) \rightarrow \mathbf{tell}(z = x))
\end{array}
$$

We can now analyze the agent $p1(x) \parallel p2(y) \parallel m'(x,y,z)$. In order to make the analysis more precise, we make the assumption that the variables occurring in the program are typed, in particular, x, y, z, x', y', and z' can only be bound to lists. This means for instance that the only equations of the form $x = \ldots$ are those of the form $x = [\,]$ or $x = [\_|x']$.

By applying Equations $AD1$–$AD6$ (or to be more precise: the bottom-up construction of the least solution of Equations $AD1$–$AD6$) we have:

$$
\begin{aligned}
\mathcal{D}^{\mathcal{A}}[\![p_1(x)]\!] &= \{\{x\},\{x,y\},\{x,z\},\{x,y,z\}\} \\
\mathcal{D}^{\mathcal{A}}[\![p_2(x)]\!] &= \{\{y\},\{x,y\},\{y,z\},\{x,y,z\}\} \\
\mathcal{D}^{\mathcal{A}}[\![m(x,y,z)]\!] &= \{\emptyset,\{x,z\},\{y,z\},\{x,y,z\}\}
\end{aligned}
$$

Hence, by using the law for parallel composition:

$$
\begin{aligned}
\mathcal{D}^{\mathcal{A}}[\![p1(x) \parallel p2(y) \parallel m'(x,y,z)]\!] = \; & \{\{x\},\{x,y\},\{x,z\},\{x,y,z\}\} \\
& \cap \\
& \{\{y\},\{x,y\},\{y,z\},\{x,y,z\}\} \\
& \cap \\
& \{\emptyset,\{x,z\},\{y,z\},\{x,y,z\}\} \\
= \; & \{\{x,y,z\}\}
\end{aligned}
$$

which shows that all finite computations of $\langle p1(x) \parallel p2(y) \parallel m'(x,y,z), true\rangle$ will bind to ground values the variables x, y, and z.

7 Conclusion

We have studied various subsets of ccp which allow a simple denotational semantics for various notions of observables. In particular, we have extended the results of [8] in two directions: First, the semantics shown in that paper has been extended to structurally confluent ccp, and proved still correct and fully abstract. Second, for the same class of programs we have developed a semantics for a more refined notion of observables (namely the input-output observables). Finally, based on these results, we have developed a framework for program analysis, by transforming those semantics in abstract semantics, following standard abstract interpretation techniques. We have shown also how to apply this framework to a generic ccp program, via a preliminary transformation into an approximating mutually-exclusive program.

References

[1] F.S. de Boer, A. Di Pierro and C. Palamidessi, An Algebraic Perspective of Constraint Logic Programming, Tech. Rep. DISI (Dipartimento di Informatica e Scienze dell'Informazione), University of Genova, 1995, *Journal of Logic and Computation*, to appear.

[2] F.S. de Boer, M. Gabbrielli, E. Marchiori and C. Palamidessi, Proving Concurrent Constraint Programs Correct, in: *Proc. 21st ACM Symp. on Principles of Programming Languages* (ACM Press, New York, 1994) 98–108.

[3] F.S. de Boer and C. Palamidessi, A Fully Abstract Model for Concurrent Constraint Programming, in: S. Abramsky and T.S.E. Maibaum, eds., *Proc.*

TAPSOFT/CAAP'91, Lecture Notes in Computer Science, Vol. 493 (Springer, Berlin, 1991) 296–319.

[4] M. Codish, M. Falaschi, K. Marriott and W. Winsborough, Efficient Analysis of Concurrent Constraint Logic Programs, in: A. Lingas, R. Karlsson and C. Carlsson, eds., *Proc. ICALP'93*, Lecture Notes in Computer Science, Vol. 700 (Springer, Berlin, 1993) 633–644.

[5] M. Falaschi, M. Gabbrielli, K. Marriott and C. Palamidessi, Compositional Analysis for Concurrent Constraint Programming, in: *Proc. 8th IEEE Symp. on Logic In Computer Science* (IEEE Computer Society Press, Los Alamitos, CA, 1993) 210–221.

[6] P. Cousot and R. Cousot, Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in: *Proc. 4th ACM Symp. on Principles of Programming Languages* (ACM Press, New York, 1977) 238–252.

[7] L. Henkin, J.D. Monk and A. Tarski, *Cylindric Algebras, Part I* (North-Holland, Amsterdam, 1971).

[8] R. Jagadeesan, V.A. Saraswat and V. Shanbhogue, Angelic non-determinism in concurrent constraint programming, Tech. Rep., Xerox Park, 1991.

[9] J. Jaffar and J.-L. Lassez. Constraint Logic Programming, in: *Proc. 14th Annual ACM Symp. on Principles of Programming Languages* (ACM Press, New York, 1987) 111–119.

[10] J. Lloyd. *Foundations of Logic Programming* (2nd edition). Springer-Verlag, 1987

[11] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, Vol. 92 (Springer, Berlin, 1980).

[12] R. Milner, *Communication and Concurrency* (Prentice Hall International, UK, 1989).

[13] P. Panangaden and V. Shanbhogue, The expressive power of indeterminate dataflow primitives, *Information and Computation* **98**(1) (1992) 99–131.

[14] V.A. Saraswat, *Concurrent Constraint Programming*, Logic Programming Series (The MIT Press, Cambridge, MA, 1993.)

[15] V.A. Saraswat and M. Rinard, Concurrent constraint programming, in: *Proc. 17th Annual ACM Symp. on Principles of Programming Languages* (ACM Press, New York, 1990) 232–245.

[16] V.A. Saraswat, M. Rinard and P. Panangaden, Semantics foundations of Concurrent Constraint Programming, in: *Proc. 18th Annual ACM Symp. on Principles of Programming Languages* (ACM Press, New York, 1991) 333–353.

[17] E. Zaffanella, G. Levi and R. Giacobazzi, Abstracting synchronisation in concurrent constraint programming, in: M. Hermenegildo and J. Penjam, eds., *Proc. 6th Int'l Symp. on Programming Languages Implementation and Logic Programming*, Lecture Notes in Computer Science, Vol. 844 (Springer, Berlin, 1994) 57–72.