Nondeterminism and infinite computations in constraint programming *

Frank S. de Boer[†], Alessandra Di Pierro[‡], and Catuscia Palamidessi[§]

Abstract

We investigate the semantics of concurrent constraint programming and of various sublanguages, with particular emphasis on nondeterminism and infinite behavior. The aim is to find out what is the minimal structure which a domain must have in order to capture these two aspects. We show that a notion of observables, obtained by the upward-closure of the results of computations, is relatively easy to model even in presence of synchronization. On the contrary modeling the exact set of results is problematic, even for the simple sublanguage of constraint logic programming. We show that most of the standard topological techniques fail in capturing this more precise notion of observables. The analysis of these failed attempts leads us to consider a categorical approach.

Contents

*This work has been supported by the HCM project EXPRESS.

†Universiteit Utrecht, Vakgroep Informatica, P.O. Box 80089, 3508 TB Utrecht, The Netherlands. E.mail: frankb@cs.ruu.nl

‡Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy. E.mail: dipierro@di.unipi.it

§DISI, Università di Genova, via Benedetto XV 3, 16132 Genova, Italy. E.mail: catuscia@disi.unige.it

1 Introduction

Constraint programming is based on a notion of a store as a set of possible values for the variables, instead of the imperative concept of store as a function from variables to values. In a sense, programming with constraints corresponds to operating directly at the level of pre and postconditions. The main characteristics which distinguish this paradigm from the imperative languages, however, is that computations evolve monotonically, in the sense that at each step the postcondition is more restricted than the precondition. In other words, the operations can only modify the store by adding further constraints. In this sense they have a pure declarative nature.

Constraint programming has been succesfully combined with another declarative paradigm, the language of logic programming ([9, 19]), thus giving raise to constraint logic programming (clp, [14, 15]). Essentially, clp extends logic programming in that it generalises the concept of a binding on a synctactical domain of data (the Herbrand universe), to an element of a domain described by a logical theory (constraint system).

An orthogonal line of research in logic programming is the incorporation of explicit concurrency mechanisms, motivated by the feeling that this language is "inherently parallel", hence suitable for parallel and distributed architectures. There have been several attempts to define mechanisms for concurrency within the logical paradigm, in order to mantain the declarative nature of the language (for a survey see [31]), but the first successful proposal was the work of Maher ([21]) in the context of clp. The intuition was to define synchronization in terms of a rule which enables a computation step when the store *entails* a given constraint.

This idea was the precursor of the *ask* primitive of concurrent constraint programming (ccp, [27, 28, 29]), which can be considered one of the most elegant proposals in this field. The computational model of ccp can be summarized as follows: All processes share a common store, which consists of all the constraints established until that moment. Parts of the store can be made local by means of a *hiding* operator, which is expressed in terms of a logical existential quantifier. Communication is achieved by the action *tell*, which adds a constraint to the store. Synchronization is achieved by an *ask* guard, which tests whether the store entails a given constraint.

Clp can be regarded as the sublanguage of ccp with no synchronization mechanism. Both of them however are *nondeterministic* languages. Another orthogonal restriction which has been investigated is the elimination of the *choice* operator, which gives rise to *deterministic ccp* ([29]). The characteristics of the latter is that all computations of an agent produce always the same results. This might be a bit surprising, because in concurrency nondeterminism is generated not only by the choice, but also by the parallel operator (different schedulings of processes can produce different results). The fact that this does not happen in ccp is due to the monotonic nature of computations.

Besides deterministic ccp one can consider other restrictions of ccp. For instance one might want to maintain the possibility of having alternative results, but in a way which is independent from the process scheduling. Such language, called *confluent ccp*, has been proposed in [11] mainly as an intermediate language for the static analysis of ccp programs by means of abstract interpretation techniques. This language is interesting in itself because it is much more expressive than deterministic ccp. In this paper we will investigate a confluent sublanguage of ccp called *restricted-choice ccp*, which results from *syntactically* restricting the choice construct to *local choice* (the same guard for all branches) and *mutually-exclusive choice* (the conditions expressed by the guards exclude each other). Actually, wrt to the particular notion of observables we consider, restricted-choice ccp boils down to *angelic ccp*, in which only local choice is allowed ([16, 20]). However our semantic constructions can be easily seen to apply to the general case of *structurally confluent ccp*, a language which is *operationally* characterized by the property that every (sub)agent is confluent. Note that clp can be considered as a proper sublanguage of angelic ccp as well, obtained by using only guards with an empty constraint.

Due to the presence of nondeterminism and synchronization, the denotational model of ccp requires relatively complex structures, like for instance *reactive sequences* ([6]) or *bounded traces operators* ([16]). These two models have been shown to be fully abstract wrt the results of finite computations (and therefore equivalent). Another approach, based on application of category theory on top of the reactive sequences, and capturing also infinite computations, has been developed in [23].

Of course, restrictions on the basic paradigm allow the construction of simpler denotational models. For instance deterministic ccp has the property that the (finite and infinite) results computed by an agent A starting from an arbitrary store can be retrieved from the *quiescent points* of A, and on the basis of this observation in [29] a very simple denotational semantics has been developed based on Scott's closure operators. The *quiescent points* of an agent actually have a compositional characterization also in the case of confluent ccp, but they only allow to retrieve the upward-closure of the results, not the results themselves. The presence of nondeterminism complicates life considerably wrt the construction of a compositional model for the exact set of results. However, if we consider finite computations only, it is possible to have a (relatively simple) model still based on closure operators. What really makes complicates matters is the attempt to capture also the results of *infinite computations*.

This problem seems to be independent from the presence of synchronization. Let us consider clp: this language also enjoys a very strong property. Namely, the results of computations can be retrieved from the results obtained by running the agent in the empty store. However, the latter set has a simple denotational description only for the finite case. There have been several attempts to model infinite computations in (constraint) logic programming. Most approaches are based on the greatest fixpoint of T_P, the immediate consequence operator which is used in logic programming for the fixpoint construction of the minimal model. Differences among these approaches depend on the kind of completion techniques applied on the underlying data structure, mainly based on partial orderings or metrics. However all these works have not been able to reach a full correspondence with the operational semantics. In the partial ordering completion of [12] only minimal answers are characterized. Furthermore the construction only works for clauses which contain at least one global variable. In the metric completion, at least in the approach found in the literature ([22, 19]) there is a basic soundness problem, because the objects which are considered are the solutions of the constraints rather than the constraints[1] themselves, and it might be the case that an infinite element is the solution of a constraint whereas its finite approximations aren't. Hence a limit element obtained in the fixpoint construction might be unobtainable operationally.

A different approach, based on adding to the program some suitable clauses containing indefinite terms, and then applying a least fixpoint construction, has been developed in [18]. However, also in this case the full correspondence is not achieved.

For the sake of completeness we cite here also [14], in which infinite computations have been studied from a declarative point of view, with the purpose of establishing a distinction between *infinite successes* and *infinite failures*. An infinite computation is "successful" whenever all partial results of the computation allow the same solution (hence the limit result has a solution). Otherwise it is considered an infinite failure. Infinite successes are shown to correspond to the difference set between the greatest and the least fixed points of T_P. The others are the difference set between $T_P \downarrow \omega$ and the greatest fixpoint of T_P. In our model the second difference set disappears, because we work on completed domains which ensure the downward continuity of T_P[2]. However also in our model a similar distinction between "successful" and "failed" infinite derivations can be made: infinite failures just correspond to the infinite computations delivering an inconsistent limit result.

In most concurrent languages, for instance the imperative languages and the languages with global nondeterminism, the denotational characterization of the operators requires complex structures, like synchronization trees or reactive sequences. On such domains there are well established techniques which allow to treat infinite computations, and they can be fruitfully applied also in the case of concurrent logic programming and concurrent constraint programming, see for instance [3], which is based on metric spaces.

In various sublanguages of ccp, however, the domain of denotations for finite computations is very abstract: sets of constraints. Such a simple domain presents in principle more difficulties for treating infinite computations, because, for instance, it does not represent the occurrence of a computation step, which is essential, in the metric approach, to get a contraction and therefore a fixpoint.

In this work we investigate what are the possibilities for treating nondeterminism and infinite computations in sublanguages of ccp while keeping the domain of denotations simple. In particular we aim at distinguishing the problems connected to the treatment of these two aspects from the problems related to synchronization, which as stated before requires anyway sequences-like structures.

---

[1] The language investigated in [22, 19] is pure logic programming, hence constraints are equalities over the Herbrand universe, and solutions are syntactical unifiers.

[2] $T_P \downarrow \omega$ is the limit of the decreasing sequence $B, T_P(B), T_P^2(B), \ldots$ where B is the Herbrand base (the top element of the domain). It can be shown that $T_P \downarrow \omega$ is the complement of the set of finite failures.

2 Constraint Systems

We adopt the approach of [28], which defines the notion of constraint system along the lines of Scott's *information systems* ([30]). Intuitively, an information system consists of a set of elements each of which represents some "consistent information", and an *entailment relation* $\vdash$ which establishes which elements can be derived from which other ones. In the view of [28], a constraint system is the same kind of structure, the only difference is the presence of an additional element representing inconsistency. The term "constraint" refers to the fact that the elements of a constraint system usually involve variables, i.e. they establish bounds to the range of values that such variables can assume.

Following [28], we regard a constraint system as a complete algebraic lattice in which the ordering $\sqsubseteq$ is the reverse of the the entailment relation ($c \sqsubseteq d$ means that d contains "more information" than c). The top element *false* represents inconsistency, the bottom element *true* is the empty constraint, and the *least upper bound* (lub) $\sqcup$ represents the join of information, i.e. the logical *and*. We refer to [28] for more details about the construction of such a structure.

Definition 2.1 A constraint system $\mathbf{C}$ is a complete algebraic lattice $(Con, Con_0, \sqsubseteq, \sqcup, true, false)$ where Con (the set of constraints) is a set ordered wrt $\sqsubseteq$, Con_0 is the set of *finite* elements, $\sqcup$ is the lub operation, and *true*, *false* are the least and greatest elements of Con, respectively. Furthermore, $\mathbf{C}$ is a Scott domain in the sense that the lub of two finite elements is also finite.

We recall that:

(i) $\mathbf{C}$ is a *complete lattice* means that every subset of Con has a least upper bound in Con;

(ii) an element $c \in Con$ is *finite* iff for any directed subset D of Con, $c \sqsubseteq \bigsqcup D$ implies $c \sqsubseteq d$ for some $d \in D$;

(iii) $\mathbf{C}$ is *algebraic* means that every element $c \in Con$ is the least upper bound of the finite elements which are smaller than c.

Example 2.2 Given an alphabet consisting of variables $x, y, z, \ldots$, function symbols $f, g, \ldots$, constant symbols (= function symbols of arity 0) $a, b, \ldots$, and the equality predicate $=$, the *Herbrand constraint system* is the partial order induced by the preorder $\langle H, \sqsubseteq \rangle$, defined as follows:

- The elements of H are sets of equations $t = u$ where t and u are terms of the alphabet,
- The relation $\sqsubseteq$ satisfies the following:
  1. $E_1 \subseteq E_2$ implies $E_1 \sqsubseteq E_2$,
  2. $\{t = t\} \sqsubseteq \emptyset$,
  3. $\{t = u\} \sqsubseteq \{u = t\}$,
  4. $\{t = u\} \sqsubseteq \{t = v, v = u\}$,
  5. if f is a function symbol of arity n, then $\{f(t_1, \ldots t_n) = f(u_1, \ldots, u_n)\} \sqsubseteq \{t_1 = u_1, \ldots t_n = u_n\}$,
  6. if f is a function symbol of arity n, then for $i \in [1, n]$, $\{t_i = u_i\} \sqsubseteq \{f(t_1, \ldots t_n) = f(u_1, \ldots, u_n)\}$,
  7. if f and g are distinct function symbols of arity m and n respectively, then, for every set E we have $E \sqsubseteq \{f(t_1, \ldots t_m) = g(u_1, \ldots, u_n)\}$,
  8. if x occurs in t and $x \not\equiv t$ then, for every E, we have $E \sqsubseteq \{x = t\}$.

Note that *true* is the equivalence class of the empty set and *false* is the equivalence class of the set of all equalities. The algebraic elements are the (equivalence classes of) finite sets.

Conditions 2, 3, 4 and 5 above correspond to the standard equality axioms (reflexivity, symmetry, transitivity and substitutivity). Conditions 6 , 7 and 8 correspond to the so-called *free-equality* axioms ([8]), which enforce the interpretation of "=" as syntactical identity, or, in other words, the interpretation of the function symbols as *data constructors*. Conditions 7 and 8 express the fact that $f(t_1, \ldots t_m) = g(u_1, \ldots, u_n)$ and $x = t$ (for x occurring in t) are equivalent to false.

The following example of Herbrand constraint system is taken from [4]. Consider an alphabet containg the variable symbols x and y, and constant symbols a and b. The Herbrand constraint system is (isomorphic to) the lattice represented in Figure 1.

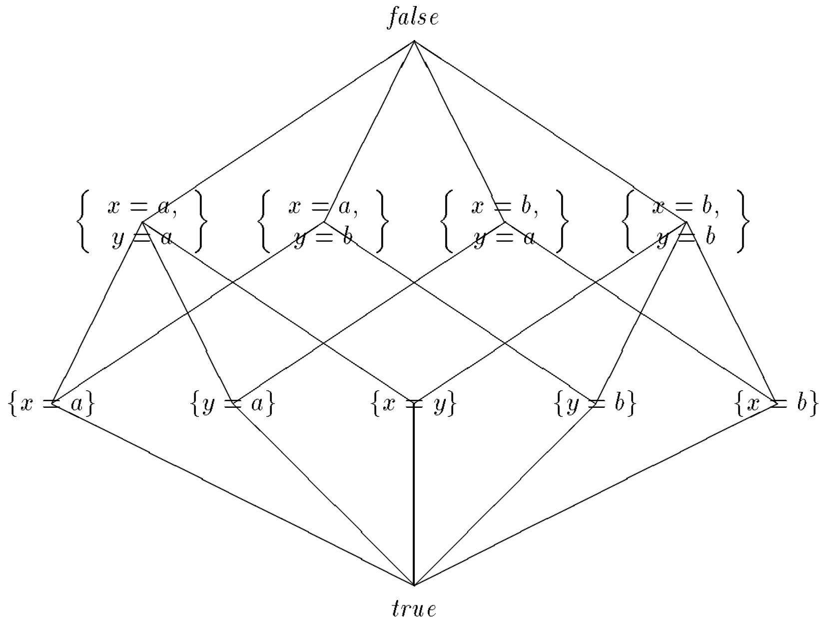

Figure 1: The Herbrand constraint lattice for x, y, a, b.

2.1 Cylindric Constraint Systems

In order to model *hiding* of local variables and *parameter passing* in constraint programming, in [28] the notion of constraint system is enriched with *cylindrification operators* and *diagonal elements*, concepts borrowed from the theory of cylindric algebras (Henkin, Monk and Tarski,[13]).

Assume given a (denumerable) set of variables Var with typical elements $x, y, z, \ldots$, and consider a family of operators $\{\exists_x \mid x \in Var\}$ (cylindrification operators) and of constants $\{d_{xy} \mid x, y \in Var\}$ (diagonal elements). Starting from a constraint system $\mathbf{C}$, define a a cylindric constraint system $\mathbf{C}'$ as the constraint system whose support set Con' is the smallest such that

$$Con' = Con \cup \{\exists_x c \mid x \in Var,\ c \in Con'\} \cup \{d_{xy} \mid x, y \in Var\}$$

modulo the identities and with the additional relations derived by the following axioms (where c, c_i, d indicate finite constraints, and $\exists_x c \sqcup d$ stands for $(\exists_x c) \sqcup d$):

A1. $\exists_x c \sqsubseteq c$,

A2. if $c \sqsubseteq d$ then $\exists_x c \sqsubseteq \exists_x d$,

A3. $\exists_x (c \sqcup \exists_x d) = \exists_x c \sqcup \exists_x d$,

A4. $\exists_x \exists_y c = \exists_y \exists_x c$,

A5. if $\{c_i\}_i$ is an increasing chain, then $\exists_x \bigsqcup_i c_i = \bigsqcup_i \exists_x c_i$,

A6. $d_{xx} = true$,

A7. if $z \neq x, y$ then $d_{xy} = \exists_z (d_{xz} \sqcup d_{zy})$,

A8. if $x \neq y$ then $c \sqsubseteq d_{xy} \sqcup \exists_x (c \sqcup d_{xy})$.

These laws give to $\exists_x$ the flavour of a first-order *existential quantifier*, as the notation suggest. The constraint d_{xy} can be interpreted as the equality between x and y. Cylindrification and diagonal elements allow us to model the variable renaming of a formula ϕ; in fact, by the above axioms, the formula $\exists_x (d_{xy} \sqcup \phi)$ can be interpreted as the formula $\phi[y/x]$, namely the formula obtained from ϕ by replacing all the free occurrences of x by y.

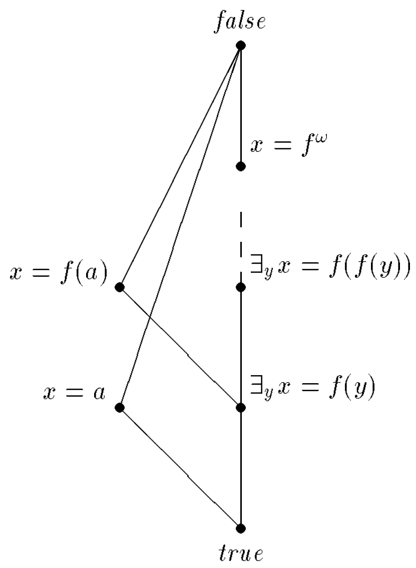

Figure 2: The cylindric Herbrand constraint system for x, a and f. The constraint $x = f^\omega$ stands for the limit of the chain $\{\exists_y x = f^i(y)\}_i$.

Example 2.3 The Herbrand constraint system in Example 2.2 can be extended to a cylindric constraint system, where $\exists_x$ just represents the standard existential quantifier. For instance, consider the alphabet which contains a constant symbol a and a monadic function symbol f. Figure 2 represents the part of the cylindric constraint system in which x (and only x) is free. For simplicity we have indicated a set $\{t = u\}$ by $t = u$.

The laws of cylindric constraint systems given above have been proposed in [28]. Axiom A5 is the only one which is not present in the standard theory of cylindric algebras; the reason to have itis to ensure that the structure of complete algebraic lattice is preserved in the construction of $\mathbf{C}'$, and this will be useful in order to deal with the results of infinite computations. All the other axioms are axioms or theorems in [13]; note however that a cylindric constraint system in general is not a cylindric algebra. In fact, a cylindric algebra is based on a Boolean algebra, whereas the structure $(Con, \sqcup, \sqcap, true, false)$, $\sqcap$ denoting the *greatest lower bound* (glb), is not necessarily a Boolean algebra. The main reasons are that $\sqcup$ and $\sqcap$ in general do not distribute, and that the complement of a constraint does not necessarily exists. As a consequence, some of the properties of cylindrification and diagonal elements which hold in [13], are not valid here. One example is the (finite and infinite) distributivity of $\exists_x$ on $\sqcap$, or, equivalently[3], the property

$$\text{if } c \sqsubseteq \exists_x d \text{ then } \exists_x c = c,$$

which expresses the idea that if c contains less information of a constraint not containing information on x, then c does not contain information on x either.

Furthermore one of the specific laws of cylindrification, $\exists_x false = false$ (Axiom C_1 in [13]), is not present here. This axiom for instance would allow us to derive $\exists_x c \sqcup \exists_y d = false$ iff $\exists_y c \sqcup \exists_x d = false$ (Theorem 1.2.15 in [13]).

One might want to enrich the theory of cylindric constraint systems with (some of) these laws, so to restrict the possible interpretations of $\exists_x$ and d_{xy}. The axioms A1-A8 above are just the properties which are necessary for the results presented in this papers[4]; any stronger theory would do as well. On the other hand, having a weaker theory allows us to establish our results for a larger class of constraint structures.

[3] It can be shown that one derives from the other, not only in the theory of cylindric algebras but also in the theory of cylindric constraint systems.

[4] Except for one result in Section 4.1 relative to mutually-exclusive choice, for which we need to assume that $\exists_x false = false$.

3 The language ccp

Concurrent constraint programming was proposed in [27] and then refined in [28, 29]. We follow here the definition of [29].

Assume given a cylindric constraint system **C** on a set of variables *Var*. Ccp agents and declarations are described by the following grammar

$$\begin{aligned} &\textit{Declarations} \quad D ::= \epsilon \mid p(x) \text{ :- } A \mid D, D \\ &\textit{Agents} \quad A ::= \mathbf{tell}(c) \mid \textstyle\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i \mid A \parallel A \mid \exists_x A \mid p(x) \end{aligned}$$

In $\mathbf{ask}(c)$ and $\mathbf{tell}(c)$ the constraint c is *finite*, i.e. it is an element of Con_0. The construct $\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i$ represents a *guarded choice*; $\parallel$ stands for *parallel composition*; $\exists_x$ is a *hiding operator*, namely it indicates that in $\exists_x A$ the variable x is *local* to A. Finally, $p(x)$ 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 the set of declarations D is fixed, and that it contains at most one declaration for each procedure name. If a procedure p is not declared we implicitly assume that D is extended so to contain a declaration $p(y)$:- $\mathbf{tell}(\mathit{false})$.

3.1 The operational model

The operational model of ccp can be described formally in the SOS style ([25]) by using a transition system $\mathcal{T} = (\mathit{Conf}, \longrightarrow)$, where *Conf* is a set of *configurations* and $\longrightarrow \subseteq \mathit{Conf} \times \mathit{Conf}$ is a transition relation. A configuration represents the state of the system at a certain moment, namely the agent which has still to be executed, and the common store. In order to describe the possible stages of the evolution of agents, we have to extend the syntax by introducing an agent **stop** which represents successful termination, and an agent $\exists_x^c A$ which represents the evolution of an agent of the form $\exists_x B$, where c is the local information which has been produced during this evolution. The agent $\exists_x A$ can then be seen as a particular case of $\exists_x^c A$: it represents the situation in which the local store is empty (as it is the case, initially, for the ccp agents). Namely, $\exists_x A = \exists_x^{true} A$.

In summary, a typical element γ of *Conf* is a pair $\langle A, c\rangle$ where c is a constraint representing the global store at that moment, and A is an extended agent ranging over the language described by the following syntax

$$\textit{Configuration-agents} \quad A ::= \mathbf{stop} \mid \mathbf{tell}(c) \mid \textstyle\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i \mid A \parallel A \mid \exists_x^c A \mid p(x) \tag{1}$$

The rules of $\mathcal{T}$ with respect to a given set of declarations D are described in Table 1.

Rule R1 expresses the behavior of an action $\mathbf{tell}(c)$, which consists of adding c to the common store.

Rule R2 describes the behavior of a guarded choice agent. A guard $\mathbf{ask}(c)$ is a test which is satisfied if the current store entails c; the agent $\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i$ selects nondeterministically one $\mathbf{ask}(c_i)$ which is *enabled* (i.e., satisfied), and then behaves like A_i. If no guards are enabled, then this agent *suspends*, waiting for other (parallel) agents to add information to the store. Note that this rule models *global non-determinism*, in fact the selection depends on the current store, which is subject to modifications by the other agents.

Rule R3 describes parallelism. A set of parallel agents evolves by *interleaving* the single executions of the agents, i.e., only one agent makes a step at each time. Note that in a configuration the store is shared among the parallel agents. The modifications produced by an agent are instantaneously visible to all parallel agents. In other words, this rule models a notion of *global* store.

Rule R4 models locality. The agent $\exists_x^d A$ behaves "almost" like A, with the difference that the variable x which is possibly present in A must be considered local, and that the information present in d has to be taken into account. It is convenient to distinguish between the *external* and the *internal* point of view. From the internal point of view, the variable x possibly occurring in the global store c is hidden. This corresponds to the usual scoping rules: the x in c is *global*, hence "covered" by the local x. Therefore, A has no access to the information on x in c, and this is formally described by filtering c with $\exists_x$. Furthermore, A can use the information (which in general concerns also the local variable x) that has been produced locally and that has been accumulated in d. In conclusion, if the store which is visible at the external level is c [5], then the

[5] Operationally $\exists_x^d A$ can only derive from a ccp agent of the form $\exists_x A'$, which has produced the local information d while evolving into $\exists_x^d A$. This local information is externally seen as $\exists_x d$, that is, $\exists_x d \sqsubseteq c$

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* $c_j \sqsubseteq d$
R3	$\dfrac{\langle A, c\rangle \longrightarrow \langle A', c'\rangle}{\begin{array}{c}\langle A \parallel B, c\rangle \longrightarrow \langle A' \parallel B, c'\rangle \\ \langle B \parallel A, c\rangle \longrightarrow \langle B \parallel A', c'\rangle\end{array}}$	
R4	$\dfrac{\langle A, \exists_x c \sqcup d\rangle \longrightarrow \langle B, \exists_x c \sqcup d'\rangle}{\langle \exists_x^d A, c\rangle \longrightarrow \langle \exists_x^{d'} B, c \sqcup \exists_x d'\rangle}$	
R5	$\langle p(x), c\rangle \longrightarrow \langle \Delta_y^x A, c\rangle$	$p(y)$:- A is the declaration for $p(x)$ in D

Table 1: The transition system $\mathcal{T}$. The notation $\Delta_y^x A$ abbreviates $\exists_\alpha^{d_{x\alpha}} \exists_y^{d_{\alpha y}} A$.

store which is visible internally by A is $(\exists_x c) \sqcup d$. Now, if A is able to make a step, thus reducing itself to B and transforming the local store into d', what we see from the external point of view is that the the agent is transformed into $\exists_x^{d'} B$, and that the information $\exists_x d$ present in the global store is transformed into $\exists_x d'$. I.e., the new information (in particular, the information concerning the local x) is accumulated in the private store of the agent, and the part of it which does not concern the local x is communicated externally.

Finally, Rule R5 models the execution of a procedure call, with a parameter passing mechanism similar to *call by reference*. The idea is to link the formal parameter y to the actual parameter x, in such a way that y inherits the constraints established on x and vice versa. Furthermore, name clashes between the formal parameter and occurrences of y in the agent must be avoided. The solution proposed in [29] is the following: if the body of $p(y)$ is A, then the call $p(x)$ is replaced by $\Delta_y^x A$, which stands for $\exists_\alpha^{d_{x\alpha}} \exists_y^{d_{\alpha y}} A$, where α is a variable which is assumed not to occur free in the declarations and in the agents. To understand this mechanism, consider first the agent $\exists_y^{d_{\alpha y}} A$: the idea is that in order to avoid clashes with external occurrences of y, the formal parameter y is made local to the body A, and its role is represented externally by α. In other words, $\exists_y^{d_{\alpha y}} A$ renames y into α (cfr. with Section 2.1: $\exists_y (d_{\alpha y} \sqcup \phi)$ stands for $\phi[\alpha/y]$). Then, $\exists_\alpha^{d_{x\alpha}}$ is applied to this agent, and this has the effect of establishing a link between the actual parameter x and α (hence y). Furthermore, it makes α local, so it can be used again for other procedure calls. I.e., for all the procedure calls one can use the same variable α.

We propose here a simpler (but equivalent) definition, which does not use intermediate variables and renames y directly into x. The idea is to define $\Delta_y^x A$ as $\exists_y^{d_{xy}} A$ in case x and y are different. When x and y coincide this definition is not correct, because $d_{yy} = \mathit{true}$, therefore $\exists_y^{d_{yy}} A = \exists_y A$, i.e. the effect would be that the actual parameter is made local to the body of the procedure. However when x and y coincide there is neither the need to establish a link, nor to hide y. Thus we define:

$$\Delta_y^x A \;=\; \begin{cases} \exists_y^{d_{xy}} A & \text{if } x \neq y \\ A & \text{if } x = y. \end{cases} \tag{2}$$

A sequence of configurations connected by the transition relation form a *computation*.

Definition 3.1 A *computation* ξ for a configuration γ_0 is a (finite or infinite) sequence $\gamma_0 \longrightarrow \gamma_1 \longrightarrow \ldots \longrightarrow$

$\gamma_n \ldots$, with $n \geq 0$.

Example 3.2 (The merge process) Consider the cylindric Herbrand constraint system on terms which denote (possibly infinite) lists. The empty list is denoted by $[\,]$, and the list whose head is a and whose tail is x is denoted by $[a \mid x]$.

The following declarations define two producers p_1 and p_2, which respectively generate a (possibly infinite) list of a's and b's, and an agent *merge*, which non-deterministically merges its two input lists x and y into an output list z.

$$\begin{array}{lll}
p_1(x) & \text{:-} & (\mathbf{ask}(true) \rightarrow \exists_{x'}\ (\mathbf{tell}(x = [a|x']) \parallel p_1(x'))) \\
 & & + \\
 & & (\mathbf{ask}(true) \rightarrow\ \mathbf{tell}(x = [\,]))
\end{array}$$

$$\begin{array}{lll}
p_2(y) & \text{:-} & (\mathbf{ask}(true) \rightarrow \exists_{y'}\ (\mathbf{tell}(y = [b|y']) \parallel p_2(y'))) \\
 & & + \\
 & & (\mathbf{ask}(true) \rightarrow\ \mathbf{tell}(y = [\,]))
\end{array}$$

$$\begin{array}{lll}
merge(x,y,z) & \text{:-} & (\mathbf{ask}(\exists_{x'}\ x = [a|x']) \rightarrow (\exists_{x'}\exists_{z'}\ (\mathbf{tell}(x = [a|x']) \parallel \mathbf{tell}(z = [a|z']) \parallel merge(x',y,z')))) \\
 & & + \\
 & & (\mathbf{ask}(\exists_{y'}\ y = [b|y']) \rightarrow (\exists_{y'}\exists_{z'}\ (\mathbf{tell}(y = [b|y']) \parallel \mathbf{tell}(z = [b|z']) \parallel merge(x,y',z')))) \\
 & & + \\
 & & (\mathbf{ask}(x = [\,]) \rightarrow\ \mathbf{tell}(z = y)) \\
 & & + \\
 & & (\mathbf{ask}(y = [\,]) \rightarrow\ \mathbf{tell}(z = x))
\end{array}$$

The agent $p_1(x) \parallel p_2(y) \parallel merge(x,y,z)$ specifies that the lists produced by $p1$ and $p2$ are merged into z.

From the operational semantics we can see that ccp has the following properties, which distinguish it from other concurrent paradigms:

Preservation of the store. During a computation the store can only increase. Formally, if $\langle A, c\rangle \longrightarrow^* \langle B, d\rangle$, then $c \sqsubseteq d$. In fact the only action which modifies the store is the tell, which can only add information.

Monotonicity with respect to the store. The intermediate stores of a computation depend monotonically on the initial store. Formally, if $\langle A, c\rangle \longrightarrow^* \langle B, d\rangle$ and $c \sqsubseteq c'$, then there exists d' such that $d \sqsubseteq d'$ and $\langle A, c'\rangle \longrightarrow^* \langle B, d'\rangle$. In fact the basic action $\mathbf{tell}(e)$ is monotonic because of the monotonicity of $\sqcup$. Moreover the test $\mathbf{ask}(e)$ has a monotonic behavior because if it is enabled in a store c then it will be enabled also in any store c' with $c \sqsubseteq c'$.

Restartability. At each point of the computation a ccp agent can restart from the beginning, with the new store, and redo the same computation steps until it reaches the same configuration. Formally, if $\langle A, c\rangle \longrightarrow^* \langle B, d\rangle$, then $\langle A, d\rangle \longrightarrow^* \langle B, d\rangle$. The key points are (a) choices are guarded by monotonic conditions, and the store during the computation can only increase, and (b) for $c \sqsubseteq d$, $c \sqcup d = d$ holds.

These properties are essential for the results presented in this paper, and will be used in the technical parts without explicit mentioning.

3.2 Fair computations

In a distributed model, in general, parallel agents are executed by different processors. As a consequence, we can reasonably assume that an agent which is enabled will be eventually executed. Formally this assumption is described in terms of certain *fairness requirements* on the computations.

In order to define fair computations, we introduce the notions of *enabled* and *active* agents, following [11]. Intuitively, the active agent is the unique "primitive" sub-agent actually reduced by a particular transition. Note that a transition is derived from the rules via a proof tree which has at the top one and only one instance of one of the axioms R1, R2, or R3. The agent of the initial configuration of this instance is the one which actually activates the transition, and which therefore we call the active agent. Formally:

Definition 3.3 Let $\gamma = \langle A, c\rangle$ and let $\tau = \gamma \longrightarrow \gamma'$ be a transition in $\mathcal{T}$. The *active agent* in τ is A if τ is an instance of Rule R1, R2, or R5. In case τ is the consequent of an instance of Rule R3 or R4, then the active agent in τ is the active agent of the antecedent transition in the rule.

Definition 3.4 An agent A is *enabled* in a configuration γ if there is a transition from γ with A as the active agent.

We are now ready to give the definition of a fair computation.

Definition 3.5 A computation is *fair* if each agent enabled at some point is eventually active[6].

Note that a finite fair computation is guaranteed to be *maximal*, namely no outgoing transitions are possible from its last configuration. Note also that this notion of fairness refers to parallelism, not to choices.

Example 3.6 In Example 3.2 every computation starting from the configuration $\langle p_1(x), \mathit{true}\rangle$ (as well as from $\langle p_2(x), \mathit{true}\rangle$) is fair. If the second branch in the declaration is never selected then the computation, and the list produced, will be infinite; finite otherwise.

The fairness requirement ensures that in the agent $p_1(x) \parallel p_2(y) \parallel \mathit{merge}(x, y, z)$, if the lists produced in x and y are both infinite then z will be infinite as well. In case the list produced in x is finite, then z will contain all the items which are produced in y, and vice versa. In other words, the merge process described by the example is *angelic*.

We will indicate by $\mathit{Comp}(\gamma)$ the set of fair computations starting with γ. The infinite fair computations, and the finite fair ones will be denoted by $\mathit{Comp}_{\mathrm{inf}}(\gamma)$ and $\mathit{Comp}_{\mathrm{fin}}(\gamma)$, respectively.

3.3 The observables

The standard notion of observables for ccp are the *results* computed by an agent for a given initial store. The result of a (fair) computation is defined as the least upper bound of all the stores occurring in the computation, which, due to the monotonic properties of ccp, form a chain. More formally, given a finite or infinite computation ξ of the form

$$\langle B_0, c_0\rangle \longrightarrow \langle B_1, c_1\rangle \longrightarrow \ldots \longrightarrow \langle B_i, c_i\rangle \longrightarrow \ldots,$$

the result of ξ, denoted by $\mathit{Result}(\xi)$, is the constraint $\bigsqcup_i c_i$. Note that for a finite computation the result coincides with the store of the last configuration.

In the following, given a set X, $\mathcal{P}(X)$ denotes the set of all the subsets of X.

Definition 3.7 (The observables) The mapping $\mathcal{O} : \mathit{Agents} \to \mathit{Con} \to \mathcal{P}(\mathit{Con})$, which gives the observables of an agent, is defined by $\mathcal{O}(A) = \mathcal{O}_{\mathrm{fin}}(A) \cup \mathcal{O}_{\mathrm{inf}}(A)$, with

$$\begin{aligned} \mathcal{O}_{\mathrm{fin}}(A)c &= \{\mathit{Result}(\xi) \mid \xi \in \mathit{Comp}_{\mathrm{fin}}(A, c)\}, \\ \mathcal{O}_{\mathrm{inf}}(A)c &= \{\mathit{Result}(\xi) \mid \xi \in \mathit{Comp}_{\mathrm{inf}}(A, c)\}, \end{aligned}$$

where $\mathit{Comp}(A, c)$ abbreviates $\mathit{Comp}(\langle A, c\rangle)$.

[6] Since the store evolves monotonically, and the ask is a monotonic operation, once an agent is enabled it is enabled forever. Therefore in ccp *strong fairness* and *weak fairness* coincide.

Example 3.8 Consider again Example 3.2. Let A denote $p_1(x) \parallel p_2(y) \parallel \mathit{merge}(x,y,z)$. A list of $n \leq \omega$ occurrences of an element a we represent by a^n. We have

$$\mathcal{O}_{\text{fin}}(A)\mathit{true} \;=\; \{\{x = a^m, y = b^n, z = w\} \mid w \text{ contains } n\ a\text{'s and } m\ b\text{'s }\}$$

$$\begin{aligned}
\mathcal{O}_{\text{inf}}(A)c \;=\; & \{\{x = a^m, y = b^\omega, z = w\} \mid w \text{ contains infinitely many } b\text{'s and at most } m\ a\text{'s }\} \\
\cup\; & \{\{x = a^\omega, y = b^n, z = w\} \mid w \text{ contains infinitely many } a\text{'s and at most } n\ b\text{'s }\} \\
\cup\; & \{\{x = a^\omega, y = b^\omega, z = w\} \mid w \text{ contains only } a\text{'s and } b\text{'s and it is infinite }\}.
\end{aligned}$$

4 Restricted-choice ccp and upward-closed observables

To investigate the problem of the denotational characterization of infinite computations, we start with a weaker notion of observables: the upward closure of $\mathcal{O}$. The significance of this notion relies on the "declarative flavour" of ccp: In the so-called *declarative interpretation*, a set of ccp declarations is regarded as a logical theory (see [5]): the parallel operator is interpreted as conjunction, the choice as disjunction, the hiding operator as existential quantifier, **tell**(c) as c, $\mathit{ask}(c) \rightarrow A$ as a form of implication, and $p(x)$:- B as the logical equivalence of $p(x)$ and B. A computation, for a given agent A and an initial store c, corresponds to an attempt of proving that $A \wedge c$ is a logical consequence of this theory. The final configuration of a computation represents the additional "hypothesis" which must be assumed in order to prove $A \wedge c$. So, the logical interpretation of a computation ξ starting from $\langle A, c\rangle$ and ending in $\langle B, d\rangle$ is:

$$D \wedge B \wedge d \models A \wedge c.$$

In particular, for $B = \mathbf{stop}$, we get $D \wedge d \models A \wedge c$.

The vice versa in general does not hold: if we have

$$D \wedge d \models A \wedge c, \tag{3}$$

then it is not guaranteed that d is the result of a computation ξ for $\langle A, c\rangle$. One reason is that d could be stronger than the hypothesis necessary for the proof. Hence, the most we can hope for is to find a computation ξ such that

$$\mathit{Result}(\xi) \sqsubseteq d.$$

i.e. such that d is in the upward-closure of $\mathit{Result}(\xi)$. The notion of upward-closed observables was introduced in logic programming and in constraint logic programming as the operational counterpart of the set of d's which satisfy (3). It could be shown that also for restricted-choice ccp (see Definition 4.2) we have such characterization, modulo some adjustment for the interpretation of the choice.

Given a set $C \subseteq \mathit{Con}$, we denote by $\Uparrow C$ (upward closure of C) the set $\{d \in \mathit{Con} \mid \text{there exists } c \in C \text{ s.t. } c \sqsubseteq d\}$. A set C is *upward-closed* if $C = \Uparrow C$. The set of the upward-closed subsets of Con will be denoted by $\mathcal{P}^{\text{uc}}(\mathit{Con})$.

Definition 4.1 (Upward-closed observables) The mapping $\mathcal{O}^{\text{uc}} : \mathit{Agents} \rightarrow \mathit{Con} \rightarrow \mathcal{P}^{\text{uc}}(\mathit{Con})$, which gives the upward-closed observables of A, in the initial store c, is defined as

$$\mathcal{O}^{\text{uc}}(A)c = \Uparrow \mathcal{O}(A)c.$$

Analogously, we define $\mathcal{O}^{\text{uc}}_{\text{inf}}(A)c = \Uparrow \mathcal{O}_{\text{inf}}(A)c$. For the finite observables we add also the value *false*, thus we define $\mathcal{O}^{\text{uc}}_{\text{fin}}(A)c = \Uparrow \mathcal{O}_{\text{fin}}(A)c \cup \{\mathit{false}\}$.

The reason why we add the store *false*, in the finite case, is a bit technical and will be clarified later. Note that $\mathcal{O}^{\mathrm{uc}}_{\mathrm{fin}}(A)c \neq \Uparrow \mathcal{O}_{\mathrm{fin}}(A)c$ only in case $\mathcal{O}_{\mathrm{fin}}(A)c = \emptyset$. Thus this definition corresponds to say that a non-terminating computation is treated like a computation which brings to an inconsistent situation.

In this section we show that, for a certain subset of ccp, this notion of observables has a very simple denotational model, based on Scott's closure operators ([30]). Afterwards we will discuss what are the difficulties in extending this model to the full ccp language.

The subset of ccp we consider is obtained by restricting the guarded choice construct. Namely, we admit only choices which are guarded either by the same condition (*local choice*) or by conditions which are mutually exclusive (*mutually exclusive choice*), in the sense that the constraints which are asked are pairwise incompatible. Agents which satisfy this requirement are *confluent*, i.e. different schedulings on the execution order of parallel agents will bring to the same set of results ([11]).

Definition 4.2 The *restricted-choice ccp* is the sub-language of ccp in which every guarded choice construct $\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i$, both in the agents and in the body of declarations, satisfies either

(i) $\forall i, j \in [1, n].\, c_i = c_j$ (local choice), or

(ii) $\forall i, j \in [1, n].\, i \neq j \Rightarrow c_i \sqcup c_j = \mathit{false}$ (mutually exclusive choice).

The denotational model we are going to describe is inspired by the semantical framework introduced in [29] for deterministic ccp, the sublanguage of ccp with guarded statements but no choice construct. This framework has been successively extended in [16] and [20] so to treat *angelic ccp*, where local choice is allowed[7]. In these two papers, however, there is an error related to the choice of the semantic domain: Their domain is not closed wrt set-intersection, hence the denotational interpretation of the parallel operator, which is modelled as set-intersection, is not well defined. We come back to this point in Example 4.12.

We use a different domain from the one choosen in [16, 20], which does not present the above described problem. Actually our construction would be still valid for *structurally confluent ccp* ([11]), a language which is operationally characterized by the property that every subagent is confluent, and which is therefore a super-language of restricted-choice ccp. We have chosen for the latter, in this paper, because it has a clear syntactical characterization.

Note that our language is, from a syntactical point of view, a super-language of angelic ccp, in that it includes also the mutually-exclusive choice. From a semantical point of view, hovever, restricted-choice ccp and angelic ccp are equivalent. In fact, a mutually-exclusive choice of the form $(\mathbf{ask}(c_1) \rightarrow A_1) + \ldots + (\mathbf{ask}(c_n) \rightarrow A_n)$ could be equivalently replaced by $(\mathbf{ask}(c_1) \rightarrow A_1) \parallel \ldots \parallel (\mathbf{ask}(c_n) \rightarrow A_n)$.

To get the drift on the construction of the denotational semantics, consider a ccp agent A as a function f_A which transforms sets of initial stores into (the upward closure of) sets of final stores, corresponding to maximal computations. It can be shown that, for $C, C', C_i \in \mathcal{P}^{\mathrm{uc}}(\mathit{Con})$, f_A satisfies the following properties.

Extensivity. $C \supseteq f_A(C)$. Due to the preservation of the store during a computation (see Section 3.1).

Monotonicity. $C \supseteq C' \Rightarrow f_A(C) \supseteq f_A(C')$. Due to the the pointwise definition of f_A.

Idempotency. $f_A(f_A(C)) = f_A(C)$. Due to restartability (see Section 3.1).

Linearity. $f_A(\bigcup_i C_i) = \bigcup_i f_A(C_i)$. Again due to the pointwise definition of f_A.

Actually monotonicity is here a consequence of linearity, but we have stated it explicitly for the sake of the presentation. Extensivity, monotonicity and idempotency characterize f_A as a closure operator ([30]) on the complete lattice $(\mathcal{P}(\mathit{Con}), \supseteq)$. The nice feature of such an operator is that it can be expressed in terms of the set of its fixponts $\tilde{f}_A = \{C \mid f_A(C) = C\}$; in fact, from these three properties it follows that

$$f_A(C) = \bigcup \{C' \mid C \supseteq C' \text{ and } C' \in \tilde{f}_A\}.$$

[7]In [16, 20] angelic ccp is not presented as local-choice ccp, since the operational semantics describes global choice instead. But, as the authors themselves point out, their operational semantics does not reflect the intended model, and from the denotational semantics and the correspondence with the observables it appears clear that the intended language amounts exactly to local-choice ccp. Note that the observables in [16, 20] are defined differently, but it is possible to prove that they are equivalent to our notion $\mathcal{O}^{\mathrm{uc}}$.

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

Table 2: Equations for the denotational semantics for restricted-choice ccp.

Furthermore, linearity allows us to take into account only the *singleton* fixpoints, namely those sets C of the form $\Uparrow c$, for $c \in \mathit{Con}$. Note now that $\Uparrow c$ is a fixpoint of f_A iff there is at least a computation $\xi \in \mathit{Comp}(A, c)$ such that $c = \mathit{Result}(\xi)$. A constraint c which satisfies this property is a *quiescent point* of A, in the sense that the activity going on in ξ will not affect the store, thus it will not be sensed by any external observer (which can only look at the evolution of the store).

The basic idea of the denotational semantics is to associate to every agent the set of its quiescent points. In general (for full ccp) it is not possible to describe compositionally the quiescent points, but we will show that it is possible for restricted-choice ccp. Let $[\![\cdot]\!] : \mathit{Agents} \rightarrow \mathcal{P}(\mathit{Con})$ denote such an association. Then $[\![\cdot]\!]$ has to satisfy the equations illustrated in Table 2, where, for $C \subseteq \mathit{Con}$, $\overline{C}$ indicates the set $\mathit{Con} \setminus C$ (the complement of C), and $\exists\!\!\!\exists_x C$ indicates the set $\{c \in \mathit{Con} \mid \text{there exists } d \in C \text{ s.t. } \exists_x c = \exists_x d\}$. This set can be also expressed as $\exists_x^{-1} \exists_x C$, where $\exists_x C = \{\exists_x c \mid c \in C\}$ and $\exists_x^{-1} D = \{c \mid \text{there exists } d \in D \text{ s.t. } \exists_x c = d\}$.

The formal justification of Equations E1-E6 is implicit in the proof of correspondence with the observables (see Sections 4.1 and 4.2). However, it should not be difficult to grasp the intuition behind them. Every constraint is a quiescent point for **stop**. The agent $\mathbf{tell}(c)$ always adds c to the stores, therefore d is a quiescent point for $\mathbf{tell}(c)$ iff $c \sqsubseteq d$. Concerning the agent $[\![\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i]\!]$, its quiescent points are all those stores d for which the agent suspends immediately, i.e. $d \in \bigcap_{i=1}^{n} \overline{\Uparrow c_i}$, plus those stores d which activate a branch $i \in [1, n]$, i.e. $c_i \sqsubseteq d$, and which are quiescent points for A_i, i.e. $d \in [\![A_i]\!]$. For the parallel operator we have that d is a quiescent point of $A \parallel B$ iff it is a quiescent point of both A and B. The last equation, which concerns the procedure call, is obvious.

The most difficult case is Equation E5. If c is a quiescent point of $\exists_x A$, then $\langle A, \exists_x c\rangle$ has a computation which brings to a quiescent point d that is, from the external point of view, the same as c: $\exists_x c = \exists_x d$. Viceversa, assume that d is a resting point of A via a computation ξ, and that $\exists_x d = \exists_x c$. Since choices are either local or mutually exclusive, there is computation ξ' starting with $\langle A, \exists_x d\rangle$ which makes, as far as possible, the same choices of ξ, and rests otherwise. Therefore ξ' brings to a result c' for which $\exists_x d \sqcup c' \sqsubseteq d$. It follows that c is a fixpoint for $\exists_x A$. Note that this is the only case in which we use the fact that A is a restricted-choice ccp agent. Example 4.36 in Section 4.3 shows that this hypothesis is really necessary. All the other equations hold also for the quiescent points of full ccp.

It is interesting to note that the equation of the choice construct can be derived from the semantic equations of angelic ccp as described in [16] by transforming the mutually-exclusive choices into parallel processes as described before.

The equations in Table 2 fix the interpretation of all the operators which can occur in an agent, but do not identify an unique $[\![\cdot]\!]$. This is a well known phenomenon due to recursion. For instance, if we have a declaration of the form

$$p(x) \text{ :- } p(x),$$

then for every set C the interpretation $[\![p(x)]\!] = C$ satisfies the equations. Which interpretation "reflects better" the operational semantics depends on what observables we want to model. For the finite observables, the denotation of $p(x)$ should be $\{false\}$, because there are no finite computations starting from $p(x)$. For infinite computations, the denotation should be $Con (= \Uparrow true)$ because every store is a quiescent point. This inspires the following definition:

Definition 4.3 Consider the set $\mathcal{P}_{\mathrm{wf}}(Con)$ of the subsets of Con which contain *false*, with the ordering $\subseteq$, and consider the natural extension of this ordering on $Agents \rightarrow \mathcal{P}_{\mathrm{wf}}(Con)$. Define $\mathcal{D}_{\mathrm{fin}}, \mathcal{D} : Agents \rightarrow \mathcal{P}_{\mathrm{wf}}(Con)$ as the least and the greatest functions, respectively, which satisfy the equations in Table 2.

In order to prove that this is a good definition, i.e. that the least and the greatest functions satisfying the equations actually exist, we associate with the set of declarations D a mapping on interpretations which is monotonic, and whose fixpoints are all the interpretations which satisfy the equations. The formal justification of Definition 4.3 follows then by the theorem of Knaster-Tarski. Let us first fix the notion of interpretation.

Definition 4.4 An interpretation is any function $I : Agents \rightarrow \mathcal{P}_{\mathrm{wf}}(Con)$.

We denote by $\mathcal{I}$ the set of all interpretations, and (with slight abuse of notation) by $\subseteq$ the ordering on $\mathcal{I}$, and by $\cup$ and $\cap$ the lub and the glb respectively.

Definition 4.5 The mapping $F : \mathcal{I} \rightarrow \mathcal{I}$, associated to the set of declarations D, is defined as follows:

1. $F(I)(\mathbf{stop}) = Con$,
2. $F(I)(\mathbf{tell}(c)) = \Uparrow c$,
3. $F(I)(\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i) = \bigcap_{i=1}^{n} \overline{\Uparrow c_i} \ \cup \ \bigcup_{i=1}^{n} (\Uparrow c_i \cap F(I)(A_i))$,
4. $F(I)(A \parallel B) = F(I)(A) \bigcap F(I)(B)$,
5. $F(I)(\exists_x A) = \exists\!\!\exists_x F(I)(A)$,
6. $F(I)(p(x)) = I(\Delta^x_y A)$, where $p(y)$:- A is the declaration of p in D.

We have the following characterization of the solutions of Equations E1-E6.

Proposition 4.6 *Let I be an interpretation. Then I is a solution of Equations E1-E6 iff I is a fixpoint of F.*

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

$$\begin{array}{ll} & I(A \parallel B) = I(A) \bigcap I(B) \\ \Leftrightarrow & \{\text{structural induction}\} \\ & I(A \parallel B) = F(I)(A) \bigcap F(I)(B) \\ \Leftrightarrow & \{\text{Definition 4.5 }\} \\ & I(A \parallel B) = F(I)(A \parallel B). \end{array}$$

□

It's easy to see that F is monotonic, hence by the theorem of Knaster-Tarski, we can rephrase Definition 4.3 as follows.

Remark 4.7 *Let $[\mathcal{I} \to \mathcal{I}]$ denote the set of monotonic functions from $\mathcal{I}$ to $\mathcal{I}$, and $lfp, gfp : [\mathcal{I} \to \mathcal{I}] \to \mathcal{I}$ denote the least fixpoint and the greatest fixpoint operators, respectively. Then*

(i) $\mathcal{D}_{\text{fin}} = lfp(F)$,

(ii) $\mathcal{D} = gfp(F)$.

In the next sections we show the correspondence between $\mathcal{D}_{\text{fin}}$ and $\mathcal{O}_{\text{fin}}$, and between $\mathcal{D}$ and $\mathcal{O}$. It will be useful to introduce the following notations.

Definition 4.8 Let $I_\perp$ be the least interpretation, i.e. the interpretation which maps every agent into $\{false\}$. Define the *bottom-up iterations* of F as:

(i) $F \uparrow 0 = I_\perp$,

(ii) $F \uparrow n+1 = F(F \uparrow n)$,

(iii) $F \uparrow \omega = \bigcup_n F \uparrow n$.

Definition 4.9 Let $I_\top$ be the greatest interpretation, i.e. the interpretation which maps every agent into Con. Define the *top-down iterations* of F as:

(i) $F \downarrow 0 = I_\top$,

(ii) $F \downarrow n+1 = F(F \downarrow n)$,

(iii) $F \downarrow \omega = \bigcap_n F \downarrow n$.

From the monotonicity of F, it follows that

Remark 4.10 $F \uparrow \omega \subseteq lfp(F)$ *and* $gfp(F) \subseteq F \downarrow \omega$.

Usually the correspondence of $lfp(F)$ and $gfp(F)$ with the operational semantics is shown via their equality with $F \uparrow \omega$ and $F \downarrow \omega$ respectively. These equalities usually are proved by exploiting the (upward and downward) continuity of an operator F. In our case, it is possible to prove that F is upward continuous, i.e. that for every increasing chain $\{I_i\}_i$ in $\mathcal{I}$, $F(\bigcup_i I_i) = \bigcup_i F(I_i)$ holds. Unfortunately, the other property does not hold: F is not downward continuous.

Example 4.11 Consider a Herbrand constraint system, where we assume that $\exists_x false = false$, and consider the declaration

$$p(x) \ \text{:-}\ \exists_y(\mathbf{tell}(x = f(y)) \parallel p(y)).$$

Let $\{I_i\}_i$ be the the decreasing chain of interpretations defined as $I_i(p(x)) = \{x = f^k(a) \mid k \geq i\} \cup \{false\}$ for every variable x. We show that $F(\bigcap_i I_i)(\exists_x p(x)) \neq \bigcap_i F(I_i)(\exists_x p(x))$. Observe that $(\bigcap_i I_i)(p(x)) = \{false\}$, hence $F(\bigcap_i I_i)(p(x)) = \{false\}$. Therefore we have $F(\bigcap_i I_i)(\exists_x p(x)) = \mathbb{E}_x F(\bigcap_i I_i)(p(x)) = \{false\}$. On the other hand $true \in \mathbb{E}_x F(I_i)(p(x)) = F(I_i)(\exists_x p(x))$ for every i, therefore $true \in \bigcap_i F(I_i)(\exists_x p(x))$.

To overcome this problem, [16] suggests to restrict to interpretations which map agents into Scott-compact sets. A set $C \subseteq Con$ is *Scott-compact* iff for every set $D \subseteq Con_0$, which is a covering for C, i.e. $C \subseteq \Uparrow D$, there exists a finite subset D' of D such that $C \subseteq \Uparrow D'$. Note that this condition would rule out the interpretations of previous example.

Unfortunately if we do not add some other property, like for instance upward-closure, Scott-compactness in itself is not preserved by set intersection, as the following example shows. Therefore the operator F would be not well defined on this class of interpretations (it uses set-intersection for the parallel operator). Note that interpretations are in general non upward-closed because of the guarded statement.

Example 4.12 Let $Con = \{true, a, x = 0, x = 1, \ldots, x = n, \ldots, false\}$ with the ordering $true \sqsubseteq a \sqsubseteq x = 0, x = 1, \ldots, x = n, \ldots, \sqsubseteq false$. Consider the sets $C = \{true, x = 0, x = 1, \ldots, x = n, \ldots, false\}$ and $D = \{a, x = 0, x = 1, \ldots, x = n, \ldots, false\}$. Then both C and D are Scott-compact, but their intersection $C \cap D = \{x = 0, x = 1, \ldots, x = n, \ldots, false\}$ is not.

In order to prove the correspondence between the denotational and the operational semantics we follow therefore a different strategy, which essentially consists of exploiting directly the finitely-branching property of the operational semantics. As a consequence of this correspondence it will turn out that the equalities $\mathit{lfp}(F) = F\uparrow\omega$ and $\mathit{gfp}(F) = F\downarrow\omega$ actually hold.

Next section is about the correspondence between $\mathcal{D}_{\text{fin}}$ and the finite observables. It has been included for completeness and because it shows a nice symmetry with the finite + infinite case.

4.1 Correspondence between the observables and the denotational semantics: finite case

To show the correspondence between $\mathcal{D}_{\text{fin}}$ and the finite observables, it is convenient to introduce the notion of *finite quiescent points*, or *resting points*, of an agent A. Those are the stores on which the activity of A arrests, either because it suspends (deadlocks) or because it terminates, i.e. it has no more actions to execute. Moreover, we add *false* as a special case because *false* is by default in the finite observables. We will show that the *resting points* actually coincide with $\mathcal{D}_{\text{fin}}$, and that from them we can retrieve the finite observables.

Definition 4.13 The *finite quiescent points* (also called *resting points*) of an agent A are the set

$$\mathit{Quie}_{\text{fin}}(A) = \{c \in \mathit{Con} \mid c \in \mathcal{O}_{\text{fin}}(A)c\} \cup \{\mathit{false}\}.$$

The function $\mathit{Quie}_{\text{fin}}$ has type $\mathit{Agents} \to \mathcal{P}_{\text{wf}}(\mathit{Con})$, hence it is an interpretation. We will show that $\mathit{Quie}_{\text{fin}}$ actually coincides with $\mathit{lfp}(F)$.

Next lemma states a property of restricted-choice ccp agents which is fundamental both for the equality between $\mathit{Quie}_{\text{fin}}$ and $\mathit{lfp}(F)$ and for the correspondence between $\mathit{Quie}_{\text{fin}}$ and the observables. Essentially it says that, in restricted-choice ccp, by starting with an input d it is always possible to approximate the results of computations which start with inputs bigger than 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 4.15).

Lemma 4.14 *Let A be a restricted-choice ccp agent, let $c \in \mathit{Con}$, $\xi \in \mathit{Comp}_{\text{fin}}(A, c)$, and assume $\mathit{Result}(\xi) \neq \mathit{false}$. Consider $d \in \mathit{Con}$ such that $d \sqsubseteq c$. Then there exists $\xi' \in \mathit{Comp}_{\text{fin}}(A, d)$ such that $\mathit{Result}(\xi') \sqsubseteq \mathit{Result}(\xi)$.*

Proof Basically the lemma follows from the following simple observation. Let $c \neq \mathit{false}$, $d \sqsubseteq c$, and $\langle \Sigma_{i=1}^{n} \mathbf{ask}(c_i) \to A_i, c\rangle \to \langle A_i, c\rangle$. Then, either there exists a transition $\langle \Sigma_{i=1}^{n} \mathbf{ask}(c_i) \to A_i, d\rangle \to \langle A_i, d\rangle$, or $\langle \Sigma_{i=1}^{n} \mathbf{ask}(c_i) \to A_i, d\rangle \not\to$, because if $c_i \not\sqsubseteq d$ then $c_j \not\sqsubseteq d$ for every j, since the guards are either local or mutually exclusive, and $d \sqsubseteq c \neq \mathit{false}$.

In order to generalize the above observation to a computation $\xi \in \mathit{Comp}_{\text{fin}}(A, c)$, with $\mathit{Result}(\xi) \neq \mathit{false}$, we consider a computation $\xi'' \in \mathit{Comp}_{\text{fin}}(A, c)$ which is obtained from ξ by changing the order of activation of the parallel agents, in such a way that at each step the active agent is (a) enabled to make the same step (in particular, in case of a choice construct, to take the same choice) as it does when it is active in ξ, and (b) in order to be enabled for this step it requires a minimal store among the agents which satisfy (a). It is easy to see that $\mathit{Result}(\xi'') = \mathit{Result}(\xi)$. Note also that each intermediate store of ξ'' is different from *false*, since it is smaller or equal to $\mathit{Result}(\xi)$.

By a straightforward induction on the length of the computation ξ'' a computation $\xi' \in \mathit{Comp}_{\text{fin}}(A, d)$ can be constructed which mimics a prefix of ξ'', in the sense that:

> the length n of ξ' is smaller or equal to the length of ξ'', and, for each $i \in [0, n]$, if $\gamma'_i \longrightarrow \gamma'_{i+1}$ and $\gamma''_i \longrightarrow \gamma''_{i+1}$ are the i-th steps of ξ' and ξ'' respectively, then γ'_i and γ''_i have the same agent, the active agent is the same in both transitions, and, if a choice construct is involved, then the selected branch is the same. Furthermore, the store e' in γ'_i is smaller or equal to the store e'' in γ''_i.

Hence we conclude that, either ξ' goes for all the length of ξ'', or it arrests before that, but in any case the final store is smaller or equal. $\square$

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

Example 4.15 Let $c, d \in \mathit{Con}$ with $d \sqsubseteq c \neq \mathit{false}$. Consider the ccp agent

$$\begin{array}{lcl} A & = & \mathbf{ask}(c) \rightarrow \mathbf{tell}(c) \\ & & + \\ & & \mathbf{ask}(d) \rightarrow \mathbf{tell}(\mathit{false}). \end{array}$$

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

The reason why in Lemma 4.14 we have to exclude the case $\mathit{Result}(\xi) = \mathit{false}$ is that the choices which lead to finite computations might be not enabled by a certain store, whereas they are of course enabled by *false*. Note that *false* is the only constraint for which the mutual-exclusion condition does not avoid the enabledness of more than one branch. This is no problem when infinite computations are considered too, but, if we consider only finite ones, then we have the following counterexample.

Example 4.16 Let $c, d \in \mathit{Con}$ with $d \sqcup c = \mathit{false}$ and consider the restricted-choice ccp definitions:

$$\begin{array}{lcl} p(x) & \text{:-} & \mathbf{ask}(c) \rightarrow \mathbf{tell}(c) \\ & & + \\ & & \mathbf{ask}(d) \rightarrow q(x) \\ & & \\ q(x) & \text{:-} & q(x). \end{array}$$

Then $\langle p(x), \mathit{false}\rangle$ has a computation with final store *false* whereas $\langle p(x), d\rangle$ has no finite computations.

Next we show that $\mathit{Quie}_{\mathrm{fin}}$ is F-closed, i.e. $F(\mathit{Quie}_{\mathrm{fin}}) \subseteq \mathit{Quie}_{\mathrm{fin}}$. For this result we need to make an assumption on the kind of constraint system we are dealing with. Namely we assume that $\exists_x \mathit{false} = \mathit{false}$. This property is rather reasonable, and it is satisfied by most constraint systems used in practice.

Proposition 4.17 *For every restricted-choice ccp agent A, $F(\mathit{Quie}_{\mathrm{fin}})(A) \subseteq \mathit{Quie}_{\mathrm{fin}}(A)$.*

Proof By structural induction on A.

A = stop. Obvious.

A = tell(c). Obvious.

$A = \sum_{=1} \mathbf{ask}(c\ ) \rightarrow A$. Assume $c \in F(\mathit{Quie}_{\mathrm{fin}})(A)$. Then either $c \in \bigcap_{i=1}^{n} \overline{\Uparrow c_i}$ or there exists $i \in [1, n]$ such that $c \in \Uparrow c_i \cap F(\mathit{Quie}_{\mathrm{fin}})(A_i)$, i.e. $c_i \sqsubseteq c$ and $c \in F(\mathit{Quie}_{\mathrm{fin}})(A_i)$. In the first case $\langle A, c\rangle$ deadlocks 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, $c \in \mathit{Quie}_{\mathrm{fin}}(A_i)$. Therefore $c \in \mathit{Quie}_{\mathrm{fin}}(A)$.

$A = A_1 \parallel A_2$. Assume $c \in F(\mathit{Quie}_{\mathrm{fin}})(A)$. Then $c \in F(\mathit{Quie}_{\mathrm{fin}})(A_1)$ and $c \in F(\mathit{Quie}_{\mathrm{fin}})(A_2)$. By the inductive hypothesis, $c \in \mathit{Quie}_{\mathrm{fin}}(A_1)$ and $c \in \mathit{Quie}_{\mathrm{fin}}(A_2)$. 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.

$A = \exists\ B$. This is the only case in which we need the restricted-choice hypothesis. Let $c \in F(\mathit{Quie}_{\mathrm{fin}})(\exists_x B)$. Then there exists $d \in F(\mathit{Quie}_{\mathrm{fin}})(B)$ such that $\exists_x c = \exists_x d$. If $d = \mathit{false}$ then $\exists_x c = \exists_x \mathit{false} = \mathit{false}$, hence $c = \mathit{false}$ and we are done. Otherwise, observe that by the inductive hypothesis $d \in \mathit{Quie}_{\mathrm{fin}}(B)$. So there exists a computation $\langle B, d\rangle \longrightarrow^* \langle B', d\rangle \not\longrightarrow$. Since $\exists_x c \sqsubseteq d$ we can apply Lemma 4.14: There exists a computation $\langle B, \exists_x c\rangle \longrightarrow^* \langle B'', \exists_x c \sqcup d'\rangle \not\longrightarrow$, for some B'' and d' with $d' \sqsubseteq d$. It follows that $\langle \exists_x B, c\rangle \longrightarrow^* \langle \exists_x^{d'} B'', c \sqcup \exists_x d'\rangle \not\longrightarrow$. Finally we observe that, since $d' \sqsubseteq d$, we have $\exists_x d' \sqsubseteq \exists_x d = \exists_x c \sqsubseteq c$ (by axioms A1 and A2 of cylindric constraint systems), and so $c \sqcup \exists_x d' = c$.

$A = p(x)$. Assume $c \in F(\mathit{Quie}_{\mathrm{fin}})(p(x))$. Then $c \in \mathit{Quie}_{\mathrm{fin}}(\Delta_y^x B)$, where $p(y)$:- B is the declaration for p in D, hence $c \in \mathit{Quie}_{\mathrm{fin}}(p(x))$. □

The Theorem of Knaster-Tarski ensures that the least fixpoint of F coincides with the least F-closed interpretation, i.e. $\mathit{lfp}(F)(A) = \mathit{min}\{I \mid F(I) \subseteq I\}$. Hence from Proposition 4.17 we have:

Corollary 4.18 *For every restricted-choice ccp agent A, $lfp(F)(A) \subseteq Quie_{\mathrm{fin}}(A)$.*

We show now the other inclusion, which actually holds for full ccp.

Proposition 4.19 *For every ccp agent A, $Quie_{\mathrm{fin}}(A) \subseteq F \uparrow \omega(A)$.*

Proof Assume $c \in Quie_{\mathrm{fin}}(A)$. Let ξ be a derivation

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

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 simultanous induction on i, the length of the computation, and on the structure of A.

A = stop. Immediate since $c \in F \uparrow 1(A)$.

A = tell(c). Immediate since $c \in F \uparrow 1(A)$.

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

$A = A_1 \parallel A_2$. Since $c \in Quie_{\mathrm{fin}}(A_1 \parallel A_2)$, 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 get stuck. Hence both $c \in Quie_{\mathrm{fin}}(A_1)$ and $c \in Quie_{\mathrm{fin}}(A_2)$ hold. By the (structural) inductive hypothesis, there exist n_1, n_2 such that $c \in F \uparrow n_1(A_1)$ and $c \in F \uparrow n_2(A_2)$. Since $\{F \uparrow i\}_i$ is an increasing chain, we have $c \in F \uparrow n(A_1)$ and $c \in F \uparrow n(A_2)$, where $n = max\{n_1, n_2\}$. Therefore $c \in F \uparrow n(A_1 \parallel A_2)$.

$A = \exists$ B. Since $c \in Quie_{\mathrm{fin}}(\exists_x B)$, there exists a computation $\langle B, \exists_x c\rangle \longrightarrow^* \langle B', \exists_x c \sqcup d\rangle \not\longrightarrow$, for some d such that $\exists_x d \sqsubseteq c$. Hence $\exists_x c \sqcup d \in Quie_{\mathrm{fin}}(B)$. By the (structural) inductive hypothesis, there exist n such that $\exists_x c \sqcup d \in F \uparrow n(B)$. Finally observe that $\exists_x(\exists_x c \sqcup d) = \exists_x c \sqcup \exists_x d = \exists_x(c \sqcup \exists_x d) = \exists_x c$.

$A = p(x)$. Let ξ' be the postfix of ξ starting from γ_1. Then $\xi' \in Comp(\Delta_y^x B, c)$ where $p(x)$:- B is the declaration for p in D. We have that $c = Result(\xi')$, hence $c \in Quie_{\mathrm{fin}}(\Delta_y^x B)$. Furthermore the length of ξ' is $i-1$, so by the inductive hypothesis there exists n such that $c \in F \uparrow n(\Delta_y^x B)$. Therefore $c \in F \uparrow n+1(A)$. □

Finally, by Corollary 4.18, Proposition 4.19 and Remark 4.10 it follows:

Theorem 4.20 *For every restricted-choice ccp agent A, $Quie_{\mathrm{fin}}(A) = \mathcal{D}_{\mathrm{fin}}(A)$.*

We show now that $\mathcal{O}_{\mathrm{fin}}^{\mathrm{uc}}$ can be retrieved from $Quie_{\mathrm{fin}}$.

Proposition 4.21

(i) *For every ccp agent A, for every $d \in Con$, we have $\mathcal{O}_{\mathrm{fin}}^{\mathrm{uc}}(A)d \subseteq \Uparrow(\Uparrow d \cap Quie_{\mathrm{fin}}(A))$.*

(ii) *For every restricted-choice ccp agent A, for every $d \in Con$, we have $\Uparrow(\Uparrow d \cap Quie_{\mathrm{fin}}(A)) \subseteq \mathcal{O}_{\mathrm{fin}}^{\mathrm{uc}}(A)d$.*

Proof

(i) If $d' \in \mathcal{O}_{\mathrm{fin}}^{\mathrm{uc}}(A)d$ then either $d' = \mathit{false}$, and in this case we are done, or there exists $d'' \in \mathcal{O}_{\mathrm{fin}}(A)d$ such that $d \sqsubseteq d'' \sqsubseteq d'$. In this latter case we have $d'' \in \Uparrow d$ and $d'' \in Quie_{\mathrm{fin}}(A)$. Therefore $d' \in \Uparrow(\Uparrow d \cap Quie_{\mathrm{fin}}(A))$.

(ii) Let $c \in \Uparrow(\Uparrow d \cap Quie_{\mathrm{fin}}(A))$. If $c = \mathit{false}$ then we are done. Otherwise there exists $c' \neq \mathit{false}$ such that $c' \sqsubseteq c$ and $c' \in \Uparrow d \cap Quie_{\mathrm{fin}}(A)$. By Lemma 4.14, there exists $d' \in \mathcal{O}_{\mathrm{fin}}(A)d$ such that $d' \sqsubseteq c'$. Therefore $c' \in \mathcal{O}_{\mathrm{fin}}^{\mathrm{uc}}(A)d$, hence, a fortiori, $c \in \mathcal{O}_{\mathrm{fin}}^{\mathrm{uc}}(A)d$. □

From Proposition 4.21 it follows that for every restricted-choice ccp agent A, for every $d \in Con$, $\mathcal{O}_{\mathrm{fin}}^{\mathrm{uc}}(A)d = \Uparrow(\Uparrow d \cap Quie_{\mathrm{fin}}(A))$ holds. Therefore from Theorem 4.20 we obtain that $\mathcal{O}_{\mathrm{fin}}^{\mathrm{uc}}$ can be retrieved from $\mathcal{D}_{\mathrm{fin}}$.

Theorem 4.22 *For every restricted-choice ccp agent A, $\mathcal{O}_{\mathrm{fin}}^{\mathrm{uc}}(A)c = \Uparrow(\Uparrow c \cap \mathcal{D}_{\mathrm{fin}}(A))$.*

Observe that also $\mathcal{D}_{\text{fin}}$ can be retrieved from $\mathcal{O}^{\text{uc}}_{\text{fin}}$, in fact for each ccp agent A, $\mathcal{D}_{\text{fin}}(A) = \mathit{Quie}_{\text{fin}}(A) = \{c \in \mathit{Con} \mid c \in \mathcal{O}^{\text{uc}}_{\text{fin}}(A)c\}$. Therefore $\mathcal{D}_{\text{fin}}$ and $\mathcal{O}^{\text{uc}}_{\text{fin}}$ are equivalent.

Note that the mutually-exclusive choice is the only case in which the assumption $\mathit{Result}(\xi) \neq \mathit{false}$ of Lemma 4.14 is used. This means that for local-choice ccp we could consider the more refined definition $\mathcal{O}^{\text{uc}}_{\text{fin}}(A)c = \Uparrow\mathcal{O}_{\text{fin}}(A)c$, we could allow interpretations not containing *false*, and avoid to force *false* into $\mathit{Quie}_{\text{fin}}$. Furthermore we would not need the assumption $\exists_x \mathit{false} = \mathit{false}$. The proof of the correspondence would remain the same, with the difference that the bottom-up iterations of F would start from the empty interpretation, and that the result $\mathcal{O}^{\text{uc}}_{\text{fin}}(A)c = \Uparrow(\Uparrow c \cap \mathcal{D}_{\text{fin}}(A))$ would be more significant, since the observables would contain more information.

4.2 Correspondence between the observables and the denotational semantics: finite + infinite case

We consider now the correspondence between $\mathcal{D}$ and all the observables, including the infinite ones. It is convenient to extend the notion of *quiescent points* to the infinite case. From the point of view of an external observer, who can see the changes of the store, but not the internal activity of the agent, a store reached at a certain point of a (finite or infinite) computation is quiescent iff it will not be modified by the rest of the computation. Thus we give the following definition.

Definition 4.23 The interpretation $\mathit{Quie} : \mathit{Agents} \to \mathcal{P}_{\text{wf}}(\mathit{Con})$, which gives the *quiescent points* of an agent A, is defined as

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

Observe that we don't need to add the value *false* like in the finite case: *false* is always included, by definition of $\mathcal{O}$.

We will show that Quie coincides with $\mathit{gfp}(F)$. We start with proving that Quie enjoys the reverse property of Proposition 4.17. Note that the fairness assumption and Axiom A5 play a fundamental role here.

Proposition 4.24 *For every ccp agent A, $\mathit{Quie}(A) \subseteq F(\mathit{Quie})(A)$.*

Proof By an easy structural induction on A. We discuss only the parallel composition and the hiding operator; the other cases are analogous.

- $\boldsymbol{A = A_1 \parallel A_2}$. Let $c \in \mathit{Quie}(A_1 \parallel A_2)$. By the fairness assumption it follows that $c \in \mathit{Quie}(A_1)$ and that $c \in \mathit{Quie}(A_2)$. By the induction hypothesis we then have $c \in F(\mathit{Quie})(A_1)$ and $c \in F(\mathit{Quie})(A_2)$. Thus by definition of F we conclude that $c \in F(\mathit{Quie})(A)$.
- $\boldsymbol{A = \exists\ B}$. Let $c \in \mathit{Quie}(\exists_x B)$, and let ξ be the corresponding computation. Consider the corresponding computation ξ' for $\langle B, \exists_x c\rangle$, and let $\exists_x c \sqcup d = \mathit{Result}(\xi') \in \mathit{Quie}(B)$. By structural induction, $\exists_x c \sqcup d \in F(\mathit{Quie})(B)$ holds. We prove now the equality

$$\exists_x(\exists_x c \sqcup d) = \exists_x c, \tag{4}$$

  which allows us to conclude $c \in F(\mathit{Quie})(A)$. Equality (4) is obvious if ξ (and thus ξ') is finite. If ξ (and thus ξ') is infinite then $\exists_x c \sqcup d = \bigsqcup_i(\exists_x c \sqcup d_i)$, where $\exists_x c \sqcup d_i$ is the store at the i-th step of ξ'. By Axiom A5 (of cylindric constraint systems) we have that $\exists_x \bigsqcup_i(\exists_x c \sqcup d_i) = \bigsqcup_i \exists_x(\exists_x c \sqcup d_i) = \bigsqcup_i(\exists_x c \sqcup \exists_x d_i) = \bigsqcup_i \exists_x(c \sqcup \exists_x d_i) = \exists_x c$ (note that $c \sqcup \exists_x d_i$ is the i-th store of ξ, which is c by assumption). □

The Theorem of Knaster-Tarski ensures that $\mathit{gfp}(F)(A) = \mathit{max}\{I \mid I \subseteq F(I)\}$. Hence from Proposition 4.24 we have:

Corollary 4.25 *For every ccp agent A, $\mathit{Quie}(A) \subseteq \mathit{gfp}(F)(A)$.*

We show now that, for restricted-choice ccp, also the other inclusion holds. We proceed by proving that $F\downarrow\omega(A) \subseteq Quie(A)$. To this end we define the n^{th} syntactical approximation of A, denoted by A^n. The idea is that in the n^{th} level of the approximation each procedure call is replaced by the syntactical approximation of its body at the previous level. The base case is represented by **stop**, which here might correspond to an unfinished computation.

Definition 4.26 Given the set of declarations $D = \{p_1(y) : -B_1, \dots p_k(y) : -B_k\}$, we define A^n as follows:

$$\begin{aligned} A^0 &= \mathbf{stop}, \\ A^{n+1} &= A[B_1^n/p_1, \dots, B_k^n/p_k], \end{aligned}$$

where $[B_1^n/p_1, \dots, B_k^n/p_k]$ denotes the operation of replacing a call $p_i(x)$ by the agent $\mathbf{ask}(true) \rightarrow \Delta_x^y B_i^n$.

In the previous definition the silent step $\mathbf{ask}(true)$ is added, when replacing a procedure call by its body, in order to have an exact correspondence with the operational rule.

Note that A^n is a finite agent, in the sense that it will not give rise to infinite computations. The nice property of these agents A^n is that their quiescent points exactly correspond to the downward iterations of F applied to A.

Proposition 4.27 *For any restricted-choice ccp agent A we have $F\downarrow n(A) = Quie(A^n)$.*

Proof The proof proceeds by induction on n and on the structure of A. We use the following observation: since A^n is finite, $Quie(A^n) = Quie_{\mathrm{fin}}(A^n)$ holds. Thus, by the results of previous section, we derive

$$F(Quie)(A^n) = Quie(A^n). \tag{5}$$

We illustrate the base and the cases of the procedure call and of the hiding operator. The proof for the other operators is similar to the latter.

$\boldsymbol{A = \mathbf{stop}}$. Immediate from the observation that $\mathbf{stop}^n = \mathbf{stop}$, for every n.

$\boldsymbol{A = \exists\ B}$.

$$\begin{aligned} & F\downarrow n(\exists_x B) \\ = & \quad \{\text{definition of } F\} \\ & \exists_x F\downarrow n(B) \\ = & \quad \{\text{structural induction}\} \\ & \exists_x Quie(B^n) \\ = & \quad \{\text{by } (5)\} \\ & \exists_x F(Quie)(B^n) \\ = & \quad \{\text{definition of } F\} \\ & F(Quie)(\exists_x(B^n)) \\ = & \quad \{\text{by } (5)\} \\ & Quie(\exists_x(B^n)) \\ = & \quad \{\text{since } \exists_x(B^n) = (\exists_x B)^n\} \\ & Quie((\exists_x B)^n). \end{aligned}$$

M1	$\dfrac{\langle A, c\rangle \longrightarrow \langle A', c'\rangle \quad \langle B, c\rangle \longrightarrow \langle B', d'\rangle}{\langle A \parallel B, c\rangle \longrightarrow \langle A' \parallel B', c' \sqcup d'\rangle}$
M2	$\dfrac{\langle A, c\rangle \longrightarrow \langle A', d\rangle \quad \langle B, c\rangle \not\longrightarrow}{\begin{array}{c}\langle A \parallel B, c\rangle \longrightarrow \langle A' \parallel B, d\rangle \\ \langle B \parallel A, c\rangle \longrightarrow \langle B \parallel A', d\rangle\end{array}}$

Table 3: Maximal parallelism.

$\boldsymbol{A = p(x)}$. For $n = 0$ we have $F \downarrow 0(p(x)) = Con = Quie(\mathbf{stop}) = Quie(p(x)^0)$. For the other cases we reason as follows

$$
\begin{array}{ll}
 & F \downarrow n+1(p(x)) \\
= & \quad \{\text{assuming that } p(y) \text{ :- } B \text{ is the declaration for } p \text{ in } D\} \\
 & F \downarrow n(\Delta^x_y B) \\
= & \quad \{\text{induction hypothesis on } n\} \\
 & Quie((\Delta^x_y B)^n) \\
= & \\
 & Quie(\mathbf{ask}(true) \rightarrow (\Delta^x_y B)^n) \\
= & \quad \{\text{Definition 4.26}\} \\
 & Quie(p(x)^{n+1}).
\end{array}
$$

□

Thanks to the above proposition, it is now sufficient to show that $\bigcap_n Quie(A^n) \subseteq Quie(A)$. A proof of this kind of property usually exploits the finitely branching structure of the computation tree of an agent. Unfortunately the fair computations generated by a certain configuration cannot be ordered into a finitely branching tree. Note that the tree generated by the transition system is finitely branching but in general it contains also unfair infinite paths.

In order to circumvent this problem, we introduce a new operational semantics based on maximal parallelism, which allows processes to proceed simultaneously, and which obliges processes to proceed as soon as they are enabled. The advantage of this semantics is that the corresponding computation tree contains only branches relative to fair computations. Note that the branching structure of a computation tree here comes only from the choice operator.

Definition 4.28 The maximal-parallelism semantics of ccp is given by the transition system $\mathcal{T}_{\rm mp}$ which is obtained by replacing Rule R4 of Table 1 by Rules M1 and M2 of Table 3.

We will indicate by $Comp_{\rm mp}(A, c)$ the computations generated by the maximal-parallelism semantics starting from $\langle A, c\rangle$. Moreover, we will use $Quie_{\rm mp}(A)$ to denote the quiescent points of A. Formally:

$$Quie_{\rm mp}(A) = \{Result(\xi) \mid \xi \in Comp_{\rm mp}(A, c)\}.$$

It is easy to see that $Quie_{\rm mp}(A) \subseteq Quie(A)$. Furthermore, for restricted-choice ccp agents we can prove also the inverse inclusion.

Proposition 4.29 *For any restricted-choice ccp agent A we have $Quie(A) = Quie_{\mathrm{mp}}(A)$.*

Proof The inclusion $Quie_{\mathrm{mp}}(A) \subseteq Quie(A)$ is immediate and actually holds for all ccp agents. The other inclusion basically follows from the simple observation that, when a mutually exclusive choice is enabled in a store, then in all smaller stores it gives rise to exactly the same transition, unless it is stuck. This observation allows us to reorder a $\mathcal{T}$-computation in such a way that it can be partitioned into blocks of consecutive transitions coming from the activation of the agents enabled in the initial configuration of the block. Such a $\mathcal{T}$-computation can be directly transformed into an equivalent (i.e. giving the same result) $\mathcal{T}_{\mathrm{mp}}$-computation. □

By the above proposition it is now sufficient to show that $\bigcap_n Quie_{\mathrm{mp}}(A^n) \subseteq Quie_{\mathrm{mp}}(A)$. As discussed before, in order to prove this inclusion it is convenient to reason about the structure of the computation tree.

Definition 4.30 A *computation tree* based on a transition system for a configuration γ, is a tree in which

(i) the root is γ, and

(ii) for each node γ', for each transition $\gamma' \longrightarrow \gamma''$, γ'' is a son of γ'.

Note that the computation tree generated by $\mathcal{T}_{\mathrm{mp}}$ is finitely branching, because each node contains a finite number of agents, and each choice construct contains a finite number of alternatives.

We will denote by $Tree_{\mathrm{mp}}(A, c)$ the computation tree generated by $\mathcal{T}_{\mathrm{mp}}$ starting with $\langle A, c\rangle$. Furthermore we will denote by $\mid Tree_{\mathrm{mp}}(A, c) \mid_n$ the maximal tree-prefix in which at most n *nested procedure calls* are activated. It is easy to see that $T_1 = \mid Tree_{\mathrm{mp}}(A, c) \mid_n$ is isomorphic to $T_2 = Tree_{\mathrm{mp}}(A^n, c)$, where the isomorphism on the nodes satisfies the following:

> Each node $\langle B_1, d\rangle$ in T_1 corresponds to a node $\langle B_2, d\rangle$ such that B_2 is obtained from B_1 by replacing each procedure call $p(x)$ in B_1 by $\mathbf{ask}(true) \rightarrow \Delta^x_y B^{n-k}$, where $p(y)$:- B is the declaration for p, and k is the number of procedure calls which have generated $p(x)$.

We can now prove that $\bigcap_n Quie_{\mathrm{mp}}(A^n) \subseteq Quie_{\mathrm{mp}}(A)$.

Proposition 4.31 *For any ccp agent A we have $\bigcap_n Quie_{\mathrm{mp}}(A^n) \subseteq Quie_{\mathrm{mp}}(A)$.*

Proof Let $c \in \bigcap_n Quie_{\mathrm{mp}}(A^n)$. Consider the following two possibilities.

1. There are infinitely many nodes in $Tree_{\mathrm{mp}}(A, c)$ with store c. In this case, by the König lemma there exists a infinite branch with all nodes having store c. Therefore the result of this branch is c, hence c is a quiescent point of A.

2. There are finitely many nodes $Tree_{\mathrm{mp}}(A, c)$ with store c. We show that in this case there exists at least one leaf node with store c, and therefore c is a resting point of A. Suppose it is not true: then there exists a k such that all nodes with store c occur in $\mid Tree_{\mathrm{mp}}(A, c) \mid_k$. This means that $\mid Tree_{\mathrm{mp}}(A, c) \mid_{k+1}$ does not contain any leaf-node with store c. But then, due to the above discussed correspondence between $Tree_{\mathrm{mp}}(A^n, c)$ and $\mid Tree_{\mathrm{mp}}(A, c) \mid_n$, also $Tree_{\mathrm{mp}}(A^{k+1}, c)$ does not contain any leaf-node with store c, hence $c \notin Quie_{\mathrm{mp}}(A^{k+1}, c)$. □

The following corollary summarizes the result obtained by Propositions 4.27, 4.29 and 4.31.

Corollary 4.32 *For every restricted-choice ccp agent A, we have $F\!\downarrow\!\omega(A) \subseteq Quie(A)$.*

Finally, by Corollaries 4.25, 4.32 and Remark 4.10 it follows:

Theorem 4.33 *For every restricted-choice ccp agent A, $Quie(A) = gfp(F)(A)$.*

We show now that $\mathcal{O}^{\mathrm{uc}}$ can be retrieved from *Quie*. The following proposition is the analogous of Proposition 4.21; the proof is similar.

Proposition 4.34

(i) *For every ccp agent A, for every $d \in Con$, we have $\mathcal{O}^{\mathrm{uc}}(A)d \subseteq \Uparrow(\Uparrow d \cap Quie(A))$.*

(ii) *For every restricted-choice ccp agent A, for every $d \in Con$, we have $\Uparrow(\Uparrow d \cap Quie(A)) \subseteq \mathcal{O}^{\mathrm{uc}}(A)d$.*

Finally next theorem, which follows immediately from the above proposition and from Theorem 4.33, states that one can retrieve $\mathcal{O}^{\mathrm{uc}}$ from $\mathcal{D}$.

Theorem 4.35 *For every restricted-choice ccp agent A, $\mathcal{O}^{\mathrm{uc}}(A)c = \Uparrow(\Uparrow c \cap \mathcal{D}(A))$.*

As in the finite case, we observe that obviously we can also retrieve $\mathcal{D}$ from $\mathcal{O}^{\mathrm{uc}}$; i.e. $\mathcal{O}^{\mathrm{uc}}$ and $\mathcal{D}$ give the same information.

4.3 Problems in extending the model to full ccp

There are two main reasons why the the model $\mathcal{D}$ does not work for full ccp. One is that the quiescent points cannot be modeled compositionally:

Example 4.36 Consider the two ccp agents

$$\begin{aligned} A_1 &= \mathbf{ask}(x=a) \rightarrow \mathbf{tell}(y=a) \\ &\quad + \\ &\quad \mathbf{ask}(true) \rightarrow \mathbf{tell}(false), \\ A_2 &= \mathbf{tell}(x=a \sqcup y=a). \end{aligned}$$

Then A_1 and A_2 have the same quiescent points ($x = a \sqcup y = a$ and all the bigger constraints), but $x = a \sqcup y = a$ is a quiescient point for $\exists_x A_2$ and not for $\exists_x A_1$.

The other reason is that $\mathcal{O}^{\mathrm{uc}}(A)$ cannot be retrieved from $Quie(A)$. In the above example, in fact, $\mathcal{O}^{\mathrm{uc}}(A_1)true = \{false\}$ whereas $\mathcal{O}^{\mathrm{uc}}(A_2)true = \Uparrow(x = a \sqcup y = a)$.

4.4 Problems in refining the model so to retrieve the exact set of results

If we restrict to deterministic ccp ([29]), namely the sublanguage of ccp where instead of the choice construct $\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i$ we can have only a synchronization statement of the form $c \rightarrow A$, then the model $\mathcal{D}$ allows us to retrieve $\mathcal{O}$. In fact in this language a computation starting from $\langle A, c\rangle$ can produce only one result: the least quiescent point of A which lies over c.

However, as soon as we consider a nondeterministic language, there are two main problems which come into the scene, which we could classify as follows:

1. Combination of nondeterminism and synchronization,
2. Combination of nondeterminism and infinite behavior.

About the first problem, the point is that we cannot distinguish whether two quiescient points come from different computation branches or rather they represent different suspension points in the same branch.

Example 4.37 Let $c \sqsubseteq d$ and consider the two agents

$$\begin{aligned} A_1 &= \mathbf{ask}(true) \rightarrow \mathbf{tell}(true) \\ &\quad + \\ &\quad \mathbf{ask}(true) \rightarrow \mathbf{tell}(d), \\ A_2 &= \mathbf{ask}(c) \rightarrow \mathbf{tell}(d). \end{aligned}$$

Then A_1 and A_2 have the same quiescent points ($true$ and d), but $\mathcal{O}(A_1)true = \{true, d\}$ whereas $\mathcal{O}(A_2)true = \{true\}$.

To solve this problem one has to encode the necessary information about the branching structure and about synchronization.

For restricted-choice ccp a solution has been proposed in [11]. The idea is to consider the denotation of a process as set of sets of quiescient points. Intuitively the inner set represents the activity of the process along one branch. Note that the branching structure is very poor: we don't see where are the choice points but only how many possible different branches are there.

For full ccp there have been two independent proposals, both shown fully abstract wrt $\mathcal{O}_{\mathrm{fin}}$[8]. One approach ([7]) is based on *reactive sequences*, i.e. sequences which encode possible behaviors of the agent in terms of assumptions-reactions on the store. The other approach ([29]) is based on *bounded traces operators*, which are particular closure operators. Intuitively, bounded trace operators are possible linearizations of the activity of a process, not only with respect to the control structure, but also with respect to the data structure. In other words, they are relative to a particular path along the constraint system. These two approaches are of course equivalent because they are both fully abstract; for the exact correspondence between them see [7].

Both the above mentioned approaches do not capture the infinite observables. However there is in literature a standard solution to extend domains based on traces (or on analogous structures) so to make it possible to capture the infinite behavior. This is the well known metric approach ([2]). The basic idea is to define the operators in such a way that every action of the agent, including the recursive call, makes its presence visible by augmenting the traces. In this way the activity of a process corresponds semantically to a contraction, which always has a fixpoint.

The problem of this approach is that in general it is not fully abstract. Recently [23] has shown that a fully abstract model for ccp can be obtained by using categorical techniques. More specifically, [23] applies the Lehmann's powerdomain construction ([17]) over the basic domain of reactive sequences.

Concerning the second problem (combination of nondeterminism and infinite behavior), the point is that it is difficult to find the appropriate domain structure in which to define an operator characterizing the limits of infinite computations, unless we encode also the history of the computation as it is done in [23].

In the work of [23] however it is not clear whether the (reactive) sequences are necessary only to deal with synchronization or they play a role also wrt the infinite behavior. In the next section we investigate what are the possibilities of capturing both nondeterminism and infinite limit results by using very abstract domains, like the one described in previous sections, which encode only observable changes.

To this purpose we abstract from the problem of synchronization and consider the sublanguage of ccp which has no suspension mechanism: the language of *constraint logic programming.*

5 Infinite computations in constraint logic programming

In this section we investigate the problem of characterizing the infinite behaviors of a sublanguage of ccp obtained by restricting the choice construct to the following form: $\Sigma_{i=1}^{n}\mathbf{ask}(true) \rightarrow A_i$. So we allow only local choice. The resulting language corresponds to the language of "constraint logic programming" ([14]). Alternatively we can describe clp via the following grammar:

$$\begin{array}{rl} \textit{Declarations} & D ::= \epsilon \mid p(x) \text{ :- } A \mid D, D \\ \textit{Agents} & A ::= \mathbf{tell}(c) \mid A + B \mid A \parallel A \mid \exists_x A \mid p(x) \end{array}$$

The local choice between two agents A and B is represented by $A + B$. From Rule R2 of the transition system $\mathcal{T}$ we derive the rule for local choice as given in Table 4[9].

More importantly, the absence of any synchronization mechanism in clp allows one to retrieve $\mathcal{O}(A)c$ from its so called *S-semantics* ([10]), which consists of the results obtained by running an agent in the empty store, i.e. $\mathcal{O}(A)true$.

[8] The notion of observables was actually a bit different from $\mathcal{O}_{\mathrm{fin}}$, but the difference is unessential here.

[9] Actually in the original definition of clp the choice is not blind as in Table 4: in [14] the choice is driven by the requirement that the immediate tell actions of the selected branch will not bring to an inconsistent store. However this condition is only motivated by efficiency reasons, and the only consequence it has on the observables is that it eliminates the *false* result. For the purposes of this work we can ignore this difference.

$$\langle A_1 + A_2, c\rangle \longrightarrow \langle A_i, c\rangle \qquad i \in \{1,2\}$$

Table 4: Local choice.

Proposition 5.1 *For any clp agent A we have $\mathcal{O}(A)c = \{c \sqcup d \mid d \in \mathcal{O}(A)true\}$.*

Proof

- $\supseteq$. First we prove by induction on the length of the derivation of a transition $\langle A, c\rangle \longrightarrow \langle B, d\rangle$ that $\langle A, c \sqcup e\rangle \longrightarrow \langle B, d \sqcup e\rangle$, for any e. We show the case of $A = \exists_x^{d_1} B$ (the other cases being straightforward): So let $\langle \exists_x^{d_1} B, c\rangle \longrightarrow \langle \exists_x^{d_2} B', c \sqcup \exists_x d_2\rangle$. It follows that $\langle B, \exists_x c \sqcup d_1\rangle \longrightarrow \langle B', \exists_x c \sqcup d_2\rangle$. By the induction hypothesis we have $\langle B, \exists_x(c \sqcup e) \sqcup d_1\rangle \longrightarrow \langle B', \exists_x(c \sqcup e) \sqcup d_2\rangle$ (note that $\exists_x c \sqsubseteq \exists_x(c \sqcup e)$, and so $\exists_x c \sqcup d_i \sqcup \exists_x(c \sqcup e) = \exists_x(c \sqcup e) \sqcup d_i$, $i = 1, 2$). From this we derive $\langle \exists_x^{d_1} B, (c \sqcup e)\rangle \longrightarrow \langle \exists_x^{d_2} B', (c \sqcup e) \sqcup \exists_x d_2\rangle$.

  By induction on the length of the computation it now immediately follows that if $\langle A, c\rangle \longrightarrow^* \langle B, d\rangle$ then $\langle A, c \sqcup e\rangle \longrightarrow^* \langle B, d \sqcup e\rangle$, for any e. Furthermore, since there is no suspension mechanism, if $\langle B, d\rangle \not\longrightarrow$ then $\langle B, f\rangle \not\longrightarrow$ holds for any f. This part of the proposition then follows by taking $c = true$.

- $\subseteq$. First we prove by induction on the length of the derivation of a transition $\langle A, c\rangle \longrightarrow \langle B, d\rangle$ that $\langle A, true\rangle \longrightarrow \langle B, d'\rangle$, for some d' such that $d = c \sqcup d'$. We show the case of $A = \exists_x^{d_1} B$ (the other cases being straightforward): So let $\langle \exists_x^{d_1} B, c\rangle \longrightarrow \langle \exists_x^{d_2} B', c \sqcup \exists_x d_2\rangle$. It follows that $\langle B, \exists_x c \sqcup d_1\rangle \longrightarrow \langle B', \exists_x c \sqcup d_2\rangle$. Without loss of generality we may assume that $d_2 = d_1 \sqcup d'$, where d' is the constraint added to the store, in case the transition from B to B' involves the execution of a tell action, or, in case the transition involves the expansion of a procedure call, d' equals $true$. It follows that $\langle B, true\rangle \longrightarrow \langle B', d'\rangle$. By the argument above it follows that $\langle B, d_1\rangle \longrightarrow \langle B', d_2\rangle$, from which we derive that $\langle \exists_x^{d_1} B, true\rangle \longrightarrow \langle \exists_x^{d_2} B', \exists_x d_2\rangle$.

  Using the above we prove by induction on the length of the computation $\langle A, c\rangle \longrightarrow^* \langle B, d\rangle$ that, for some d' such $d = c \sqcup d'$, $\langle A, true\rangle \longrightarrow^* B, d'\rangle$: Let $\langle A, c\rangle \longrightarrow \langle A', c'\rangle \longrightarrow^* \langle B, d\rangle$. By the induction hypothesis we have $\langle A', true\rangle \longrightarrow^* \langle B, d'\rangle$, for some d' such that $d = c' \sqcup d'$. Moreover, by the above argument, $\langle A, true\rangle \longrightarrow \langle A', d''\rangle$, for some d'' such that $c' = c \sqcup d''$. It follows that $\langle A', d''\rangle \longrightarrow^* \langle B, d' \sqcup d''\rangle$, and thus $\langle A, true\rangle \longrightarrow^* \langle B, d' \sqcup d''\rangle$. Furthermore $d = c' \sqcup d' = c \sqcup d' \sqcup d''$. Since there is no suspension mechanism this inclusion is hereby proven. □

The mapping $\mathcal{O}(\cdot)true : Agents \rightarrow \mathcal{P}(Con)$ can therefore play here the role of the resting points in previous section. The difference is that $\mathcal{O}(\cdot)true$ is more refined than $Quie$:

Example 5.2 Let $c \sqsubseteq d$ and consider the two agents

$$\begin{aligned} A_1 &= \mathbf{tell}(c) + \mathbf{tell}(d) \\ A_2 &= \mathbf{tell}(c). \end{aligned}$$

We have $Quie(A_1) = Quie(A_2)$ whereas $\mathcal{O}(A_1)true = \{c, d\} \neq \mathcal{O}(A_2)true = \{c\}$.

For the finite observables things are easy: $\mathcal{O}_{\text{fin}}(A)true$ can be characterized compositionally as the smallest set which satisfies the equations in Table 5.

The formal justification as usual is given by showing that $\mathcal{O}(A)true$ is the least fixpoint of a mapping F associated to the equations. Interpretations are ordered, as before, by set inclusion. The proof can be constructed following the lines of Section 4.1, and we do not go into it. We only give the definition of F.

Definition 5.3 Let $\mathcal{I}$ now denote the set of interpretations assigning to each clp agent a set of constraints. The mapping $F : \mathcal{I} \rightarrow \mathcal{I}$, associated to the set of declarations D, is defined as follows:

$$\mathcal{O}_{\text{fin}}(\mathbf{stop})true = \{true\}$$

$$\mathcal{O}_{\text{fin}}(\mathbf{tell}(c))true = \{c\}$$

$$\mathcal{O}_{\text{fin}}(A + B)true = \mathcal{O}_{\text{fin}}(A)true \cup \mathcal{O}_{\text{fin}}(B)true$$

$$\mathcal{O}_{\text{fin}}(A \parallel B)true = \{c \sqcup d \mid \ c \in \mathcal{O}_{\text{fin}}(A)true,\ d \in \mathcal{O}_{\text{fin}}(B)true\}$$

$$\mathcal{O}_{\text{fin}}(\exists_x A)true = \{\exists_x c \mid \ c \in \mathcal{O}_{\text{fin}}(A)true\}$$

$$\mathcal{O}_{\text{fin}}(p(x))true = \mathcal{O}_{\text{fin}}(\Delta^x_y A)true \quad \text{where } p(y) \text{ :- } A \text{ is the declaration for } p \text{ in } D$$

Table 5: Denotational semantics for the finite observables of clp agents.

1. $F(I)(\mathbf{stop}) = \{true\}$,
2. $F(I)(\mathbf{tell}(c)) = \{c\}$,
3. $F(I)(A + B) = F(I)(A) \cup F(I)(B)$,
4. $F(I)(A \parallel B) = \{c \sqcup d \mid \ c \in F(I)(A),\ d \in F(I)(B)\}$,
5. $F(I)(\exists_x A) = \{\exists_x c \mid \ c \in F(I)(A)\}$,
6. $F(I)(p(x)) = I(\Delta^x_y A)$, where $p(y)$:- A is the declaration of p in D.

Let us now consider the problem of extending this model to a model which additionally includes the results of infinite behaviors.

5.1 Modeling the exact set of results: various attempts

A natural candidate to model the result of infinite computations would be the greatest fixpoint of F. But unfortunately, iterating from the set of all constraints only generates upward closed sets, thus identifying agents which do not produce anything and agents which have the possibility to establish some constraint.

Example 5.4 (Greatest fixpoint) Consider the agents $p(x)$ and $q(x)$ defined by the declarations

$$\begin{aligned} p(x) &\ \text{:-}\ \ p(x) + \mathbf{tell}(c) \\ q(x) &\ \text{:-}\ \ q(x). \end{aligned}$$

In the greatest fixpoint approach both the agents $p(x)$ and $q(x)$ will be associated to the set $\Uparrow\{true\}$. Note however that $\mathcal{O}(p(x))true = \{true, c\}$ and $\mathcal{O}(q(x))true = \{true\}$.

The above observation suggests us to look for a limit of a sequence of interpretations starting with an interpretation which assigns the set $\{true\}$ to each procedure call. The problem now arises with respect to which ordering we can define the limit. Simple set-inclusion does not generate a chain.

Example 5.5 Consider the agent $p(x)$ defined by the declaration

$$p(x) :\!-\ p(x) \parallel \mathbf{tell}(c).$$

If we start with assigning to $p(x)$ the set $\{true\}$, after the first iteration we obtain for $p(x)$ the set $\{c\}$.

So we have to look for an ordering $\leq$ on sets of constraints such that, for example, $\{true\} \leq \{c\}$. A natural solution would be to consider the *Smyth preordering* ([32]): $C \leq D$ iff for every $d \in D$ there exists $c \in C$ such that $c \sqsubseteq d$. This ordering however identifies a set with its upward closure, and as such it gives rise to the same identifications os the greatest fixpoint approach.

Example 5.6 (Smyth preordering) Consider the agents $p(x)$ and $q(x)$ in Example 5.4. If we apply F to the interpretation which assigns to each agent the set $\{true\}$ (the smallest set in the Smyth powerdomain), then after the first iteration we reach the fixpoint which consists of the sets $\{true, c\}$ for $p(x)$ and $\{true\}$ for $q(x)$. These sets are Smyth equivalent.

One could try other relations, like the Hoare preordering or the Egli-Milner preordering (see for example [26]). The latter is the most refined, as it corresponds to the intersection of the other two. However, next example shows that also the Egli-Milner preordering is not suitable for our purposes.

Example 5.7 (Egli-Milner preordering) Let $c \sqsubseteq d$. Consider the agents $p(x)$ and $q(x)$ defined by the declarations

$$\begin{array}{lll} p(x) & :\!- & p(x) + \mathbf{tell}(c) + \mathbf{tell}(d) \\ q(x) & :\!- & q(x) + \mathbf{tell}(d). \end{array}$$

Starting from the set $\{true\}$ (the smallest set in the Egli-Milner powerdomain), after the first iteration we reach the fixpoint which consists of the sets $\{true, c, d\}$ for $p(x)$ and $\{true, d\}$ for $q(x)$. These sets are Egli-Milner equivalent.

Instead of looking for an appropriate ordering we could try to define the limit in terms of an appropriate *metrics* on sets of constraints (for the general metric approach to the semantics of concurrent programming languages see [2]). However the example given below clearly shows that this approach does not work either.

Example 5.8 (Metrics) Consider the following declarations:

$$\begin{array}{lll} p(x) & :\!- & q(x) \\ q(x) & :\!- & p(x) + (r(x) \parallel \mathbf{tell}(c)) \\ r(x) & :\!- & \mathbf{tell}(\mathit{false}). \end{array}$$

Table 6 depicts the meaning of the calls $p(x)$, $q(x)$ and $r(x)$[10] after some iterations of the mapping F of Definition 5.3. For $4 \leq i$ we have that $I_i = I_{i+2}$ and $I_i \neq I_{i+1}$. Therefore the sequence is not converging with respect to any kind of metrics.

One might try to extract the "stable" informations from the sequence of interpretations; more precisely, to consider a notion of limit defined as follows:

$$lim_i(I_i)(A) = \{\sqcup_i c_i \mid c_i \in I_i(A) \text{ and } \{c_i\}_i \text{ is a chain}\}. \tag{6}$$

Unfortunately, also this approach fails:

[10] Note that $I_1(p'(x)) = I_0(\Delta^x_y A) = \{true\}$, for any call $p'(x)$, assuming the declaration $p'(y) :\!-\ A$. On the other hand $I_1(r(x) \parallel \mathbf{tell}(c)) = F(I_0)(r(x) \parallel \mathbf{tell}(c)) = \{c\}$.

	$p(x)$	$q(x)$	$r(x)$
I_0	$\{true\}$	$\{true\}$	$\{true\}$
I_1	$\{true\}$	$\{true\}$	$\{true\}$
I_2	$\{true\}$	$\{c, true\}$	$\{false\}$
I_3	$\{c, true\}$	$\{true, false\}$	$\{false\}$
I_4	$\{true, false\}$	$\{c, true, false\}$	$\{false\}$
I_5	$\{c, true, false\}$	$\{true, false\}$	$\{false\}$
I_6	$\{true, false\}$	$\{c, true, false\}$	$\{false\}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 6: Unapplicability of the metric approach.

Example 5.9 Enrich the declarations in Example 5.8 with the following:

$$poq(x) \quad :\!- \quad p(x) + q(x).$$

We have that, for all $i \geq 4$, $I_i(poq(x)) = \{true, c, false\}$. Thus the limit of $\{I_i\}_i$ according to (6) assigns to $poq(x)$ the set $\{true, c, false\}$. However only *true* and *false* are in the observables of $poq(x)$.

The problem with the definition (6) is that it consider also chains consisting of (partial) results which come from different branches. Encoding in some way the links among constraints belonging to the same branch could bring to the solution of our problem. A way to do so is to consider an appropriate *category*. The idea, inspired by the work of [23], is to consider objects $X, Y, \ldots$ which are (multi)sets of constraints and morphisms $f : X \rightarrow Y$ which are relations $f \subseteq X \times Y$ such that $\langle c, d\rangle \in f$ iff $c \sqsubseteq d$, and such that each element of Y is in relation with (at least) one element on X. This construction corresponds to the Lehmann's powerdomain ([17]). Roughly, the idea is that $Y = F(X)$ establishes a morphism $f : X \rightarrow Y$ where a pair $\langle c, d\rangle \in f$ represents the fact that c contributes to the generation of d in the application of F. For instance, in Table 6, in the column for $q(x)$, the constraint *false* in I_3 is related to c in I_2.

Note that a morphism $f : X \rightarrow Y$ exists if and only if X is smaller than Y according to the Smyth preordering; however the categorical approach gives more information and it allows to distinguish sets which are identified by the ordering. In fact $X \leq Y$ and $Y \leq X$ implies that X and Y are Smyth equivalent, but the corresponding morphisms $f : X \rightarrow Y$ and $g : Y \rightarrow X$ might not imply the isomorphism of X and Y in the category (g might not be the inverse of f).

In the next subsection we will show that a model for infinite behaviors of clp agents can be defined as the *colimit* of an ω-chain consisting of its finite approximations, represented by the iterations of a functor F starting from the initial object in the category.

5.2 The categorical approach

We recall the basic notions about category theory. For more details see for instance [24].

A *category* consists of a collection of *objects* $X, Y, \ldots$, *morphisms* $f : X \rightarrow Y$, and a composition operator which assigns to each pair of morphisms $f : X \rightarrow Y$, $g : Y \rightarrow Z$ a morphism $g \circ f : X \rightarrow Z$ satisfying:

Associativity. For any morphisms $f : X \rightarrow Y$, $g : Y \rightarrow Z$, $h : Z \rightarrow U$, the equality $h \circ (g \circ f) = (h \circ g) \circ f$ holds.

Identity. For each object X there exists a morphism $ID_X : X \rightarrow X$ such that, for any morphism $f : X \rightarrow Y$, $ID_Y \circ f = f$ and $f \circ ID_X = f$ holds.

For the semantic construction below we introduce here the definitions of the notions of a *functor* and the notions of a *cocone* and *colimit* for a diagram $\mathbf{D}$ of a given category: A functor from a category $\mathbf{A}$ into a category $\mathbf{B}$ is a mapping assigning to each object X of $\mathbf{A}$ an object $F(X)$ of $\mathbf{B}$ and to each morphism $f : X \rightarrow Y$ in $\mathbf{A}$ a morphism $F(f) : F(X) \rightarrow F(Y)$ in $\mathbf{B}$ such that for any two morphisms $f : X \rightarrow Y$, $g : Y \rightarrow Z$ in $\mathbf{A}$

1. $F(ID_X) = ID_{F(X)}$
2. $F(g \circ f) = F(g) \circ F(f)$.

A cocone for a diagram $\mathbf{D}$ of a given category is an object X and a collection of morphisms $f_i : D_i \rightarrow X$ for each D_i of $\mathbf{D}$ such that for each morphism $g : D_i \rightarrow D_j$ in $\mathbf{D}$ we have $f_j \circ g = f_i$. A colimit for $\mathbf{D}$ then is a cocone $\{f_i : D_i \rightarrow X\}$ with the property that for any cocone $\{f'_i : D_i \rightarrow X'\}$ there is a unique morphism $g : X \rightarrow X'$ such that $f'_i = g \circ f_i$.

We intend to apply Lehmann's powerdomain construction ([17]) which consists, given a complete partially ordered set $(Con, \sqsubseteq)$ of the following construction of the categorical powerdomain $Cat(Con)$. Objects of $Cat(Con)$ are multisets of Con, where an additional mechanism for identifying different occurrences of an element of a multiset is assumed. A morphism $f : X \rightarrow Y$ is a relation $f \subseteq X \times Y$ such that whenever $\langle c, d\rangle \in f$ then $c \sqsubseteq d$, and for every $d \in Y$ there exists $c \in X$ such that $\langle c, d\rangle \in f$. The composition of two morphisms $f : X \rightarrow Y$ and

$g : Y \to Z$ is defined by: $\langle c, e\rangle \in g \circ f$ iff there exists a $d \in Y$ such that $\langle c, d\rangle \in f$ and $\langle d, e\rangle \in g$. For a ω-chain in $Cat(Con)$

$$X_0 \xrightarrow{f_0} X_1 \xrightarrow{f_1} X_2 \ldots$$

the colimit is given by the multiset Y consisting of the elements $\sqcup_i c_i$, where $c_i \in X_i$ and $\langle c_i, c_{i+1}\rangle \in f_i$, together with the morphisms $g_k : X_k \to Y$ such that $\langle c, d\rangle \in g_k$ iff $d = \sqcup_i c_i$ with $\langle c_i, c_{i+1}\rangle \in f_i$ and $c = c_k$.

For the construction of a denotational model for clp we introduce the following instantiation of the above scheme. Given a cylindrical constraint system $\mathbf{C}$ we denote by $\mathcal{P}^{\mathrm{m}}_{\mathrm{ne}}(Con)$ the set of (non-empty) multisets of constraints.

Definition 5.10 Given a cylindrical constraint system $\mathbf{C}$, let $\mathcal{I}$ denote the category of *interpretations* $I : Agents \to \mathcal{P}^{\mathrm{m}}_{\mathrm{ne}}(Con)$ with morphisms $f : I \to J$ such that, for every agent A, $f(A) : I(A) \to J(A)$ is a morphism in $Cat(\mathbf{C})$.

The composition of two morphisms $f : I \to J$ and $g : J \to K$ is given by: $f \circ g(A) = f(A) \circ g(A)$, for every agent A.

We want to define the denotational model as the *colimit* of a ω-chain in $\mathcal{I}$ consisting of its finite approximations. These finite approximations are obtained by iteratively applying a functor F to the initial object I_0 of the category $\mathcal{I}$ which assigns to each agent the multiset $\{true\}$. (An object X of a category is called initial if, for every object Y, there exists exactly one morphism from X to Y.) The functor F is defined as follows:

Definition 5.11 Given a set of declarations D we define a functor $F : \mathcal{I} \to \mathcal{I}$ as follows:

1. $F(I)(\mathbf{tell}(c)) = \{c\}$,
2. $F(I)(A + B) = F(I)(A) \uplus F(I)(B)$,
3. $F(I)(A \parallel B) = F(I)(A) \sqcup F(I)(B)$,
4. $F(I)(\exists_x A) = \exists_x(F(I)(A))$,
5. $F(I)(p(x)) = I(\Delta^x_y A)$.

In the second clause $\uplus$ denotes multiset union. The operations $\sqcup : \mathcal{P}^{\mathrm{m}}_{\mathrm{ne}}(Con) \times \mathcal{P}^{\mathrm{m}}_{\mathrm{ne}}(Con) \to \mathcal{P}^{\mathrm{m}}_{\mathrm{ne}}(Con)$ and $\exists_x : \mathcal{P}^{\mathrm{m}}_{\mathrm{ne}}(Con) \to \mathcal{P}^{\mathrm{m}}_{\mathrm{ne}}(Con)$ are defined as the natural pointwise extensions of their corresponding operations on constraints:

$$X \sqcup Y = \{c \sqcup d :|\ c \in X,\ d \in Y\}$$

and

$$\exists_x(X) = \{\exists_x c \mid\ c \in X\}.$$

In the last clause we assume D to contain the declaration $p(y)$:- A. Analogously we define $F(f) : F(I) \to F(J)$ for every morphism $f : I \to J$:

1. $F(f)(\mathbf{tell}(c)) = \{\langle c, c\rangle\}$,
2. $F(f)(A + B) = F(f)(A) \uplus F(f)(B)$,
3. $F(f)(A \parallel B) = F(f)(A) \sqcup F(f)(B)$,
4. $F(f)(\exists_x A) = \exists_x(F(f)(A))$,
5. $F(f)(p(x)) = f(\Delta^x_y A)$.

where the operations $\sqcup$ and $\exists_x$ denote now the pointwise extensions of the corresponding operations, first to pairs of constraints, and then to sets of pairs of constraints. Again in the last clause D is assumed to contain the declaration $p(y)$:- A.

We leave it to the reader to check that F is well-defined, that is, $F(f)$ is a morphism, and that F indeed is a functor, that is, $F(f \circ g) = F(f) \circ F(g)$, for any morphisms f and g, and $F(ID_I) = ID_{F(I)}$ (note that $ID_I(A) = \{\langle c, c\rangle \mid c \in I(A)\}$, for any interpretation I and agent A).

Consider next the following ω-chain in $\mathcal{I}$ obtained by iteratively applying F to the initial object I_0:

$$I_0 \xrightarrow{f_0} F(I_0) \xrightarrow{F(f_0)} F(F(I_0)) \ldots$$

Formally, we define a sequence of interpretations I_j, with I_0 the initial object and $I_{j+1} = F(I_j)$, and morphisms $f_j : I_j \to I_{j+1}$, with f_0 the unique morphism from I_0 to I_1 and $f_{j+1} = F(f_j)$.

Let I_ω denote the colimit of the above ω-chain. It can be defined as follows:

Definition 5.12 $I_\omega(A) = \{\bigsqcup_i c_i \mid \langle c_i, c_{i+1}\rangle \in f_i(A)\}$.

It should be noted that I_ω is a multiset, that is, for any sequence $(c_i)_i$ of constraints such that $\langle c_i, c_{i+1}\rangle \in f_i(A)$ there is a corresponding occurrence of $\bigsqcup_i c_i$ in I_ω. It follows from the general Lehmann's powerdomain construction that I_ω is the colimit of the ω-chain defined above, together with the following morphisms $g_k : I_k \to I_\omega$: $\langle c, d\rangle \in g_k(A)$ iff there exists $c_i \in I_i(A)$ such that $d = \bigsqcup_i c_i$, $\langle c_i, c_{i+1}\rangle \in f_i(A)$, and $c = c_k$.

Next we establish the precise relationship between the denotational model I_ω and the observables. We introduce for each agent A the set $Comp^i(A)$ which consists of all (maximal prefixes of) computations of $Comp(A, true)$ with a maximal number of *nested* body replacements *less* than i. We introduce the convention that for every agent A $Comp^0(A) = \{\langle A, true\rangle\}$. Thus, for example, $Comp^1(p(x)) = \{\langle p(x), true\rangle\}$, $Comp^{i+1}(p(x)) = \langle p(x), true\rangle \circ Comp^i(\Delta^x_y A)$, for $i > 0$, assuming the declaration $p(y)$:- A. To prove the correctness and completeness of the model I_ω it then suffices to prove for each agent A and $i \geq 0$ the existence of bijections $\pi_i(A)$ which map each (occurrence of a) constraint c of $I_i(A)$ to a computation of $Comp^i(A)$ resulting in c such that, additionally, for every two constraints c and d: $\langle c, d\rangle \in f_i(A)$ iff $\pi_i(A)(c)$ is a prefix of $\pi_{i+1}(A)(d)$.

The above is summarized by the following proposition:

Proposition 5.13 *For each agent A and $i > 0$ there exist bijections $\pi_i(A) : I_i(A) \to Comp^i(A)$ such that the final store of $\pi_i(A)(c)$ equals c, and for every two constraints c and d we have that $\langle c, d\rangle \in f_i(A)$ iff $\pi_i(A)(c)$ is a prefix of $\pi_{i+1}(A)(d)$.*

Proof The proof proceeds by a simultaneous induction on A and $i > 0$.

$A = \mathbf{tell}(c)$. It suffices to note that $Comp^i(A) = \{\langle \mathbf{tell}(c), true\rangle \to \langle \mathbf{stop}, c\rangle\}$.

$A = A_1 + A_2$. Since $I_i(A_1 + A_2) = I_i(A_1) \uplus I_i(A_2)$, $f_i(A_1 + A_2) = f_i(A_1) \uplus f_i(A_2)$, and $Comp^i(A_1 + A_2) = Comp^i(A_1) \cup Comp^i(A_2)$, this case follows immediately from the induction hypothesis for A_1 and A_2.

$A = A_1 \parallel A_2$. Note that by the fairness assumption there exists a bijection between $Comp^i(A_1) \times Comp^i(A_2)$ and $Comp^i(A)$. Furthermore we have that $I_i(A_1 \parallel A_2) = \{c \sqcup d \mid c \in I_i(A_1),\ d \in I_i(A_2)\}$ and $\langle c, d\rangle \in f_i(A)$ iff there exist constraints c_1, c_2 and d_1, d_2 such that $c = c_1 \sqcup c_2$, $d = d_1 \sqcup d_2$, $\langle c_1, d_1\rangle \in f_i(A_1)$, and $\langle c_2, d_2\rangle \in f_i(A_2)$. Thus also this case follows in a straightforward manner from the induction hypothesis for A_1 and A_2.

$A = \exists\ B$. Since $I_i(\exists_x B) = \exists_x(I_i(B))$ and $f_i(\exists_x B) = \exists_x(f_i(B))$, it suffices to observe that $Comp^i(A) = \exists_x(Comp^i(B))$, where $\exists_x$ is the natural extension to sets of the operation $\exists_x(\xi)$ which consists of applying $\exists_x$ to every store of ξ. Thus a simple application of the induction hypothesis for B settles this case.

$A = p(x)$. Since $I_i(p(x)) = I_{i-1}(\Delta^x_y B)$ and $f_i(p(x)) = f_{i-1}(\Delta^x_y B)$, it suffices to observe that $Comp^i(p(x)) = \langle p(x), true\rangle \circ Comp^{i-1}(\Delta^x_y B))$. Thus this case follows ina straightforward manner from the induction hypothesis for $i - 1$. □

Theorem 5.14 (Correspondence between I_ω and the observables) *For each agent A, $I_\omega(A)$ is the multiset of constraints consisting of the limit of each path in $Comp(A, true)$.*

Proof By the above proposition there exists a one-to-one correspondence between all sequences of constraints $(c_i)_i$, such that $\langle c_i, c_{i+1}\rangle \in f_i(A)$ and $Comp(A, true)$. □

6 Conclusions

Deterministic ccp has a very simple model based on closure operators which allows to retrieve the results of finite and infinite computations from the quiescent points of an agents. In this work we have shown that, if we restrict to observe the upward-closure of the results, the model can be smoothly extended to a nondeterministic version of ccp which allows local choice and mutually-exclusive choice.

When one whishes to observe the exact set of results, however, then the presence of nondeterminism complicate things considerably with respect to infinite computations. We have shown that standard set-theoretic, order-theoretic and metric approaches all fail even in the simple case of clp (ccp without synchronization), and we have proposed a solution based on category theory.

To our opinion these results, both the negative and the positive ones, do not depend on the specific nature of ccp being based on computing constraints. They should apply as well to generic nondeterministic languages working on arbitrary data domains, provided that the partial results of computations evolve monotonically and that their limit is defined.

Acknowledgements

We wish to thank Marta Kwiatkowska and Eugenio Moggi for helpful discussions. Furthermore, we are grateful to Paul Gastin, Maurice Nivat and Jan Rutten for the organization of the MASK workshop on *Topology and Completion in semantics*, which stimulated us to structure the material and to write this paper. We want also to thank all the participants to that workshop, who created a pleasant and stimulating scientific atmosphere. Finally, we want to thank the referees who's comments improved the paper considerably.

References

[1] K. R. Apt, H. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, Los Altos, Ca., 1988.

[2] J.W. de Bakker and J.I. Zucker Processes and the denotational semantics of concurrency. Information and Control, volume 54, pages 70–120, 1982.

[3] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. Semantic models for Concurrent Logic Languages. *Theoretical Computer Science*, 86(1), 3–33, 1991.

[4] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. A paradigm for asynchronous communication and its application to concurrent constraint programming. In *Logic Programming Languages: Constraints, Functions, and Objects*, Series in Logic Programming, pages 82–114. The MIT Press, 1993.

[5] F.S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving Concurrent Constraint Programs Correct. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages*, 1993.

[6] F.S. de Boer and C. Palamidessi. A fully abstract model for concurrent constraint programming. In S. Abramsky and T.S.E. Maibaum, editors, *Proc. of TAPSOFT/CAAP*, volume 493 of *Lecture Notes in Computer Science*, pages 296–319. Springer-Verlag, 1991.

[7] F.S. de Boer and C. Palamidessi. On the semantics of concurrent constraint programming. Invited paper in *Proc. of ALPUK 92, Workshops in Computing*, pages 145–173, Springer-Verlag, 1992.

[8] K.L. Clark. Predicate logic as a computational formalism. Res. Report DOC 79/59, Imperial College, Dept. of Computing, London, 1979.

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

[10] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[11] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence and Concurrent Constraint Programming. Technical Report, Dipartimento di Informatica, Università di Pisa, Pisa. 1994.

[12] W.G. Golson. Toward a declarative semantics for infinite objects in logic programming. *Journal of Logic Programming*, 5:151–164, 1988.

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

[14] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119, 1987.

[15] J. Jaffar and M.J. Maher Constraint Logic Programming: A survey Technical report, IBM Thomas J. Watson Research Center, Yorktown Heights, NY. 1994. To appear in a special issue of the *Journal of Logic Programming*.

[16] R. Jagadeesan, V.A. Saraswat, and V. Shanbhogue. Angelic non-determinism in concurrent constraint programming. Technical report, Xerox Park, 1991.

[17] D.J. Lehmann. Categories for Fixed-point Semantics. PhD thesis, Hebrew University of Jerusalem, 1976.

[18] G. Levi and C. Palamidessi. Contributions to the semantics of logic perpetual processes. *Acta Informatica*, 25(6):691–711, 1988.

[19] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987. Second edition.

[20] M. Z. Kwiatkowska. Infinite Behavior and Fairness in Concurrent Constraint Programming. In J. W. de Bakker, W. P.de Roever, and G. Rozenberg, editors, *Semantics: Foundations and Applications*, volume 666 of *Lecture Notes in Computer Science*, pages 348–383, Beekbergen The Netherlands, June 1992. REX Workshop, Springer-Verlag, Berlin.

[21] M. J. Maher. Logic semantics for a class of committed-choice programs. In Jean-Louis Lassez, editor, *Proc. of the Fourth International Conference on Logic Programming*, Series in Logic Programming, pages 858–876, Melbourne, 1987. The MIT Press.

[22] M.A. Nait Abdallah. On the interpretation of infinite computations in logic programming. In J. Paredaens, editor, *Proc. of Automata, Languages and Programming*, volume 172, pages 374–381. Springer Verlag, 1984.

[23] S. Nystrom and B. Jonsson. Indeterminate concurrent constraint programming: A fixpoint semantics for non-terminating computations. In D. Miller, editor, *Proc. International Symposium on Logic Programming'93*, pages 335–352. The MIT Press, 1993.

[24] B.C. Pierce. Basic Category Theory for Computer Scientists. Foundations of Computing Series, 1991.

[25] G. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[26] G. Plotkin. Domains. Department of Computer Science, University of Edimburgh, 1992. Post-graduate lecture notes in advanced domain theory (incorporating the 'Pisa notes' 1981).

[27] V.A.Saraswat. *Concurrent Constraint Programming*. PhD thesis, Carnegie-Mellon University.1989. In ACM distinguished dissertation series. The MIT Press, 1993.

[28] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. Seventeenth Annual ACM Symp. on Principles of Programming Languages*, pages 232–245, New York, 1990.

[29] V.A. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of concurrent constraint programming. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages*, New York, 1991.

[30] D. Scott. Domains for denotational semantics. In *Proc. of ICALP*, 1982.

[31] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.

[32] M.B. Smyth. Powerdomains. Journal of Computer and System Sciences, 16:23-36, 1978.