ECOLE NORMALE SUPERIEURE

# Expressiveness and Complexity of Concurrent Constraint Programming: a Finite Model Theoretic Approach

François FAGES
Sylvain SOLIMAN
Victor VIANU

LIENS - 98 - 14

Département de Mathématiques et Informatique

CNRS URA 1327

# Expressiveness and Complexity of Concurrent Constraint Programming: a Finite Model Theoretic Approach

**François FAGES**
**Sylvain SOLIMAN**
**Victor VIANU***

**LIENS - 98 - 14**

December 1998

Laboratoire d'Informatique de l'Ecole Normale Supérieure
45 rue d'Ulm 75230 PARIS Cedex 05

Tel : (33)(1) 01 44 32 30 00

Adresse électronique : fages@dmi.ens.fr , soliman@dmi.ens.fr

* University of California
San Diego, U.S.A.

Adresse électronique : vianu@cs.ucsd.edu

# Expressiveness and Complexity of Concurrent Constraint Programming: a Finite Model Theoretic Approach

François Fages* Sylvain Soliman* Victor Vianu†

**Abstract**

*We study the expressiveness and complexity of concurrent constraint programming languages over finite domains. We establish strong connections between these languages and query languages in finite model theory. The bridge to finite model theory yields new (and sometimes quite surprising) results on the expressiveness and complexity of concurrent constraint languages, including several powerful normal forms. These results provide new insight into the impact of various semantics and features of concurrent constraint programming languages on their expressiveness and complexity.*

## 1 Introduction

Concurrent constraint programming (CC) [Sar87] emerged as a combination of concurrent logic programming and constraint logic programming (CLP) [JL87]. The basic idea is the one of a set of agents communicating through a shared store of constraints, that are logical formulas on the unknowns of the problem. Each agent can either write a constraint to the store (tell operation), or synchronize with other agents by suspending its execution until the store entails some constraint (ask operation). The ask operation is the main addition of CC to CLP languages, and generalizes the co-routining facilities of CLP languages, like the `freeze` or `delay` predicates of Prolog.

The class CC($\mathcal{S}$) of concurrent constraint languages is parameterized by the constraint system $\mathcal{S}$ which defines the logical interpretation of the constraints. Of particular interest is the class CC($\mathcal{FD}$) of concurrent constraint languages over finite domains, which is widely used in the area of search and combinatorial optimization problems [JM94]. In CC($\mathcal{FD}$), concurrency primitives are used to express constraint propagation algorithms and complex resolution methods. Deep guards are a generalization of the ask operation where the condition, the guard, can be an arbitrary agent instead of a constraint. Deep guards are heavily used in the AKL and Oz implementations [SSW87] to express encapsulated search, branch&bound optimization and complex search procedures.

The expressiveness and complexity of concurrent programming languages, particularly relevant to the family of CC languages, are largely unexplored. In particular, it is of interest to understand the impact on expressiveness and complexity of various semantics and features, such as:

- the choice of observables (success or terminal store).
- the satisfaction-completeness assumption on the constraint system (complete or open),
- the type of guards (flat or deep).

In this paper we study the complexity and expressiveness of variants of CC($\mathcal{FD}$) with different features and semantics, seen as query languages over finite relational structures. We consider CC($\mathcal{FD}$) languages parameterized along the three dimensions mentioned above: the choice of observables (success vs. terminal store), the satisfaction completeness assumption (complete vs. open), and the type of guards (no guards, flat, deep). We begin by establishing connections with query languages in finite model theory, such as FO, fixpoint logics, and Datalog$^\neg$. The connections are interesting in their own right, as they shed new light on the significance of the CC($\mathcal{FD}$) parameters from the viewpoint of finite model theory. For example, the complete constraint systems generally yield inflationary query languages[1] whose complexity is within P, and open constraint systems yield languages with a powerful combination of negation and nondeterminism allowing to express higher

*Ecole Normale Supérieure, 45, rue d'Ulm, 75005 Paris, {fages,soliman}@ens.fr

†U.C.San Diego, vianu@cs.ucsd.edu. Work done in part while visiting ENS.

[1] Intuitively, the term *inflationary* refers to the fact that space used throughout query evaluation is nondecreasing.

complexity classes. A variant of CC based on linear logic corresponds to a non-inflationary language. Such distinctions are important and well understood in the theory of query languages[2]. As another example, guards in CC are closely related to negation in query languages, and the formal semantics we provide for deep guards with terminal store observable is inspired by the well-founded semantics for Datalog with negation.

More concretely, the benefits of the bridge to finite model theory are numerous: building upon results in finite model theory, we are able to obtain new results on the relative expressiveness and complexity of the CC($\mathcal{FD}$) languages. For example, variants of CC($\mathcal{FD}$) expressing NP, PHIER, and PSPACE are shown. In the presence of order, other fragments capture P. As an important side effect, we obtain several powerful normal forms for CC($\mathcal{FD}$) languages. For example, we show that CC($\mathcal{FD}$) with deep guards collapses to CC($\mathcal{FD}$) with flat guards on ordered domains. The complexity results in the paper are summarized in Figure 1 of the concluding section. Such results provide new insight into the impact of various semantics and features of concurrent constraint programming languages on their expressiveness and complexity on finite domains.

The paper is organized as follows. Preliminaries on finite model theory, and basic definitions for concurrent constraint programming are presented in Section 2. CC($\mathcal{FD}$) languages with success semantics are studied in Section 3, and Section 4 focuses on languages with terminal store semantics. The conclusion contains a summary of our results and discusses the impact of various CC($\mathcal{FD}$) features and semantics on expressiveness and complexity. Due to space limitations, many details are omitted. An Appendix contains additional preliminaries, the formal specification of CC($\mathcal{FD}$) semantics, and several proof sketches.

# 2 Preliminaries

We briefly review basic results from finite model theory used in the paper. Next, we provide basic definitions of CC languages and semantics.

## 2.1 Finite Model Theory

Finite-model theory focuses on finite relational structures and logic languages defining properties of such structures or mappings among relational structures (usually called *queries*). We recall several languages that play a central role in finite-model theory. FO denotes first-order logic over relational vocabulary. Codd's well-known algebraization of FO and the fact that FO is in (uniform) $AC_0$ [Imm87b] (and, thus, in a reasonable sense, takes constant parallel time) explain the appeal of FO as a database query language. However, FO has well-known limitations in expressive power: for example, it cannot express the connectivity of a graph [Fag75, AU79]. To overcome such shortcomings, various extensions of FO with recursion have been proposed. Although widely varying paradigms can be used to extend FO (e.g. procedural, logic, or logic programming), most languages based on the various paradigms are equivalent to either the *fixpoint* or the *while* queries [CH82]. Some of these languages, used in the paper, are reviewed in the Appendix. They include the fixpoint logics IFP, LFP (expressing *fixpoint*) and PFP (expressing *while*), and languages inspired by Prolog such as Datalog and Datalog$^\neg$ with stratified and well-founded semantics.

The expressive power of a logic language is measured in finite-model theory by the set of queries it can express. Two languages are equivalent if they express the same set of queries. A language $L$ subsumes $L'$ if each query expressible in $L'$ is also expressible in $L$. Among the fixpoint logics reviewed in the Appendix, PFP subsumes IFP and the equivalent LFP, which in turn strictly subsume FO. Whether IFP (LFP) and PFP are equivalent (i.e. whether *fixpoint* = *while*) is an open question, shown to be equivalent to another open problem in complexity theory, whether P = PSPACE [AV95]. Among Datalog$^{(\neg)}$ languages, Datalog$^\neg$ (with inflationary or well-founded semantics) strictly subsumes stratifiable Datalog$^\neg$, which strictly subsumes semi-positive Datalog$^\neg$, which in turn strictly subsumes Datalog. It is easy to see that FO is equivalent to non-recursive stratifiable Datalog$^\neg$. More surprisingly, IFP is equivalent to inflationary Datalog$^\neg$, which can be viewed as a syntactically much simpler fragment of IFP (existential and without nesting of fixpoint operators). This yields a useful normal form for IFP queries.

The *computational complexity* of a problem is the amount of resources, such as time or space, required by a machine that solves the problem. Complexity theory traditionally has focused on the computational complexity of problems. A more recent branch of complexity theory, started by Fagin in [Fag74, Fag75] and developed during the 1980s, focuses on the *descriptive complexity* of problems, which is the com-

[2] See [AHV95] for a discussion of inflationary and noninflationary query languages.

plexity of describing problems in some logical formalism [Imm87a]. One of the exciting developments in complexity theory is the discovery of a very intimate connection between computational and descriptive complexity. This intimate connection was first discovered by Fagin, who showed that the properties of complexity NP are precisely those expressible in existential 2nd-order logic [Fag74] (cf. [JS74]). This was later generalized by showing that full second-order logic expresses PHIER, the polynomial hierarchy [S77]. Another demonstration of this connection was shown by Immerman and Vardi, who discovered that IFP expresses precisely the set of queries of complexity P on structures equipped with a total order (henceforth called *ordered structures*) [Imm86, Var82]. Similarly, PFP expresses precisely the queries of complexity PSPACE on ordered structures [Var82]. Note that the order assumption is crucial: in the absence of order, both IFP and PFP fail to express "simple" queries such as parity (i.e. whether the size of a given unary relation is even or odd). Among the Datalog languages, semi-positive Datalog$^\neg$ expresses P on ordered structures equipped *min* and *max*, and stratifiable Datalog$^\neg$ and Datalog$^\neg$ (with inflationary or well-founded semantics) express P on ordered structures.

## 2.2 Concurrent constraint programming

We assume a first-order language over a countably infinite set $V$ of variables, a set $\Sigma$ of function and relation symbols, and the usual logical connectives. The set of variables (resp. free variables) occurring in a formula $A$ is written $v(A)$ (resp. $fv(A)$). Lists of variables (resp. terms) are written $\vec{x}$ (resp. $\vec{t}$), $\phi[\vec{t}/\vec{x}]$ denotes the formula $\phi$ in which variables $\vec{x}$ have been replaced by terms $\vec{t}$.

**Constraint systems** A *constraint language* $\mathcal{C}$ is a fragment of first-order logic containing $True$, $False$ and closed under renaming, conjunction and existential quantification. A *constraint system* $\mathcal{S}$ is given by a constraint language $\mathcal{C}$ and a first-order logical theory, i.e. a recursive set of axioms over the signature. We will only consider constraint systems for which entailment and consistency are decidable. That is, given a constraint system $\mathcal{S}$ with constraint language $\mathcal{C}$, it is decidable whether[3] $\mathcal{S}, c \vdash d$ where $c, d$ are constraints in $\mathcal{C}$. In particular constraint consistency, i.e. whether $\mathcal{S}, c \vdash False$, is decidable. When it is clear form the context we shall omit $\mathcal{S}$ in the entailment relation.

An essential distinguishing characteristic among constraint systems is the *satisfaction-completeness property* introduced in [JL87] for CLP languages:

**Definition 2.1** *A constraint system $\mathcal{S}$ is* satisfaction-complete *if for any constraint $c \in \mathcal{C}$, either $\mathcal{S} \vdash \exists(c)$ or $\mathcal{S} \vdash \neg\exists(c)$.*

Note that in a satisfaction-complete constraint system, a constraint $c$ is satisfiable, $\mathcal{S} \vdash \exists(c)$, iff $c$ is realizable in *any* model $\mathcal{M}$ of $\mathcal{S}$, $\mathcal{M} \models \exists(c)$ (all models of a satisfaction-complete constraint system are equivalent with respect to constraint realizability). Therefore one can see a satisfaction-complete constraint system as a description of constraint realizability in a fixed interpretation.

We shall consider both cases of satisfaction-complete constraint systems (*complete* in short) and *open constraint systems* which do not fix the interpretation of all relation symbols.

**$\mathcal{FD}$ constraint systems** In this paper we shall be concerned with constraint systems over finite domains $\mathcal{FD}$. In this case we assume a finite alphabet of constants $\{a_1, ..., a_n\}$, the equality predicate $=$, a finite set of relation symbols $r_1, ..., r_k$ and an infinite alphabet of variables.

Unless specified otherwise, the $\mathcal{FD}$ constraint language we consider is the least set containing the atomic constraints formed over $=$, $r$, and $\neg r$, closed by conjunction and existential quantification.

The $\mathcal{FD}$ constraint systems we consider are presented with the following strong equality axioms i)-iv):

i) $\forall x \; x = x$

ii) $\forall x, y, \vec{z} \; x = y \wedge r(x, \vec{z}) \Rightarrow r(y, \vec{z})$ for every relation symbol $r$

iii) $\forall x, y, z \; x = y \wedge y = z \Rightarrow x = z$

iv) $a_i \neq a_j$ for $i \neq j$

the domain closure axiom v):

v) $\forall x \; x = a_1 \vee ... \vee x = a_n$

plus axioms specifying (completely or partially) the truth tables of the relation symbols:

vi) $r_p(a_i, a_j), ..., \neg r_q(a_k, a_l), ...$

Note that the axioms i)-iv) are the same for all $\mathcal{FD}$ constraint systems. Axiom v) specifies a finite domain. The axioms in vi) specify completely or partially a finite relational structure over that domain. In the first case the constraint system is complete; otherwise, it is open. Indeed, it is easy to see that an $\mathcal{FD}$ constraint system is complete iff for each pair of

[3] By slight abuse of notation, we write $\mathcal{S} \vdash c$ to mean that $c$ is provable from the axioms of $\mathcal{S}$ by logical axioms and rules (provability is independent of $\mathcal{C}$).

constants $a_i, a_j$ in the finite domain, either $r(a_i, a_j)$ or $\neg r(a_i, a_j)$ is an axiom.

We shall consider also totally ordered $\mathcal{FD}_{\leq}$ constraint systems formed with the binary predicate $\leq$, and possibly unary predicates *min* and *max* specifying the least and greatest constants.

Note that one can also introduce function symbols, together with equality axioms with constants, so that the intended domain of interpretation is still finite and axiom v) is preserved. One way to achieve this is to translate a function symbol $f(x, y)$ as a relation $r_f(z, x, y)$ standing for $z = f(x, y)$, and presented with axioms of the form in vi).

**CC($\mathcal{FD}$) programs** We next present the syntax and semantics of CC($\mathcal{FD}$) programs. Informally, a program consists of a set of declarations defining procedures by agents and a call to one initial agent or procedure. We first define agents, then declarations, and finally programs. We assume fixed an $\mathcal{FD}$ constraint system $\mathcal{S}$ with constraint language $\mathcal{C}$. Agents are denoted by $A, B, \ldots$ The syntax of agents is given by the following grammar:

$$A ::= p(\vec{x}) \mid tell(c) \mid A \parallel A \mid A + A \mid \exists x A \mid \forall \vec{x}(A \rightarrow A)$$

where:

- $\vec{x}$ is a list of distinct variables,
- $c$ is a constraint in $\mathcal{C}$,
- $\parallel$ stands for parallel composition,
- $+$ represents nondeterministic choice,
- $\exists$ stands for variable hiding,
- $\rightarrow$ denotes blocking ask.

In an ask agent, the agent at the left of the arrow is called *the guard*. The ask agents are written with an universal quantifier to indicate the variables which are bound in the guard (this is often implicit in CC notations [Sar87]). *Deep guards* allow any agent to be in a guard. *Flat guards* allow only constraints in the guards: the guarding agent is a tell agent, as in $tell(c) \rightarrow A$ which, as expected, reduces to $A$ if the store entails $c$ and suspends otherwise (see the operational semantics below). The atomic agents $p(\vec{x}) \ldots$ are called *process names* or *procedure names*. Note that, formally, atomic agents are always written with distinct variable arguments, the equalities between arguments are expressed by an accompanying set of constraints. By abuse of notation however, we will write $p(a)$ for $\exists x\ (tell(x = a) \parallel p(x))$.

The syntax of *declarations* $\mathcal{D}$ is given by the following grammar:

$$D ::= p(\vec{x}) :: A$$

$$\mathcal{D} ::= \epsilon \mid [D]_{\sim}, \mathcal{D}$$

where $A$ is an agent, and $[D]_{\sim}$ denotes the congruence class of declaration $D$ up to variable renaming. As usual, we assume that in a declaration $D = p(\vec{x}) :: A$, all the free variables occurring in $A$ occur in the head $\vec{x}$. $[D]_{\sim}$ is thus the class of $D$ up to the renaming of the variables in the head of $D$. A *program* $\mathcal{D}.A$ is a list of declarations $\mathcal{D}$ together with an agent $A$.

**Operational semantics** The operational semantics of CC($\mathcal{FD}$) programs is defined on configurations (rather than agents). A *configuration* is a triple $(\vec{x}; c; \Gamma)$, where $\vec{x}$ is a set of *hidden* (existentially quantified) variables, $c$ is a constraint (the store), and $\Gamma$ is a list of agents. $\mathcal{K}$ will denote the set of configurations, and $\overline{Set}$ the complement in $\mathcal{K}$ of a set of configurations $Set$. The *store* of a configuration $(\vec{x}; c; \Gamma)$ is the constraint $\exists \vec{x} c$.

The operational semantics is defined using a transition system which does not take into account specific evaluation strategies. We distinguish a congruence relation $\equiv$ (between configurations and expressions) from the transition relation $\longrightarrow$ (between configuration congruence classes[4]). We will consider two possible ways for a computation to succeed from a given store:

(i) by reaching a *success configuration*, that is a configuration where no agents are pending and whose store is consistent; and

(ii) by reaching a *terminal configuration*, that is a configuration from which no further transitions can be made although there may be pending agents, and whose store is again consistent.

We will consider first the success semantics (i), which is the classical semantics for CC in presence of deep guards. Then, motivated by computational limitations of the success semantics, we will consider the terminal store semantics (ii). Note that the transition relation among configurations will have to be defined differently for (ii), in order to make the condition of success of a deep guard consistent with the terminal store semantics.

[4] By abuse of notation, we shall keep the same notations for the configurations and their congruence classes modulo $\equiv$.

**Queries defined by CC($\mathcal{FD}$) programs** The focus of this paper is on the ability of CC($\mathcal{FD}$) programs to define properties and queries on finite structures. This will allow us to compare various classes of CC($\mathcal{FD}$) programs among each other and with logic languages from finite model theory. It will also allow us to evaluate classes of CC($\mathcal{FD}$) programs in terms of the complexity of properties or queries they define.

Viewing CC($\mathcal{FD}$) programs as query defining mechanisms is quite natural [KG94]. As discussed earlier, complete $\mathcal{FD}$ constraint systems completely specify a finite structure over a relational vocabulary. Let $\mathbf{I}$ be a finite structure over fixed relational vocabulary $\sigma$ and let $\mathcal{S}_\mathbf{I}$ denote the complete $\mathcal{FD}$ constraint system with relational vocabulary $\sigma$ defining $\mathbf{I}$. When dealing with open constraint systems, we shall assume that the constraint system $\mathcal{S}$ is a conservative extension of $S_\mathbf{I}$. Let $\mathcal{D}.A$ be a CC($\mathcal{FD}$) program, where $fv(A) = x_1 \ldots x_k$. The query defined by $\mathcal{D}.A$ and $\mathcal{S}$ is the mapping associating to each finite structure $\mathbf{I}$ over $\sigma$ the $k$-ary relation consisting of all tuples $\langle a_1, \ldots a_k \rangle$ for which there exists a terminal or success store[5] $d$ for $\mathcal{D}.A$ with initial store $True$ such that[6] $(x_1 = a_1 \wedge \ldots \wedge x_k = a_k) \wedge d$ is $\mathcal{S}$-satisfiable.

A class $L$ of CC($\mathcal{FD}$) programs is *subsumed* by another class $L'$ of CC($\mathcal{FD}$) programs if for every program $P$ in $L$ there exists a program $P'$ in $L'$ defining the same query. Two classes of CC($\mathcal{FD}$) programs are *equivalent* if they subsume each other, i.e. they define the same set of queries. Similar comparisons can be made to any query-defining language, such as FO, Datalog, Datalog$^\neg$, etc. Similarly, we will say that a class $L$ of CC($\mathcal{FD}$) programs *expresses a complexity class* C of queries if $L$ expresses precisely the set of queries of complexity C.

# 3 CC($\mathcal{FD}$) with Success Semantics

We first define the operational semantics of CC($\mathcal{FD}$) with success semantics. CC($\mathcal{FD}$) programs equipped with this semantics are denoted by CC$_s$($\mathcal{FD}$). We then consider the expressiveness and complexity of CC$_s$($\mathcal{FD}$) with complete and open constraint systems, and within each the fragments with no guards, flat guards, and deep guards. We also briefly consider a variant of CC$_s$($\mathcal{FD}$) based on linear logic constraint systems, denoted LCC($\mathcal{FD}$).

[5] Depending on which the semantics requires.

[6] In a complete constraint system (i.e. if $\mathcal{S} = \mathcal{S}_\mathbf{I}$) that condition is equivalent to $(x_1 = a_1 \wedge \ldots \wedge x_k = a_k), \mathcal{S} \vdash d$.

## 3.1 The success semantics

A *success configuration* for an agent $A$ and an initial store $c$ is a configuration $(\vec{x}; d; \epsilon)$ such that $(\emptyset; c; A) \longrightarrow^* (\vec{x}; d; \epsilon)$ and $d$ is consistent with $\mathcal{S}$. The *operational semantics* of a program $D.A$ is thus defined relatively to an initial store $c$ and an $\mathcal{FD}$ constraint system $\mathcal{S}$, as the input/output relation provided by the success stores, that is:

$$\mathcal{O}(D.A; c) = \{\exists \vec{x} d \in \mathcal{C} \mid (\emptyset; c; A) \longrightarrow^* (\vec{x}; d; \epsilon), \; d \text{ is consistent with } \mathcal{S}\}$$

The *structural congruence* $\equiv$ is the least congruence satisfying the rules of Table 1 in Appendix. The *transition relation* $\longrightarrow \in \mathcal{K} \times \mathcal{K}$ is defined as usual in Table 2 in Appendix.

Intuitively, a deep guard $A$ is satisfied in a store $c$ if there is a finite derivation from $A$ in $c$ with a success store equivalent to $c$. In particular a flat guard, i.e. the agent $tell(d)$ in an ask agent $tell(d) \rightarrow A$, is satisfied if the constraint $d$ is entailed by the current store.

## 3.2 CC$_s$($\mathcal{FD}$) with complete $\mathcal{FD}$ constraint systems

In this section we consider several classes of CC$_s$($\mathcal{FD}$) programs with complete constraint systems. These differ in the power of the guards used in their blocking ask constructs. As we shall see, the type of guards (flat or deep) has a significant impact on the expressiveness and complexity of CC$_s$($\mathcal{FD}$) programs. We will also consider the special case of CC$_s$($\mathcal{FD}_\leq$) programs. Recall that these are CC$_s$($\mathcal{FD}$) programs whose constraint system $\mathcal{S}$ specifies a total order $\leq$ on the finite domain. We obtain three kinds of results: (i) equivalences with Datalog($^\neg$) languages, (ii) complexity characterizations, and (iii) comparisons among CC$_s$($\mathcal{FD}$) languages and normal forms.

**CC$_s$($\mathcal{FD}$) and Datalog$^{(\neg)}$** Our first result relates CC$_s$($\mathcal{FD}$) with various types of guards to Datalog$^{(\neg)}$ languages. All $\mathcal{FD}$ constraint systems are assumed complete.

**Theorem 3.1** *(i) CC$_s$($\mathcal{FD}$) without guards is equivalent to semi-positive Datalog$^\neg$.*
*(ii) CC$_s$($\mathcal{FD}$) with flat guards strictly subsumes semi-positive Datalog$^\neg$.*
*(iii) CC$_s$($\mathcal{FD}$) with deep guards is subsumed by Fixpoint (and Datalog$^\neg$).*

**Proof.** (i) The simulation of semi-positive Datalog$^\neg$ is very straightforward, as negation in semi-positive Datalog$^\neg$ only relates to *edb* relations, which

are completely defined in the constraint system. For example, the following program

$$\begin{array}{lll} t(x,y) & \leftarrow & \neg r(x,y) \\ t(x,y) & \leftarrow & \neg r(x,z), t(z,y) \end{array}$$

computing transitive closure on the complement is simulated by the $\mathrm{CC}_s(\mathcal{FD})$ program without guards $\mathcal{D}.A$ where $A :: t(x,y)$ and $\mathcal{D}$ contains the declaration

$$t(x,y) :: tell(\neg r(x,y)) + \exists z(tell(\neg r(x,z)) \parallel t(z,y)).$$

The simulation of $\mathrm{CC}_s(\mathcal{FD})$ programs without guards by semi-positive Datalog$^\neg$ is equally straightforward.

(ii) As $\mathrm{CC}_s(\mathcal{FD})$ with flat guards subsumes $\mathrm{CC}_s(\mathcal{FD})$ without guards, the same translation applies.

An example of a query expressible by $\mathrm{CC}_s(\mathcal{FD})$ programs with flat guards but not by semi-positive Datalog$^\neg$ is finding the minimum and maximum elements of a totally ordered finite domain. For example, the minimum is defined by $min(x) :: \exists y(tell(y \geq x) \rightarrow tell(True))$.

(iii) The inclusion in *fixpoint* is straightforward, using the denotational semantics for $\mathrm{CC}_s(\mathcal{FD})$ with deep guards, given in Table 3 in Appendix. □

**Complexity** Theorem 3.1 in conjunction with known results from finite-model theory yields connections between classes of $\mathrm{CC}_s(\mathcal{FD})$ programs and complexity classes of queries. In the presence of order, we obtain the following expressiveness results.

**Theorem 3.2** *The following classes of $CC_s(\mathcal{FD}_\leq)$ programs with complete constraint systems express precisely the queries over ordered domains computable in polynomial time:*

*(i) $CC_s(\mathcal{FD}_\leq)$ with flat guards.*

*(ii) $CC_s(\mathcal{FD}_\leq)$ with deep guards.*

**Proof.** The inclusion of P in $\mathrm{CC}_s(\mathcal{FD}_\leq)$ with flat guards follows from the fact that the latter subsumes semi-positive Datalog$^\neg$ and can define *min* and *max* of an ordered domain (see proof of Theorem 3.1), and the fact that semi-positive Datalog$^\neg$ defines P on ordered domains equipped with *min* and *max*. The inclusion of $\mathrm{CC}_s(\mathcal{FD}_\leq)$ with deep guards in P follows from from Theorem 3.1 (iii), and the fact the *fixpoint* queries are in P. □

Theorem 3.2 shows, in particular, that $\mathrm{CC}_s(\mathcal{FD}_\leq)$ with flat guards and $\mathrm{CC}_s(\mathcal{FD}_\leq)$ with deep guards collapse on ordered domains. This yields a strong normal form for $\mathrm{CC}_s(\mathcal{FD}_\leq)$:

**Corollary 3.3** *Every $CC_s(\mathcal{FD}_\leq)$ program with deep guards is equivalent to some $CC_s(\mathcal{FD}_\leq)$ program with flat guards.*

## 3.3 $\mathrm{CC}_s(\mathcal{FD})$ with open $\mathcal{FD}$ constraint systems

In this section we consider the expressiveness and complexity of $\mathrm{CC}_s(\mathcal{FD})$ programs with open $\mathcal{FD}$ constraint systems. As earlier, we consider constraint system with equality and a relational vocabulary. Among relations in the vocabulary, we distinguish a set of *input relations* which hold the finite structure input to the program. Since the constraint system is open, the input finite structure may be incompletely specified.

$\mathrm{CC}_s(\mathcal{FD})$ programs without guards on open constraint systems turn out to be equivalent to semi positive Datalog$^\neg$. Therefore, we only consider in this section programs with guards. As in the previous section we consider programs with flat guards, and unrestricted deep guards. Our main result provides exact characterizations of these languages in terms of complexity classes.

**Theorem 3.4** *On open constraint systems $CC_s(\mathcal{FD})$ with flat guards and $CC_s(\mathcal{FD})$ with deep guards are equivalent and expresses precisely the* NP *queries.*

**Proof.** We use the characterization of NP by ∃SO (existential second-order logic). Consider an ∃SO formula $\exists\ S\ \varphi(S)$ where $\varphi$ is in FO. This is simulated by a $\mathrm{CC}_s(\mathcal{FD})$ program with flat guards as follows:
(i) nondeterministically construct a relation $S$;
(ii) nondeterministically construct a total order $\leq$ on the finite domain.
(iii) simulate the evaluation of $\varphi(S)$.

Once $S$ and $\leq$ are constructed (this is more detailed in the appendix), $\varphi(S)$ can be simulated by Theorem 3.2. Indeed, $\mathrm{CC}_s(\mathcal{FD})$ with flat guards and complete constraint systems expresses P on ordered structures, and therefore can simulate $\varphi(S)$ making use of $\leq$.

For the converse inclusions, note that a $\mathrm{CC}_s(\mathcal{FD})$ program with deep guards can be executed in NP. Indeed, a successful derivation in $\mathrm{CC}_s(\mathcal{FD})$ with an open constraint system can be described as a guess of a completion of the constraint system followed by a

successful derivation in the complete constraint system which is in P by Theorem 3.2. □

### 3.4 LCC($\mathcal{FD}$) with linear $\mathcal{FD}$ constraint systems

In this section we study a variant of CC languages where constraints can be removed from the stores, thus combining constraint propagation with state change. We use constraint systems based on Girard's linear logic [Gir87]. Intuitively, linear logic is a calculus of resource consumption where a linear implication consumes its premises to establish the conclusion. This naturally models state change in a logical setting. There have been various proposals to combine CC programming with linear logic (see [SL92], [Pal97]). We use here the linear concurrent constraint language LCC described in [FRS98a, FRS98b], and used for example in [Sch98] to express global constraint solvers where non-monotonic imperative data structures have to be handled.

A *linear constraint system* is a generalization of a classical constraint system, where the constraint language is a fragment of first-order linear logic containing 1 (true), 0 (false), closed under renaming, tensor product $\otimes$ (conjunction) and existential quantification, and where the axioms are linear logic formulas. Linear constraint systems generalize classical constraint systems as the latter can be retrieved from the usual translation of classical logic into linear logic (see [Gir87] p. 81): a classical atom $A$ is translated by prefixing it with the bang operator $!A$, signifying arbitrary duplication of $A$. Thus, linear deduction from $!A$ coincides with classical deduction from atom $A$.

A *linear $\mathcal{FD}$ constraint system* is defined similarly to the previous sections with the following axioms:

i) $!(\forall x\ x = x)$
ii) $!(\forall x, y, \vec{z}\ !(x = y) \otimes r(x, \vec{z}) \multimap r(y, \vec{z}))$ for every relation symbol $r$
iii) $!(\forall x, y, z\ !(x = y) \otimes !(y = z) \multimap !(x = z))$
iv) $!(a_i \neq a_j)$ for $i \neq j$
v) $!(\forall x\ !(x = a_1) \oplus ... \oplus !(x = a_n))$
vi) $r_p(a_i, a_j), ..., \overline{r}_q(a_k, a_l), ...$
vii) $\forall \vec{x}\ r(\vec{x}) \otimes \overline{r}(\vec{x}) \multimap 0$

Equality axioms constitute the classical part of the system, and are thus prefixed with !, whereas the other relations can be consumed by linear implication. Axiom ii) specifies that the linear relations are modulo equality. Axioms vi) specify (partially) a finite structure using relation symbol $\overline{r}$ for the negation of $r$, and axiom vii) specifies that $r$ and $\overline{r}$ are inconsistent.

An *LCC program* is simply a CC program with a linear constraint system, with flat guards only (deep guards for LCC programs have not been investigated yet). The rules for linear tell and ask are provided in Table 4

The constraints are accumulated in the store by tensor product, and are consumed by ask agents. In the case of $\mathcal{FD}$ constraint systems, these operations of tensor product and consumption are in fact simple multiset operations over the relations modulo equality. The expressiveness and complexity of LCC($\mathcal{FD}$) are characterized by the following:

**Theorem 3.5** *LCC($\mathcal{FD}$) expresses the* PSPACE *queries.*

**Proof.** To prove that LCC($\mathcal{FD}$) expresses all the PSPACE queries, we show the following:
(i) using the nondeterminism afforded by the open constraint system, LCC($\mathcal{FD}$) can guess a total order $\leq$ of the domain (and can also define its *min* and *max* elements);
(ii) like CC$_s$($\mathcal{FD}$) with flat guards, LCC($\mathcal{FD}$) can simulate semi-positive Datalog$^\neg$ queries, so can express P using $\leq$, *min* and *max*. As a corollary, FO can be expressed.
(iii) the recursion available in LCC($\mathcal{FD}$), together with the noninflationary ability provided by linear logic, allow to simulate PFP.
(iv) the result now follows from the fact that PFP expresses PSPACE on ordered structures. More details on the simulation of PFP are provided in Appendix. □

## 4 Terminal store semantics

In this section we consider the operational semantics of CC($FD$) programs based on the observation of terminal stores, instead of success store: the observed output of a program is the set of stores of the terminal configurations, no matter whether these configurations contain suspended agents or not [SRP91, BGMP97, BGP96].

In this context the semantics of deep guards is changed in order to check whether the current store is a terminal store (not necessarily a success store) of the guard agent. This simple intuition introduces however some complications in the definition of the transition relation in the presence of recursion through deep guards.

Intuitively a deep guard $A$ is satisfied in a store $c$ if there is a finite derivation from $A$ in $c$ with a terminal store equivalent to $c$. In particular a flat

guard, i.e. the agent $tell(d)$ in an ask agent $tell(d) \rightarrow A$, is satisfied if the constraint $d$ is entailed by the current store.

The formal difficulty comes from recursion through deep guards, as in a declaration like

$$A :: B \rightarrow A, \qquad B :: A \rightarrow B.$$

Intuitively, the problem is similar to giving semantics to Datalog$^\neg$ programs where negation is used recursively. As for Datalog$^\neg$, there are many possible solutions. We adopt a semantics inspired by the *well-founded* semantics for Datalog$^\neg$ [VGRS91], which results in intuitively appealing behavior of programs with deep guards. In particular, the resulting semantics agrees with the usual operational semantics of $\mathrm{CC}_{ts}$ for flat guards or stratified deep guards.

The key to the semantics of deep guards is to define the set of deep guards that succeed. This is done similarly to the alternating fixpoint computation of the well-founded semantics [VG89] (see Appendix). We define $\longrightarrow_{GS}$, a version of the transition relation parameterized by a guard success set $GS$, as the least transitive relation on configurations satisfying the rules of Table 5. The *structural congruence* $\equiv$ is the same as before. $\longrightarrow_{GS}$ is then used to define inductively approximations of the set of guards that should succeed, $GS_{2i}$ being upper approximations and $GS_{2i+1}$ being lower ones, keeping in mind that a guard succeeds when it leads to a terminal store identical to the current one. From these we can define $\longrightarrow$ as in Table 6, taking the limit $GS = \bigcup_{i\geq 0} GS_{2i+1}$ as guard success set, $GF = \bigcup_{i\geq 0} \overline{GS_{2i}}$ as guard failure set, and rewriting to an inconsistent store the ask agents with a guard in $\overline{GS \cup GF}$ (i.e. the undefined part of the well-founded semantics). The computation is illustrated by three examples provided in Appendix.

With the transition relation $\longrightarrow \in \mathcal{K} \times \mathcal{K}$ defined as above (using Tables 5 and 6), we can now complete the definition of the terminal store semantics. A *terminal configuration* for an agent $A$ and an initial store $c$ is a configuration $(\vec{x}; d; \Gamma)$ such that $(\emptyset; c; A) \longrightarrow^* (\vec{x}; d; \Gamma)$, $(\vec{x}; d; \Gamma) \not\longrightarrow$ and $d$ is consistent with $\mathcal{S}$. The *terminal stores operational semantics* of a program $D.A$ is thus defined by

$$\mathcal{O}^{ts}(D.A; c) = \{\exists \vec{x} d \in \mathcal{C} \mid (\emptyset; c; A) \longrightarrow^* (\vec{x}; d; \Gamma) \not\longrightarrow, \; d \text{ is consistent with } \mathcal{S}\}$$

It is worth noting that reachability of terminal stores is not monotonic. For example, with declarations

$$A :: (tell(c) \rightarrow tell(e)), \; B :: (A \rightarrow tell(d))$$

the agent $B$ in the empty initial store $True$ reaches terminal store $d$, i.e. $< \emptyset; True; B > \longrightarrow < \emptyset; d; \epsilon >$, whereas it does not reach a terminal store containing $d$ from an initial store entailing $c$:

$$< \emptyset; c; B > \longrightarrow < \emptyset; c; (A \rightarrow tell(d)) > \not\longrightarrow .$$

## 4.1 Complete $\mathcal{FD}$ constraint systems

In this section we consider several classes of $\mathrm{CC}_{ts}(\mathcal{FD})$ programs with complete constraint systems. These differ in the power of the guards used in their blocking ask constructs. As we shall see, the type of guards has a significant impact on the expressiveness and complexity of $\mathrm{CC}_{ts}(\mathcal{FD})$ programs. As for the success semantics, we obtain results on equivalences with Datalog($^\neg$) languages, complexity characterizations, and comparisons among $\mathrm{CC}_{ts}(\mathcal{FD})$ languages and normal forms. We will also consider the special case of $\mathrm{CC}_{ts}(\mathcal{FD}_{\leq})$ programs.

In addition to programs with flat guards and deep guards, we consider in this section an intermediate class of $\mathrm{CC}_{ts}(\mathcal{FD})$ programs with *stratifiable* deep guards. These are defined using a dependency graph among the agents of the program. Let $\mathcal{D}.A$ be a $\mathrm{CC}_{ts}(\mathcal{FD})$ program. The dependency graph associated with $\mathcal{D}.A$ is constructed as follows. The nodes consist of all agents in $\mathcal{D}$. There is an edge from $B$ to $A$ if $B$ occurs in the definition of $A$, or if $p(\vec{x})$ is used in the definition of $A$ and $p(\vec{x}) :: B$ is in $\mathcal{D}$. Furthermore, the edge from $B$ to $A$ is labeled *ask* if $\forall \vec{y}(B \rightarrow C)$ appears in the definition of $A$. A $\mathrm{CC}_{ts}(\mathcal{FD})$ program $\mathcal{D}.A$ has *stratifiable deep guards* if its dependency graph contains no cycle with an edge labeled *ask*.

**$\mathrm{CC}_{ts}(\mathcal{FD})$ and Datalog$^{(\neg)}$** Our first result relates $\mathrm{CC}_{ts}(\mathcal{FD})$ with various types of guards to usual query languages. All $\mathcal{FD}$ constraint systems are assumed complete.

**Theorem 4.1** *(i) $CC_{ts}(\mathcal{FD})$ without guards is equivalent to semi-positive Datalog$^\neg$.*
*(ii) $CC_{ts}(\mathcal{FD})$ with flat guards strictly subsumes semi-positive Datalog$^\neg$.*
*(iii) $CC_{ts}(\mathcal{FD})$ with stratifiable deep guards is equivalent to stratifiable Datalog$^\neg$.*
*(iv) $CC_{ts}(\mathcal{FD})$ with deep guards is equivalent to Datalog$^\neg$ with well-founded semantics (and thus expresses the* fixpoint queries*).*

**Proof.** The simulations involved in (i)-(ii) are straightforward and similar to those in Theorem 3.1. Negation in stratifiable Datalog$^\neg$ is simulated in

$\mathrm{CC}_{ts}(\mathcal{FD})$ programs using stratifiable deep guards. Essentially, a rule of the form

$$r(\vec{x}) \leftarrow \ldots \neg p(\vec{y}) \ldots$$

is simulated by an agent

$$A :: \ldots \parallel (p(\vec{y}) \rightarrow tell(False)) \parallel \ldots$$

If the Datalog$^\neg$ program is semi-positive, only flat guards are used. If the Datalog$^\neg$ program is stratifiable, the resulting $\mathrm{CC}_{ts}(\mathcal{FD})$ program has stratifiable deep guards. Some care is however needed, due to the operational semantics of guards. Indeed, a guard $A$ with free variables $\vec{x}$ is satisfied in a configuration with store $c$ if it is satisfied for *all* values of $\vec{x}$ in $\pi_{\vec{x}}(c)$. This means, for example, that the semi-positive program $t(x) \leftarrow p(x), \neg q(x)$ computing the difference of $p$ and $q$ is *not* equivalent to $t(x) :: p(x) \parallel (q(x) \rightarrow tell(False))$. To achieve equivalence, one must force a derivation where $x$ is instantiated when the guard $p(x)$ is tested. This can be done by adding an ask agent for enumeration as follows:

$$\begin{array}{rcl} t(x) & :: & \forall y(tell(y = y) \rightarrow x = y) \parallel p(x) \\ & & \parallel (q(x) \rightarrow tell(False)). \end{array}$$

This idea can be extended to fully simulate semi-positive and stratifiable Datalog$^\neg$ programs by $\mathrm{CC}_{ts}(\mathcal{FD})$ programs. Conversely, a stratifiable deep guard can be easily simulated in stratifiable Datalog$^\neg$. A flat guard requires three strata. To see this, note that an agent of the form $tell(c) \rightarrow A$ on store $d$ is logically equivalent to

$$(\forall \vec{x}(tell(d) \rightarrow c) \wedge A) \vee \exists \vec{x}(d \wedge \neg c)$$

which can be simulated by a stratifiable Datalog$^\neg$ program with three strata. An example of a query expressible by $\mathrm{CC}_{ts}(\mathcal{FD})$ programs with flat guards but not by semi-positive Datalog$^\neg$ is finding the minimum and maximum elements of a totally ordered finite domain. For example, the minimum is defined by $min(x) :: (tell(\exists y\ (y < x)) \rightarrow tell(False))$.

Consider (iv). The equivalence of $\mathrm{CC}_{ts}(\mathcal{FD})$ program with deep guards and Datalog$^\neg$ with well-founded semantics is straightforward because of the similarity of the deep guards semantics with the well-founded semantics. □

**Complexity** Theorem 4.1 in conjunction with known results from finite-model theory yields connections between classes of $\mathrm{CC}_{ts}(\mathcal{FD})$ programs and complexity classes of queries. First, all $\mathrm{CC}_{ts}(\mathcal{FD})$ programs with complete constraint systems can be evaluated in polynomial time. Moreover, in the presence of order, we obtain the following expressiveness results.

**Theorem 4.2** *The following classes of $CC_{ts}(\mathcal{FD}_{\leq})$ programs with complete constraint systems express precisely the queries over ordered domains computable in polynomial time:*
*(i) $CC_{ts}(\mathcal{FD}_{\leq})$ with deep guards.*
*(ii) $CC_{ts}(\mathcal{FD}_{\leq})$ with stratifiable deep guards; and*
*(iii) $CC_{ts}(\mathcal{FD}_{\leq})$ with flat guards.*

**Relative expressiveness and normal forms** The previous results allow us to compare different classes of $\mathrm{CC}_{ts}(\mathcal{FD})$ programs with complete constraint systems. From Theorem 4.1 and known separation results among Datalog$^{(\neg)}$ languages we have:

**Theorem 4.3** *(i) $CC_{ts}(\mathcal{FD})$ with deep guards strictly subsumes $CC_{ts}(\mathcal{FD})$ with stratifiable deep guards.*
*(ii) $CC_{ts}(\mathcal{FD})$ with stratifiable deep guards strictly subsumes $CC_{ts}(\mathcal{FD})$ with flat guards.*
*(iii) $CC_{ts}(\mathcal{FD})$ with flat guards strictly subsumes $CC_{ts}(\mathcal{FD})$ programs with no guards.*

In contrast to the separation results of Theorem 4.3, in the general case, all languages with guards collapse on ordered domains:

**Theorem 4.4** *The following classes of $CC_{ts}(\mathcal{FD}_{\leq})$ programs are equivalent:*
*(i) $CC_{ts}(\mathcal{FD}_{\leq})$ with deep guards.*
*(ii) $CC_{ts}(\mathcal{FD}_{\leq})$ with stratifiable deep guards.*
*(iii) $CC_{ts}(\mathcal{FD}_{\leq})$ with flat guards.*

As a side effect, as in the case of the success semantics, this yields strong normal forms for $\mathrm{CC}_{ts}(\mathcal{FD}_{\leq})$ programs. Most notably:

**Corollary 4.5** *Every $CC_{ts}(\mathcal{FD}_{\leq})$ program with deep guards is equivalent to a $CC_{ts}(\mathcal{FD}_{\leq})$ program with flat guards.*

**Deep guards with arithmetic constraints** We have seen that $\mathrm{CC}_{ts}(\mathcal{FD})$ with deep guards expresses the *fixpoint* queries. It is natural to wonder if there is a variant of $\mathrm{CC}_{ts}(\mathcal{FD})$ with deep guards that corresponds to PFP (and so expresses the *while* queries). We next present such a language. Consider $\mathrm{CC}_{ts}(\mathcal{FD}^{ind})$ whose declarations include inductive definitions of processes. All inductively defined

processes contain one integer variable and are defined by declarations of the form:

$$\begin{array}{lcl} t(\vec{x},0) & :: & A \\ t(\vec{x},i+1) & :: & B(i) \end{array}$$

where $A$ contains no inductively defined processes, and $B(i)$ contains inductively defined processes $p(\vec{y},i)$. Declarations of non-inductively defined processes may use inductively defined processes, but all non-inductively defined processes are nonrecursive. The predicate $+$ is only used in inductive definitions, as shown above. A $\mathrm{CC}_{ts}(\mathcal{FD}^{ind})$ program is of the form $\mathcal{D}.p(\vec{x})$ where $p$ is non-inductively defined.

We can show the following:

**Theorem 4.6** *(i) $CC_{ts}(\mathcal{FD}^{ind})$ is equivalent to PFP.*
*(ii) $CC_{ts}(\mathcal{FD}^{ind}_{\leq})$ expresses precisely the* PSPACE *queries on ordered domains.*

**Proof.** It is sufficient to show (i), since (ii) follows from (i) and the fact that PFP expresses PSPACE on ordered structures. Consider (i). The simulation of PFP by $\mathrm{CC}_{ts}(\mathcal{FD}^{ind})$ is straightforward. The converse is harder and uses results on the inductive definability of $FO^k$-types, including a total order on them [AV95, DLW95, KV92]. More details are provided in the Appendix. □

## 4.2 Open $\mathcal{FD}$ constraint systems

In this section we consider the expressiveness and complexity of $\mathrm{CC}_{ts}(\mathcal{FD})$ programs with open $\mathcal{FD}$ constraint systems. As earlier, we consider constraint system with equality and a relational vocabulary. Among relations in the vocabulary, we distinguish a set of *input relations* which hold the finite structure input to the program. Since the constraint system is open, the input finite structure may be incompletely specified. To prevent the content of input relations from changing throughout the evaluation of a program, we prohibit programs from including agents of the form $tell(r(\vec{x}))$ where $r$ is in the input signature, except within guards.

$\mathrm{CC}_{ts}(\mathcal{FD})$ programs without guards behave the same in the case of open constraint systems as in the case of complete systems already examined – they are equivalent to Datalog. Therefore, we only consider in this section programs with guards. As in the previous section we consider programs with flat guards, stratifiable deep guards, and unrestricted deep guards. Our main result provides exact characterizations of these languages in terms of complexity classes.

**Theorem 4.7** *(i) $CC_{ts}(\mathcal{FD})$ with flat guards expresses precisely the* NP *queries.*
*(ii) $CC_{ts}(\mathcal{FD})$ with stratifiable guards expresses precisely the* PHIER *queries.*
*(iii) $CC_{ts}(\mathcal{FD})$ with deep guards expresses precisely the* PSPACE *queries.*

**Proof.** The upper bounds are mostly straightforward. The simulations for the lower bounds make use of the nondeterminism allowed by the open constraint system and the various forms of negation provided by the guards. The simulation of PSPACE queries by $\mathrm{CC}_{ts}(\mathcal{FD})$ with deep guards is done using a nondeterministic variant of IFP with alternation that expresses APTIME (and therefore PSPACE) on ordered structures [AVV97]. See Appendix for more details. □

Note that the relationship between the $\mathrm{CC}_{ts}(\mathcal{FD})$ programs with various kinds of guards remains unresolved in the case of open constraint systems. Moreover, from Theorem 4.7 it follows that these relationships are equivalent to well-known open problems in complexity theory.

The results on languages expressing PSPACE yield a rather unexpected connection between languages with complete $\mathcal{FD}$ constraint systems and languages with open $\mathcal{FD}$ constraint systems:

**Corollary 4.8** *The following are equivalent*[7]*:*
*(i) $CC_{ts}(\mathcal{FD}^{ind}_{\leq})$ with complete constraint systems.*
*(ii) $CC_{ts}(\mathcal{FD})$ with open constraint systems.*

# 5 Conclusion

The results presented in the paper provide considerable information on the expressiveness and complexity of $\mathrm{CC}(\mathcal{FD})$ with various semantics and features. We briefly summarize what we have learned.

**Complexity** We were able to exhibit languages expressing complexity classes from polynomial time to polynomial space, as shown in Figure 1. Not surprisingly, order is needed in all cases to express P, whereas higher complexity classes can be expressed without order. The class PSPACE is especially interesting, because it is expressed by languages which seem very different computationally: $\mathrm{CC}_{ts}(\mathcal{FD}^{ind}_{\leq})$ is similar to the *while* queries, and requires order to express PSPACE; $\mathrm{LCC}(\mathcal{FD})$ provides a mix of noninflationary computation and the ability to simulate order; lastly, $\mathrm{CC}_{ts}(\mathcal{FD})$ is inflationary in flavor

[7] Ignoring the differences in the ordering assumptions.

| complexity class | CC($\mathcal{FD}$) | | |
|---|---|---|---|
| | constraint system | guards | language |
| P | complete | flat | $CC_s(\mathcal{FD}_{\leq})$ $CC_{ts}(\mathcal{FD}_{\leq})$ |
| | | deep | $CC_s(\mathcal{FD}_{\leq})$ $CC_{ts}(\mathcal{FD}_{\leq})$ |
| NP | open | flat | $CC_s(\mathcal{FD})$ $CC_{ts}(\mathcal{FD})$ |
| | | deep | $CC_s(\mathcal{FD})$ |
| PHIER | open | stratified | $CC_{ts}(\mathcal{FD})$ |
| PSPACE | complete | deep | $CC_{ts}(\mathcal{FD}^{ind}_{\leq})$ |
| | open | deep | $CC_{ts}(\mathcal{FD})$ $LCC(\mathcal{FD})$ |

Figure 1: Complexity classes expressed by CC languages

but expresses PSPACE due to its ability to simulate polynomial-time alternation.

**Impact of order** As in finite model theory, the presence of order has drastic consequences on the expressiveness of CC($\mathcal{FD}$) languages. Languages with flat or deep guards, success or terminal store semantics, become equivalent on ordered domains (with complete constraint systems). As a side effect, this yields powerful normal forms for such languages. In particular, deep guards are not needed for expressiveness, although they may considerably facilitate programming.

**Complete and open constraint systems** All CC($\mathcal{FD}$) languages stay within P in the context of complete constraint systems (except $CC_{ts}(\mathcal{FD}^{ind}_{\leq})$). Thus, open constraint systems are generally needed to go beyond P. Computationally, open constraint systems introduce nondeterminism. This allows expressing NP, PHIER, and PSPACE (via APTIME).

**Flat and deep guards** In the absence of order, flat guards are generally weaker than deep guards. Indeed, there is a strong analogy between guards and negation in query languages. In complete constraint systems, CC with flat guards generally corresponds to semi-positive Datalog$^{\neg}$, and CC with deep guards is analogous to languages allowing recursion through negation. In open constraint systems, guards provide a powerful combination of negation and nondeterminism.

**Success and terminal store semantics** The two semantics differ in the manner and extent to which negation can be simulated. In the context of complete constraint systems, the success and terminal store semantics stand to each other in a relationship similar to LFP and IFP: LFP provides limited negation on non-inductively defined relations, whereas IFP can explicitly negate inductively defined relations. Nonetheless, LFP is closed under complement and turns out to be equivalent to IFP [GS86]. Likewise, $CC_s(\mathcal{FD})$ and $CC_{ts}(\mathcal{FD})$ are equivalent in the context of complete constraint systems. The equivalence does not however extend to open constraint systems: $CC_s(\mathcal{FD})$ is confined to NP, whereas $CC_{ts}(\mathcal{FD})$ can express PSPACE.

The results in the paper provide a fairly complete picture of the expressiveness and complexity of CC($\mathcal{FD}$) on finite domains. The connection to finite model theory has proven fruitful, and there remain many aspects to be explored. For example, it may be of interest to transfer known normal forms for query languages to CC($\mathcal{FD}$). Furthermore, desirable computational properties and semantics of query languages can inspire new variants of CC($\mathcal{FD}$). The terminal store semantics, suggested by the well-founded semantics, is one such example. Another possibility we have considered (but not developed here due to lack of space), is to study a semantics of CC($\mathcal{FD}$) analogous to the *certainty* and *possibility* semantics for nondeterministic query languages (in the sense that each query can generate several possible answers, see [ASV90]). Such nondeterminism arises naturally in CC($\mathcal{FD}$) with open constraint systems, where a program can generate different sets of terminal stores on a given input.

Clearly, the connection between CC($\mathcal{FD}$) and finite model theory is natural and can lead to beneficial cross fertilization.

# References

[AHV95] S. Abiteboul, R. Hull and V. Vianu. *Foundations of Databases*. Addison Wes-

ley, 1995.

[ASV90] S.Abiteboul, E. Simon, and V.Vianu. Non-deterministic Languages to Compute Deterministic Transformations. In *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, 218–229, 1990.

[AV89] S. Abiteboul and V. Vianu. Fixpoint Extensions of First-Order Logic and Datalog-Like Languages. In *Proc. Logic in Computer Science*, 71-79, 1989.

[AV95] S. Abiteboul and V. Vianu. Computing with First-Order Logic. *J. of Computer and System Sciences*, 50:2, 309-335, 1995.

[AVV97] S. Abiteboul, M. Vardi and V. Vianu. Fixpoint Logics, Relational Machines, and Computational Complexity. *J. of the ACM* , 44(1), 30-56, 1997.

[AU79] A.V. Aho and J.D. Ullman. Universality of data retrieval languages. In *Proc. 6th ACM Symp. on Principles of Programming Languages*, pages 110–117, 1979.

[BGMP97] F.S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. *ACM-TOPLAS*, 19(5):685–725, 1997.

[BGP96] F.S. de Boer, M. Gabbrielli and C. Palamidessi. Proving correctness of constraint logic programs with dynamic scheduling. *Proc. Static Analysis SAS*, Srpinger-Verlag, LNCS 1145, p.83-97, 1996.

[CH82] A. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25:99–128, 1982.

[CKS81] A.K. Chandra, D.C. Kozen and L. Stockmeyer. Alternation. *J. of the ACM* 28, 114–133, 1981.

[DLW95] A. Dawar, S. Lindell and S. Weinstein. Infinitary Logic and Inductive Definability over Finite Structures. *Information and Computation* 119(2): 160-175 (1995).

[EF98] H-D. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer Verlag, 1995.

[Fag74] R. Fagin. Generalized first–order spectra and polynomial–time recognizable sets. In R. M. Karp, editor, *Complexity of Computation, SIAM-AMS Proceedings, Vol. 7*, pages 43–73, 1974.

[Fag75] R. Fagin. Monadic generalized spectra. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 21:89–96, 1975.

by François Fages, Paul Ruet and Sylvain Soliman. This is a LIENS research report R98-5 submitted to a journal in May 98. It is available as ps and dvi.

[FRS98a] F. Fages, P. Ruet, S. Soliman. Linear concurrent constraint programming: operational and phase semantics. *Research Report LIENS-98-5*, May 1998.

[FRS98b] F. Fages, P. Ruet, S. Soliman. Phase semantics and verification of concurrent constraint programs. *Logic in Computer Science LICS'98*, Indianapolis. 1998.

[Gir87] J.Y. Girard. Linear logic. *Theoretical Computer Science, Vol. 50(1)*, 1987.

[GS86] Gurevich, Y., and S.Shelah. Fixed-point extensions of first-order logic. *Annals of Pure and Applied Logic* 32, North Holland (1986), 265-280. Also in *Proc. 26th Symp. on Found. of Computer Science* (1985), 346-353.

[Imm86] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68:86–104, 1986.

[Imm87a] N. Immerman. Expressibility as a complexity measure: results and directions. In *Second Structure in Complexity Conference*, pages 194–202, 1987.

[Imm87b] N. Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16:760–778, 1987.

[JL87] J. Jaffar and J-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, January 1987.

[JM94] J. Jaffar and M.J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19-20:503–581, May 1994.

[JS74] N. G. Jones and A.L. Selman. Turing machines and the spectra of first-order formulas. *Journal of Symbolic Logic*, 39:139–150, 1974.

[KG94] P. C. Kolaitis and D.Q. . Goldin Constraint programming and database query languages. In *Proc. 2nd Symposium Theoretical Aspects of Computer Software STACS*, Sendai, Japan, Springer-Verlag LNCS 789, p.96-120, 1994.

[KV92] P. G. Kolaitis and M. Y. Vardi. Fixpoint Logic vs. Infinitary Logic in Finite-Model Theory. In *Proc. Logic in Computer Science* ,46-57, 1992.

[Pal97] C. Palamidessi. Concurrent constraint programming. Invited talk, *JFPLC'97*, Orleans, Ed. Hermès, Paris. 1997.

[Sar87] V.A. Saraswat. Concurrent constraint programming. ACM Doctoral Dissertation Awards. MIT Press, 1993.

[Sch98] V. Schachter. Linear concurrent constraint programming over reals. *Constraint Programming CP'98*, Pisa. 1998.

[SL92] V.A. Saraswat and P. Lincoln, Higher-order linear concurrent constraint programming, *Draft Xerox 1992.*

[SRP91] V.A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *POPL'91: Proceedings 18th ACM Symposium on Principles of Programming Languages*, 1991.

[SSW87] C. Schulte and G. Smolka and J. Würtz. Encapsulated Search and Constraint Programming in Oz. Second Workshop on Principles and Practice of Constraint Programming, A.H. Borning, Orcas Island, USA, Springer-Verlag LNCS 874, pp.134–150, May 1994.

[S77] L. Stockmayer. The polynomial hierarchy. *Theoretical Computer Science*, 3:1–22, 1977.

[VG89] A. Van Gelder. The Alternating Fixpoint of Logic Programs with Negation. In *Proc. ACM Symp. on Principles of Database systems*, 1–11, 1989.

[VGRS91] A. Van Gelder, K.A. Ross and J.S. Schlipf. The Well-Founded Semantics for General Logic Programs. *J. of the ACM*, 38, 620–650, 1991.

[Var82] M. Y. Vardi. The complexity of relational query languages. In *Proc. 14th ACM Symp. on Theory of Computing*, pages 137–146, 1982.

# Appendix

## A: Preliminaries on Finite Model Theory

We refer to [EF98] for a general exposition of finite-model theory. We review here several query languages used in the paper. A discussion of this material in the context of database theory can be found in [AHV95].

**Fixpoint and While** One approach to extending FO with recursion is provided by *fixpoint logics*. These allow inductive definitions of relations by FO formulas. Let $\varphi(x_1, \ldots, x_m, t)$ be a 1st-order formula in which $t$ is a $m$-ary relation symbol (not included in the input vocabulary) and let $\mathbf{I}$ be a finite structure over the input vocabulary. The formula $\varphi(t)$ allows to define inductively the following sequence of relations:

$$\begin{array}{lcl} t_0 & = & \emptyset \\ t_{n+1} & = & \{(a_1, \ldots, a_m) \mid \mathbf{I}, t_n \models \varphi(t)(a_1, \ldots, a_m)\} \end{array}$$

The sequence $\{t_n\}_{n \geq 0}$ converges if for some $k$, $t_k = t_{k+1}$. Then the limit $t_k$ of the sequence is denoted $\mu_t(\varphi(t))$. If the limit does not exist, $\mu_t(\varphi(t))$ is defined to be empty. The extension of FO with the operator $\mu_t$ is called *partial fixpoint logic* (PFP). Note that the sequence $\{t_n\}_{n \geq 0}$ defined by consecutive applications of $\varphi(t)$ is not guaranteed to converge. Convergence can be ensured with the following modification of the definition of $\{t_n\}_{n \geq 0}$:

$$\begin{array}{lcl} t_0 & = & \emptyset \\ t_{n+1} & = & t_n \cup \{(a_1, \ldots, a_m) \mid \mathbf{I}, t_n \models \\ & & \varphi(t)(a_1, \ldots, a_m)\} \end{array}$$

Indeed, the sequence is now increasing by definition and so there exists $k$ such that $t_k = t_{k+1}$. The limit of the sequence thus defined is denoted $\mu_t^+(\varphi(t))$, and FO extended with the operator $\mu_t^+(\varphi(t))$ is called *inflationary fixpoint logic* (IFP). For example, the connectivity of a graph given by a binary relation $r$ is defined in IFP by:

$$\forall x \forall y \ \mu_t(\varphi(t))(x, y)$$

where $\varphi(t)$ is the FO formula:

$$r(x,y) \vee \exists z\ (r(x,z) \wedge t(z,y)).$$

A variant of IFP is the *positive fixpoint logic* LFP. This requires each inductively defined predicate to occur positively in the formula used in the inductive definition. Despite this restriction, LFP remains equivalent to IFP [GS86].

**Datalog and Datalog$^\neg$** Another popular paradigm for recursive query languages is based on logic programming. We consider first Datalog, which can be viewed as a "pure" relational version of Prolog. The syntax of Datalog is essentially that of Horn Clauses. A *Datalog rule* is an expression of the form :

$$r_1(u_1) \leftarrow r_2(u_2), ..., r_n(u_n)$$

where for some $n \geq 1$, $r_1, ..., r_n$ are relation names, $u_1, ..., u_n$ are tuples of variables or constants of appropriate arities. Furthermore, each variable occurring in the head must also occur in the body. A *Datalog program* $P$ is a finite set of Datalog rules. The relations occurring in heads of rules are the *intensional* relations (*idb*) of $P$, and the others are the *extensional* relations (*edb*) of $P$. Each Datalog program defines a query whose input is a finite structure over its edb relations, and whose output is the finite structure it defines over its idb relations. Datalog can be given a model-based semantics, a fixpoint semantics, or a top-down semantics based on SLD-resolution. The three semantics turn out to be equivalent. With the model-based semantics, the result of a Datalog program is the unique minimal model satisfying the sentences corresponding to the rules in the program, and containing the input database. The fixpoint semantics consists of firing the rules of the program with all applicable valuations until a fixpoint is reached. For example, the following Datalog program computes the transitive closure $t$ of the graph whose edges are given in a binary relation $r$:

$$\begin{array}{lcl} t(x,y) & \leftarrow & r(x,y) \\ t(x,y) & \leftarrow & r(x,z), t(z,y) \end{array}$$

Datalog provides recursion, but no negation. In particular, it is incomparable to $FO$. Datalog syntax can be easily extended to include negation in rule bodies, yielding Datalog$^\neg$. A Datalog$^\neg$ rule is an expression of the form

$$r(h) \leftarrow b_1(u_1), ..., b_n(u_n),$$

where: (i) $r$ is a relation name, (ii) $b_i$ are relation names (in which case $b_i$ is called *positive*) or $\neg r_i$ where $r_i$ is a relation name (then $b_i$ is called *negative*), (iii) $h$ and $u_i$ are tuples of variables or constants of appropriate arities, and (iv) every variable in $h$ occurs in some positive $b_i(u_i)$. A Datalog$^\neg$ program is a finite set of Datalog$^\neg$ rules. We will consider two syntactic restrictions of Datalog$^\neg$: semi-positive, and stratified. A Datalog$^\neg$ program is *semi-positive* if the for each negative literal $\neg r_i(u_i)$ occurring in a rule, $r$ is an edb relation. The semantics of a semi-positive Datalog$^\neg$ program is the same as that of Datalog, where $\neg r$ is interpreted as the complement of $r$. A Datalog$^\neg$ program is *stratifiable* if the natural predicate dependency graph associated to the program (where edges are labeled $\neg$ for negative dependencies) contains no cycle with an edge labeled $\neg$. The stratifiability condition allows for an elegant extension of the semantics of semi-positive Datalog$^\neg$: each relation is completely defined before any negative test on it is evaluated. Once the relation is defined, it is treated as an edb relation relative to the remaining rules. Thus, stratifiable Datalog$^\neg$ programs are evaluated by partitioning the rules into "strata" defining one or several relations with no negative dependencies among each other, then evaluating the strata one by one in an order consistent with the negative dependencies. Following is a stratifiable Datalog$^\neg$ program defining in $\overline{t}$ the complement of the transitive closure of a graph whose nodes are given in a unary relation $n$ and whose edges are given in a binary relation $r$:

$$\begin{array}{lcl} t(x,y) & \leftarrow & r(x,y) \\ t(x,y) & \leftarrow & r(x,z), t(z,y) \\ \overline{t}(x,y) & \leftarrow & n(x), n(y), \neg t(x,y) \end{array}$$

Lastly, consider the full Datalog$^\neg$. We consider one semantics called the *well-founded* semantics [VGRS91], which extends the semantics of stratified Datalog$^\neg$. The well-founded semantics is based on the idea that a given program may not necessarily provide complete information on all facts. Instead, some facts may simply be indifferent to it, and the answer should be allowed to say that the truth value of those facts is *unknown*. This yields a 3-valued approach. We briefly describe an *alternating fixpoint* computation of the set of true and false facts defined by a Datalog$^\neg$ under the well-founded semantics. The idea of the computation is the following. We define an alternating sequence $\{\mathbf{I}_i\}_{i\geq 0}$ of consecutive approximations of the true and false facts defined by the program. The intuition behind the construction of the sequence $\{\mathbf{I}_i\}_{i\geq 0}$ is the following. The sequence starts with $\perp$ (all facts are false), which is an overestimate of the negative facts in the answer. From this overestimate we compute $\mathbf{I}_1$, which includes all

positive facts that can be inferred from $\perp$ using an immediate consequence operator inferring both positive and negative facts (we omit the definition). This is clearly an overestimate of the positive facts in the answer, so the set of negative facts in $\mathbf{I}_1$ is an underestimate of the negative facts in the answer. Using this underestimate of the negative facts we compute $\mathbf{I}_2$, whose positive facts will now be an underestimate of the positive facts in the answer. By continuing the process we see that the even-indexed instances provide underestimates of the positive facts in the answer and the odd-indexed ones provide underestimates of the negative facts in the answer. Then the limit of the even-indexed instances, denoted $\mathbf{I}_*$ provides the positive facts in the answer and the limit of the odd-indexed instances, denoted $\mathbf{I}^*$, provides the negative facts in the answer. We illustrate the alternating fixpoint computation with a simple example. Consider the propositional Datalog$^\neg$ program

$$\begin{array}{lcl} p & \leftarrow & \neg r \\ q & \leftarrow & \neg r, p \\ s & \leftarrow & \neg t \\ t & \leftarrow & q, \neg s \\ u & \leftarrow & \neg t, p, s. \end{array}$$

Let us perform the alternating fixpoint computation described above. We start with $\mathbf{I}_0 = \{\neg p, \neg q, \neg r, \neg s, \neg t, \neg u\}$. We obtain the following sequence of instances:

$$\begin{array}{lcl} \mathbf{I}_1 & = & \{p, q, \neg r, s, t, u\}, \\ \mathbf{I}_2 & = & \{p, q, \neg r, \neg s, \neg t, \neg u\}, \\ \mathbf{I}_3 & = & \{p, q, \neg r, s, t, u\}, \\ \mathbf{I}_4 & = & \{p, q, \neg r, \neg s, \neg t, \neg u\}. \end{array}$$

Thus, $\mathbf{I}_* = \mathbf{I}_4 = \{p, q, \neg r, \neg s, \neg t, \neg u\}$ and $\mathbf{I}^* = \mathbf{I}_3 = \{p, q, \neg r, s, t, u\}$. Finally, the set of positive and negative facts defined by the well-founded semantics is $\{p, q, \neg r\}$. The other facts have unknown truth value. A more detailed exposition of the well-founded semantics can be found in [AHV95].

## B: Proof sketches

**Proof of Theorem 3.4** We briefly describe how to construct, nondeterministically, a relation $S$ and then a total order $\leq$, with the following CC$_s(\mathcal{FD})$ program on open constraint systems:

$$\begin{array}{l} guess_S :: \\ \quad ( \forall\, \vec{x}\ (tell(True) \rightarrow tell(S(\vec{x}))) \parallel tell(done_S(x)) \\ \qquad \parallel guess_S ) \\ + ( \forall\, \vec{x}\ (tell(True) \rightarrow tell(\neg S(\vec{x}))) \\ \qquad \parallel tell(done_S(x)) \parallel guess_S ) \\ + \exists x (tell(done_S(s)) \rightarrow order) \end{array}$$

where the $done_S$ relation checks the completeness of the definition of $S$, and $order$ nondeterministically constructs a total order $\leq$ of the domain in a similar way.

The next step is to guess an order $\leq$ in the same way:

$$\begin{array}{l} order :: \\ \quad ( \forall\, x, y\ (tell(True) \rightarrow tell(\leq (x, y))) \\ \qquad \parallel tell(done_{order}(x, y)) \parallel order) \\ + ( \forall\, x, y\ (tell(True) \rightarrow tell(\neg \leq (x, y))) \\ \qquad \parallel tell(done_{order}(x, y)) \parallel order) \\ + \exists x \exists y (tell(done_{order}(x, y)) \rightarrow transitivity) \end{array}$$

and to check that it has the expected properties:

$$\begin{array}{l} transitivity :: \\ \quad ( \forall\, x, y, z\ (tell(True) \rightarrow (tell(\neg \leq (x, y)) \\ \qquad + tell(\neg \leq (y, z)) \ + \ tell(\leq (x, z))) \\ \qquad \parallel tell(done_{trans}(x, y, z)) \parallel transitivity) \\ + \exists x \exists y \exists z (tell(done_{trans}(x, y, z)) \rightarrow reflexivity) \end{array}$$

the other properties are defined in the same way.

**Proof of Theorem 3.5** We outline the simulation of PFP by LCC($\mathcal{FD}$). We describe the simulation of one application of the partial fixpoint operator, $\mu_r(\phi(r))$ where $\phi(r)$ is in FO (this is sufficient in view of the normal form for PFP stating that a single application of $\mu$ is sufficient [AV95]).

Consider the CC$_s(\mathcal{FD})$ program with flat guards simulating $\phi(r)$ on ordered structures in open constraint systems. An LCC($\mathcal{FD}$) program simulating $\phi(r)$ is obtained by replacing all constraints $c$ by $!c$ except $r$ which is defined as a linear constraint. We also have to add a $tell(r(\vec{x}))$ after each ask consuming an $r(\vec{x})$, and an ask $r(\vec{x}) \rightarrow True$ after each corresponding tell. We thus obtain two procedures $phi\_r(\vec{x})$ and $notphi\_r(\vec{x})$ encoding $\phi(r)$ and $\neg\phi(r)$. We also need a procedure $succ(x, y)$ defining the successor relation corresponding to $\leq$, and its straightforward extension to tuples of variables. We finally need a procedure $r\_equal\_s$ succeeding if the relations $r$ and $s$ are equal.

The following LCC($\mathcal{FD}$) program then simulates the partial fixpoint operator:

$$init :: \exists \vec{x} (min(\vec{x}) \parallel s\_is\_r(\vec{x}))$$

$$\begin{array}{lcl} s\_is\_r(\vec{x}) & :: & r(\vec{x}) \rightarrow \exists \vec{y} (succ(\vec{x}, \vec{y}) \\ & & \parallel tell(r(\vec{x}) \otimes s(\vec{x})) \\ & & \parallel s\_is\_r(\vec{y})) \end{array}$$

$$
\begin{array}{lll}
 & + & \overline{r}(\vec{x}) \rightarrow \exists \vec{y}(succ(\vec{x},\vec{y}) \\
 & & \parallel tell(\overline{r}(\vec{x}) \otimes \overline{s}(\vec{x})) \\
 & & \parallel s\_is\_r(\vec{y})) \\
 & + & max(\vec{x}) \parallel main
\end{array}
$$

initializes $s$ as $r$ when the first parameter given is the minimum.

$$
main :: \forall \vec{y}(r(\vec{y}) \rightarrow tell(\overline{r}(\vec{y})) \parallel loop)
$$

changes one slot of $r$ to be sure it is different of $s$ and begins the loop.

$$
\begin{array}{lll}
loop & :: & r\_equal\_s \\
 & + & \exists \vec{x}(min(\vec{x}) \parallel r\_is\_s(\vec{x}))
\end{array}
$$

tests if $r$ and $s$ are equal (end) or loops.

$$
\begin{array}{lll}
r\_is\_s(\vec{x}) & :: & s(\vec{x}) \otimes r(\vec{x}) \rightarrow \exists \vec{y}(succ(\vec{x},\vec{y}) \\
 & & \parallel tell(r(\vec{x}) \otimes s(\vec{x})) \parallel r\_is\_s(\vec{y})) \\
 & + & \overline{s}(\vec{x}) \otimes r(\vec{x}) \rightarrow \exists \vec{y}(succ(\vec{x},\vec{y}) \\
 & & \parallel tell(\overline{s}(\vec{x}) \otimes \overline{r}(\vec{x})) \parallel r\_is\_s(\vec{y})) \\
 & + & s(\vec{x}) \otimes \overline{r}(\vec{x}) \rightarrow \exists \vec{y}(succ(\vec{x},\vec{y}) \\
 & & \parallel tell(r(\vec{x}) \otimes s(\vec{x})) \parallel r\_is\_s(\vec{y})) \\
 & + & \overline{s}(\vec{x}) \otimes \overline{r}(\vec{x}) \rightarrow \exists \vec{y}(succ(\vec{x},\vec{y}) \\
 & & \parallel tell(\overline{s}(\vec{x}) \otimes \overline{r}(\vec{x})) \parallel r\_is\_s(\vec{y})) \\
 & + & max(\vec{x}) \parallel \exists \vec{y}(min(\vec{y}) \\
 & & \parallel s\_is\_phi(\vec{y}))
\end{array}
$$

assigns to $r$ the current value of $s$.

$$
\begin{array}{lll}
s\_is\_phi(\vec{x}) & :: & s(\vec{x}) \rightarrow \exists \vec{y}(succ(\vec{x},\vec{y}) \parallel phi\_r(\vec{x}) \\
 & & \parallel tell(s(\vec{x})) \parallel s\_is\_phi(\vec{y})) \\
 & + & s(\vec{x}) \rightarrow \exists \vec{y}(succ(\vec{x},\vec{y}) \parallel tell(\overline{s}(\vec{x})) \\
 & & \parallel notphi\_r(\vec{x}) \parallel s\_is\_phi(\vec{y})) \\
 & + & \overline{s}(\vec{x}) \rightarrow \exists \vec{y}(succ(\vec{x},\vec{y}) \parallel phi\_r(\vec{x}) \\
 & & \parallel tell(s(\vec{x})) \parallel s\_is\_phi(\vec{y})) \\
 & + & \overline{s}(\vec{x}) \rightarrow \exists \vec{y}(succ(\vec{x},\vec{y}) \parallel tell(\overline{s}(\vec{x})) \\
 & & \parallel notphi\_r(\vec{x}) \parallel s\_is\_phi(\vec{y})) \\
 & + & max(\vec{x}) \parallel loop)
\end{array}
$$

assigns to $s$ the value of $\phi$(r) and loops. □

**Proof of Theorem 4.6** It is sufficient to show (i), since (ii) follows from (i) and the fact that PFP expresses PSPACE on ordered structures. Consider (i). The simulation of PFP by $\mathrm{CC}_{ts}(\mathcal{FD}^{ind})$ is straightforward. The stages in the evaluation of partial fixpoints are defined inductively. A partial fixpoint is reached if and when two consecutive stages of are equal. Note that one cannot test directly if two consecutive stages of $t(\vec{x},i)$ are the same, since the condition $j = i+1$ cannot occur in the definition of a predicate. Instead, a new predicate $\overline{t}(\vec{x},\vec{y},i)$ is defined inductively so that for $i > 1$, $\overline{t}(\vec{x},\vec{y},i)$ holds iff $t(\vec{x},i)$ and $t(\vec{y},i-1)$ hold. This allows to test when two consecutive stages are equal.

Showing that $\mathrm{CC}_{ts}(\mathcal{FD}^{ind})$ is subsumed by PFP is less straightforward. Let $P$ be a $\mathrm{CC}_{ts}(\mathcal{FD}^{ind})$ program. The inductive definitions in $P$ are mimicked by a partial fixpoint. This requires using the Simultaneous Induction Lemma [GS86] showing that a simultaneous inductive definition of multiple relations can be simulated by an inductive definition of a single relation. A nontrivial subtlety involves detecting within PFP when the entire answer to $P$ has been computed, then forcing convergence. To explain how this is achieved, we will need the notion of $k$-size of a finite structure, introduced in [AV95]. The $k$-size of a finite structure $\mathbf{I}$ is the number of equivalence classes of tuples of arity at most $k$ over the finite domain of $\mathbf{I}$ defined by: $t \equiv_k t'$ iff $\mathbf{I} \models \varphi(t) \leftrightarrow \mathbf{I} \models \varphi(t')$ for every FO formula $\varphi$ with $k$ variables. Let $k$ be the number of non-integer variables used in the declarations of $P$. The simulation is based on the following key observations:

1. In order to compute the answer to $P$ on input $\mathbf{I}$ it is sufficient to compute the inductively defined processes for integers bounded by an exponential in the $k$-size of $\mathbf{I}$.

2. On input $\mathbf{I}$, PFP can simulate a counter up to an exponential in the $k$-size of $\mathbf{I}$.

Part (2) is based on a result showing that the equivalence classes of $\equiv_k$ and a total order on them can be defined by an IFP formula (and therefore by a PFP formula) [AV95, DLW95, KV92]. The ordered equivalence classes (whose number recall is the $k$-size of $\mathbf{I}$) allows to inductively define a counter up to an exponential in the $k$-size of $\mathbf{I}$. The counter allows to force convergence of the partial fixpoint within the PFP formula, as desired. □

**Proof of Theorem 4.7** Consider (i), and let $\mathcal{D}.A$ be a $\mathrm{CC}_{ts}(\mathcal{FD})$ program with flat guards. To show

that the query defined by $\mathcal{D}.A$ can be evaluated in NP, we first note that, to compute the answer to the query on input $\mathbf{I}$ it is sufficient to consider derivations of length polynomial in the size of $\mathbf{I}$. This is done by choosing an evaluation strategy where consecutive executions of non-equivalent *tell* agents are separated in the derivation by constant number of steps. Then it is sufficient to note that there are polynomially many non-equivalent *tell* agents, and that the length of derivations following the last *tell* is also polynomial, as it is isomorphic to the execution of a $\mathrm{CC}_{ts}(\mathcal{FD})$ program with flat guards with a complete constraint system.

To see that $\mathrm{CC}_{ts}(\mathcal{FD})$ with flat guards expresses all NP queries, we use the characterization of NP by $\exists$SO (existential second-order logic). Consider an $\exists$SO formula $\exists\ S\ \varphi(S)$ where $\varphi$ is in FO. This is simulated by a $\mathrm{CC}_{ts}(\mathcal{FD})$ program with flat guards as follows:

1. non-deterministically construct a relation $S$ using *tell* agents; Additionally, non-deterministically construct a total order $\leq$ on the finite domain.

2. simulate the evaluation of $\varphi(S)$.

To illustrate (1) note that guessing $S$ can be done by a program of the form:

$$t :: (\ \forall\ \vec{x}\ (tell(True) \rightarrow tell(S(\vec{x}))) \parallel t)\ +\ A$$

where $A$ uses the $S$ defined by the recursion. Guessing an order $\leq$ is similar, but in addition $A$ must verify that $\leq$ is total. Once $S$ and $\leq$ are constructed, the remainder of the execution treats them as completely defined structures. By Theorem 4.2, $\mathrm{CC}_{ts}(\mathcal{FD})$ with flat guards and complete constraint systems expresses P on ordered structures, and therefore can simulate $\varphi(S)$ making use of $\leq$.

The proof of (ii) is a straightforward extension of the proof of (i). Whereas flat guards express $\exists$SO, the nesting provided by stratified deep guards allows to express all of SO. The result then follows from the fact that SO expresses PHIER [S77].

Consider (iii). We first show that $\mathrm{CC}_{ts}(\mathcal{FD})$ with unrestricted deep guards expresses APTIME (alternating polynomial time), then use the result that APTIME = PSPACE [CKS81]. The simulation of a $\mathrm{CC}_{ts}(\mathcal{FD})$ program $P$ with unrestricted deep guards by an APTIME Turing machine uses the fact that the depth of nesting of guards in each derivation of $P$ is bounded by a polynomial in the finite domain, and that the number of non-equivalent *tell* instructions executed is bounded by the same. This yields the polynomial bound on the number of steps of the alternating Turing machine needed in the simulation. The alternation is used to simulate the execution of deep guards. For the converse, we use a result showing that a certain non-deterministic variant of IFP with alternation expresses APTIME on ordered structures [AVV97]. The nondeterminism stems from a choice of one among two $FO$ formulas $\varphi_1, \varphi_2$ to be used at each stage in the definition of a fixpoint. The final answer consists of the intersection of all possible fixpoints obtained by the different choices. Note that, since APTIME is closed under complement, we can equivalently take the semantics to be the union of the complements of all resulting fixpoints. Let us denote this variant of IFP by AIFP. The simulation of an alternating inflationary fixpoint defined by two FO inflationary formulas $\varphi_1, \varphi_2$ is done by a $\mathrm{CC}_{ts}(\mathcal{FD})$ program $\mathcal{D}.A$ where $A :: answer(x, y)$ and $\mathcal{D}$ is of the form

$$\begin{array}{lcl} answer(x,y) & :: & (t(x,y) \rightarrow tell(False)) \\ t(x,y) & :: & (\varphi_1(t)(x,y) + \varphi_2(t)(x,y) \\ & & +\varphi_1(\emptyset)(x,y) + \varphi_2(\emptyset)(x,y)) \parallel t \\ & & +\ \ fp(t, \varphi_1, \varphi_2) \end{array}$$

where $\varphi_i(t)(x, y)$ is a process simulating $\varphi_i$ applied to $t$, $i \in \{1, 2\}$, and $fp(t, \varphi_1, \varphi_2)$ is true iff $\varphi_i(t) = t$, $i \in \{1, 2\}$. □

## C: Semantics of CC

The operational semantics of CC is defined in the paper with references to table 1 for the structural congruence relation on configurations, and table 2, 4 or 5, 6 for the transition system.

The denotational semantics of a CC program $D.A$ associates to an initial store $\sigma \in \mathcal{C}$ a set of stores $[\![D.A]\!] \in \mathcal{P}(\mathcal{C})$. The denotational semantics of terminal stores with flat guards has been studied in [SRP91, BGMP97, BGP96]. We present here a generalization to the denotational semantics of CC languages with deep guards for the observation of both successes and terminal stores.

We consider the following operations on sets of stores:

$$\exists_x S = \{\exists x \sigma \mid \sigma \in S\}$$

$$S_1 \wedge S_2 = \{\sigma_1 \wedge \sigma_2 \mid \sigma_1 \in S_1,\ \sigma_2 \in S_2\}$$

The denotational semantics of success stores (resp. of terminal stores) denoted by $[\![A]\!]\sigma(e)$ (resp. $[\![A]\!]^{ts}\sigma(e)$) of a program $D.A$ is defined in Table 3.

| | |
|---|---|
| **Parallel** | $(\vec{x}; c; \Delta, A \parallel B, \Gamma) \equiv (\vec{x}; c; \Delta, B \parallel A, \Gamma) \equiv (\vec{x}; c; \Delta, A, B, \Gamma)$ |
| **Hiding** | $\dfrac{y \notin \vec{x} \cup fv(c, \Gamma)}{(\vec{x}; c; \exists y A, \Gamma) \equiv (\vec{x}, y; c; A, \Gamma)} \quad \dfrac{y \notin fv(c, \Gamma)}{(\vec{x}, y; c; \Gamma) \equiv (\vec{x}; c; \Gamma)}$ |
| **$\alpha$-Conversion** | $\dfrac{z \notin v(A)}{\exists y A \equiv \exists z A[z/y]}$ |

Table 1: CC($\mathcal{FD}$) structural congruence

| | |
|---|---|
| **Tell** | $(\vec{x}; c; tell(d), \Gamma) \longrightarrow (\vec{x}; c \wedge d; \Gamma)$ |
| **Ask** | $\dfrac{(\vec{x}; c; A[\vec{t}/\vec{y}]) \longrightarrow^* (\vec{z}; d; \epsilon) \quad \mathcal{S} \vdash \exists \vec{x} c \Leftrightarrow \exists \vec{z} d}{(\vec{x}; c; \forall \vec{y}(A \rightarrow B), \Gamma) \longrightarrow (\vec{x}; c; B[\vec{t}/\vec{y}], \Gamma)}$ |
| **Procedure calls** | $\dfrac{[p(\vec{y}) :: A]_\sim \in \mathcal{D}}{(\vec{x}; c; p(\vec{y}), \Gamma) \longrightarrow (\vec{x}; c; A, \Gamma)}$ |
| $\equiv$ | $\dfrac{(\vec{x}; c; \Gamma) \equiv (\vec{x}'; c'; \Gamma') \longrightarrow (\vec{y}'; d'; \Delta') \equiv (\vec{y}; d; \Delta)}{(\vec{x}; c; \Gamma) \longrightarrow (\vec{y}; d; \Delta)}$ |
| **Blind choice** | $(\vec{x}; c; A + B, \Gamma) \longrightarrow (\vec{x}; c; A, \Gamma)$<br>$(\vec{x}; c; A + B, \Gamma) \longrightarrow (\vec{x}; c; B, \Gamma)$ |

Table 2: CC($\mathcal{FD}$) transition relation.

**Example** We illustrate the computation of the success set of deep guards with three examples. Consider first:

```
A ::  tell(c) → B      B ::  tell(d) → tell(e)
```

In this flat guard example one can see that $GS_1$ will contain any store stronger than $e$ for `A` and `B`, and the expected stores for `tell(c)` (stronger than $c$), and `tell(d)` (stronger than $d$). From this lower approximation (already correct for the flat guards) we get $GS_2$ where we reach a fixpoint with the usual terminating stores: $(\emptyset; \sigma; A) \in GS$ iff $\sigma \vdash c \wedge d \wedge e$ or $\sigma \nvdash c$ or $\sigma \nvdash d$. $(\emptyset; \sigma; B) \in GS$ iff $\sigma \vdash d \wedge e$ or $\sigma \nvdash d$.

```
A ::  B → tell(c)      B ::  tell(d) → tell(e)
```

In this stratified deep guard example one can see that $GS_1$ will contain any store stronger than $c$ for `A` and stronger than $e$ for `B`, and the expected stores for `tell(d)`. We then get $(\emptyset; \sigma; A) \in GS_2$ iff $\sigma \vdash c \wedge e$ or $\sigma \nvdash e$. $(\emptyset; \sigma; B) \in GS_2$ iff $\sigma \vdash d \wedge e$ or $\sigma \nvdash d$. And finally $GS_3$ reaches the expected fixpoint: $(\emptyset; \sigma; A) \in GS$ iff $\sigma \vdash c \wedge d \wedge e$ or $\sigma \vdash c$ and $\sigma \nvdash d$ or $\sigma \vdash c$ and $\sigma \nvdash e$. $(\emptyset; \sigma; B) \in GS_2$ iff $\sigma \vdash d \wedge e$ or $\sigma \nvdash d$.

```
A ::  B → tell(c)      B ::  A → tell(c)
```

In this non-stratified deep guard example one can see that $GS_1$ will contain any store stronger than $c$ for `A` and `B`, thus $GS_2$ is $\mathcal{K}$ again and we get $GS = GS_1$. This defines $\longrightarrow$ such that $(\emptyset; d; A) \longrightarrow^* (\emptyset; d; \epsilon)$ if $d \vdash c$ and $(\emptyset; d; A) \longrightarrow^* (\emptyset; False; \epsilon)$ otherwise.

$$
\begin{array}{ll}
[\![D.tell(c)]\!](e)\sigma & = \{\sigma \wedge c\} \\
[\![D.\forall \vec{x}(A \rightarrow B)]\!](e)\sigma & = \bigcup_{\{\vec{t} \mid \sigma \in [\![D.A[\vec{t}/\vec{x}]]\!](e)\sigma\}} [\![D.B(\vec{t}/\vec{x})]\!](e)\sigma \\
[\![D.A \parallel B]\!](e)\sigma & = \bigcup_{\sigma_0 \in \overline{\sigma}} [\![D.A]\!](e)\sigma_0 \cap [\![D.B]\!](e)\sigma_0 \text{ where } \overline{\sigma} = \mu\Psi\sigma \\
 & \text{and } \Psi(f)\sigma = [\![A]\!]^{ts}(e)f(\sigma) \wedge [\![D.B]\!]^{ts}(e)f(\sigma) \\
[\![\exists x A]\!](e)\sigma & = \exists_x [\![D.A]\!](e)\sigma \\
[\![D.A + B]\!](e)\sigma & = [\![D.A]\!](e)\sigma \cup [\![D.B]\!](e)\sigma \\
[\![D.p(\vec{x})]\!](e)\sigma & = \mu\Psi \text{ where } \Psi(f) = [\![D \setminus \{p\}.A]\!](e\{f/p\})\sigma \\
[\![D.p(\vec{x})]\!](e)\sigma & = e(p) \text{ if } p \notin D \\
 & \\
[\![D.tell(c)]\!]^{ts}(e)\sigma & = \{\sigma \wedge c\} \\
[\![D.\forall \vec{x}(A \rightarrow B)]\!]^{ts}(e)\sigma & = \bigcup_{\vec{t}} (\mathcal{C} \setminus [\![D.A[\vec{t}/\vec{x}]]\!]^{ts}) \cap [\![D.B(\vec{t}/\vec{x})]\!]^{ts}(e)\sigma \\
[\![D.A \parallel B]\!]^{ts}(e)\sigma & = [\![D.A]\!]^{ts}(e)\sigma \cap [\![D.B]\!]^{ts}(e)\sigma \\
[\![\exists x A]\!]^{ts}(e)\sigma & = \exists_x [\![D.A]\!]^{ts}(e)\sigma \\
[\![D.A + B]\!]^{ts}(e)\sigma & = [\![D.A]\!]^{ts}(e)\sigma \cup [\![D.B]\!]^{ts}(e)\sigma \\
[\![D.p(\vec{x})]\!]^{ts}(e)\sigma & = \mu\Psi \text{ where } \Psi(f)\sigma = [\![D \setminus \{p\}.A]\!]^{ts}(e\{f/p\})\sigma \\
[\![D.p(\vec{x})]\!]^{ts}(e)\sigma & = e(p) \text{ if } p \notin D
\end{array}
$$

Table 3: CC($\mathcal{FD}$) denotational semantics for successes and terminal stores.

| | |
|---|---|
| **Linear Tell** | $(\vec{x}; c; tell(d), \Gamma) \longrightarrow_{LCC} (\vec{x}; c \otimes d; \Gamma)$ |
| **Linear Ask** | $\dfrac{c \vdash_{\mathcal{C}} d \otimes e[\vec{t}/\vec{y}]}{(\vec{x}; c; \forall \vec{y}(e \rightarrow A), \Gamma) \longrightarrow_{LCC} (\vec{x}; d; A[\vec{t}/\vec{y}], \Gamma)}$ |

Table 4: Linear tell and ask

| | |
|---|---|
| **Tell** | $(\vec{x}; c; tell(d), \Gamma) \longrightarrow_{GS} (\vec{x}; c \wedge d; \Gamma)$ |
| **Ask** | $\dfrac{(\vec{x}; c; A[\vec{t}/\vec{y}]) \in GS}{(\vec{x}; c; \forall \vec{y}(A \rightarrow B), \Gamma) \longrightarrow_{GS} (\vec{x}; c; B[\vec{t}/\vec{y}], \Gamma)}$ |
| **Procedure calls** | $\dfrac{[p(\vec{y}) :: A]_{\sim} \in \mathcal{D}}{(\vec{x}; c; p(\vec{y}), \Gamma) \longrightarrow_{GS} (\vec{x}; c; A, \Gamma)}$ |
| $\equiv$ | $\dfrac{(\vec{x}; c; \Gamma) \equiv (\vec{x}'; c'; \Gamma') \longrightarrow_{GS} (\vec{y}'; d'; \Delta') \equiv (\vec{y}; d; \Delta)}{(\vec{x}; c; \Gamma) \longrightarrow_{GS} (\vec{y}; d; \Delta)}$ |
| **Blind choice** | $(\vec{x}; c; A + B, \Gamma) \longrightarrow_{GS} (\vec{x}; c; A, \Gamma)$<br>$(\vec{x}; c; A + B, \Gamma) \longrightarrow_{GS} (\vec{x}; c; B, \Gamma)$ |

Table 5: $\mathrm{CC}_{ts}(\mathcal{FD})$ transition relation relative to a guard success set $GS$.

$GS_0 = \mathcal{K}$
$GS_i = \{(\vec{x}; c; A) \mid (\vec{x}; c; A) \longrightarrow^*_{GS_{i-1}} (\vec{y}; d; \Gamma) \not\longmapsto_{GS_{i-1}} \text{ and } \mathcal{S} \vdash \exists \vec{x} c \Leftrightarrow \exists \vec{y} d\}$
$GS = \bigcup_{i \geq 0} GS_{2i+1}$
$GF = \bigcup_{i \geq 0} \overline{GS_{2i}}$
$\longrightarrow = \longrightarrow_{GS} \cup \{(\vec{x}; c; \forall \vec{y}(A \rightarrow B), \Gamma) \longrightarrow (\emptyset; False; \epsilon) \mid \exists \vec{t}\, (\vec{x}; c; A[\vec{t}/\vec{y}]) \in \overline{GS \cup GF}\}$

Table 6: Inductive definition of the transition relation for $\mathrm{CC}_{ts}(\mathcal{FD})$ with deep guards.