Coinductive Constraint Logic Programming

Neda Saeedloei and Gopal Gupta

Department of Computer Science,
University of Texas at Dallas,
Richardson, TX 75080, USA
{neda.saeedloei,gupta}@utdallas.edu

Abstract. Constraint logic programming (CLP) has been proposed as a declarative paradigm for merging constraint solving and logic programming. Recently, coinductive logic programming has been proposed as a powerful extension of logic programming for handling (rational) infinite objects and reasoning about their properties. Coinductive logic programming does not include constraints while CLP's declarative semantics is given in terms of a least fixed-point (i.e., it is inductive) and cannot directly support reasoning about (rational) infinite objects and their properties. In this paper we combine constraint logic programming and coinduction to obtain *co-constraint logic programming* (*co-CLP* for brevity). We present the declarative semantics of co-CLP in terms of a greatest fixed-point and its operational semantics based on the *coinductive hypothesis rule*. We prove the equivalence of these two semantics for programs with rational terms.

1 Introduction

Constraint logic programming [7] is a natural and expressive paradigm which combines two declarative paradigms: logic programming [9] and constraint solving. CLP's declarative semantics is defined in terms of a least fixed-point, i.e., it is inductive. Therefore, it cannot be directly used for reasoning about (rational) infinite objects and their properties. Coinductive logic programming (co-LP) has recently been proposed as a powerful technique for finitely reasoning about (rational) infinite structures and their properties [14, 13], however, it does not support constraints. Constraints and (rational) infinite computations arise together in many interesting applications such as verification of real-time and *cyber-physical* systems [8, 4]. A formalism that can support both constraints and computations over (rational) infinite structures is needed. This paper focuses on developing such a formalism.

Consider the following program, which describes an infinite list whose pairwise consecutive elements obey a constraint. This program is not semantically meaningful either in constraint logic programming or in coinductive logic programming.

```
stream([X, Y | T]) :- {Y - X >= 3}, stream(T).
```

However, we would like a query such as `?- stream(L).` to return a positive answer in a finite amount of time. The operational semantics of CLP does not allow for infinite proofs such as a proof for the query above and the operational semantics of co-LP does not allow constraints. While this is only an artificial,

simple example to illustrate the point, there are many real world applications that (i) exhibit infinite behavior, and (ii) interact with the environment while operating under constraints imposed by physical entities such as time, temperature, pressure, etc. For example a reactor temperature control system [6] is a traditional example of a cyber-physical system which exhibits both aspects mentioned above: one of the components of this system, the core controller, runs forever in order to keep the temperature of the system between two thresholds. Two physical quantities involved in this system are time and temperature, and the behavior of the system is specified by placing constraints on them. Another example is the generalized railroad crossing problem [5]. This system is controlled by several components: controller, gate and tracks that work in parallel and run forever. The behavior of this system is also specified by constraints between the times at which different events in the system take place. These systems can be naturally modeled as *coinductive constraint logic programs*.

In the past we explored applications that involved both CLP and co-LP [11, 12], however, lack of a formalism that combines both hindered progress. We develop such a formalism in this paper that combines CLP and co-LP. We extend the constraint logic programming scheme to coinductive constraint logic programming (co-CLP for brevity). Co-CLP allows direct reasoning over (rational) infinite objects and their properties when constraints are used to express the relation between the objects. We present the declarative and operational semantics of co-CLP and show their equivalence for programs with rational structures.

Co-CLP allows predicates to be annotated as either inductive or coinductive. Inductive predicates can invoke coinductive predicates and vice versa; however, no cycles through alternating induction and coinduction are allowed. The declarative semantics of co-CLP is defined in terms of greatest fixed-point semantics. The operational semantics is defined as an extension of the top-down execution model of CLP with a coinductive hypothesis rule. This operational semantics allows us to obtain finite derivations for (rational) infinite proofs. In this paper, we limit our attention to coinductive constraint logic programs without negation.

The contributions of this paper include developing a new paradigm, called co-CLP, which combines traditional constraint logic programming and coinductive logic programming, as well as its declarative and operational semantics. However, unlike the traditional CLP, user-defined functions are allowed in co-CLP. Co-CLP allows for logic programming with finite and (rational) infinite data structures in the presence of constraints; that is, co-CLP can be used directly to reason about logical statements regarding both finite and (rational) infinite objects and their properties with constraints imposed on them. We first present an overview of coinductive logic programming [14].

2 Coinductive Logic Programming

Standard logic programming (which computes least fixed-points) cannot be used to reason about infinite objects (which belong to the greatest fixed-points). Recently *coinduction* [1] has been introduced into logic programming by Simon et al [14] to overcome this problem. Coinductive logic programming can be used for reasoning about unfounded sets, behavioral properties of (interactive) pro-

grams, etc., as well as elegantly proving liveness properties in model checking, type inference in functional programming, etc. [3].

Coinductive logic programming is a dual of logic programming: it supports reasoning directly about infinite objects and infinite properties. The declarative semantics of coinductive logic programming allows the universe of terms to contain infinite terms, in addition to the traditional finite terms. It also allows for the model to contain ground goals that have either finite or infinite *idealized* proofs. Coinductive logic programming provides an operational semantics—similar to SLD resolution [15]—for computing the greatest fixed-point of a logic program. This operational semantics (called co-SLD resolution) relies on the *coinductive hypothesis rule* and systematically computes elements of the *gfp* via backtracking. It is briefly described below through a simple example. Consider a normal logic programming definition of a stream (list) of bits given as program P1 below:

```
stream([]).
stream([H | T]) :- bit(H), stream(T).
bit(0).
bit(1).
```

Under SLD resolution, the query `?- stream(X).` will systematically produce all finite streams of bits one by one, starting from the empty stream `[]`. Suppose we remove the base case and call the resulting program P2. In standard logic programming the query `?- stream(X).` fails, since the lfp model of P2 does not contain any instances of `stream/1`. The problems are two-fold: (i) the Herbrand universe does not contain infinite terms, (ii) the least Herbrand model does not allow for infinite proofs, such as the proof of `stream(X)`; yet these concepts are commonplace in computer science, and a sound mathematical foundation exists for them in the field of hyperset theory [1]. Coinductive logic programming extends the traditional declarative and operational semantics of LP to allow reasoning over infinite and cyclic structures and properties. In the coinductive logic programming paradigm the declarative semantics of the predicate `stream/1` above is given in terms of *infinitary Herbrand* (or *co-Herbrand*) *universe, infinitary Herbrand* (or *co-Herbrand*) *base* [9], and *maximal models* (computed using greatest fixed-points) [14].

The query `?- stream(X).` under the coinductive interpretation of P2 should produce all infinite sized streams as answers, e.g.,

```
X = [1, 1, 1, ... ],
X = [1, 0, 1, 0, ... ],
```

etc. The model of P2 does contain instances of `stream/1` (but proofs may be of infinite length).

If we take a coinductive interpretation of program P1, then we get all finite and infinite streams as answers to the query `?- stream(X).` Coinductive logic programming allows programmers to manipulate infinite structures. As a result, unification must be extended and the "occurs check" [9] removed: unification equations such as `X = [1 | X]` are allowed in coinductive logic programming.

The operational semantics under coinduction is given in terms of the *coinductive hypothesis rule*: during execution, if the current resolvent R contains a call C' that unifies with an ancestor call C, then the call C' succeeds; the new

resolvent is $R'\theta$ where $\theta = mgu(C, C')$ and R' is obtained by deleting C' from R. With this extension, a clause such as

```
p([1 | T]) :- p(T).
```

and the query `?- p(Y).` will produce an infinite answer `Y = [1 | Y]`.

In coinductive logic programming the alternative computations started by a call are not only those that begin with unifying the call and the head of a clause, but also those that begin with unifying the call and one of its ancestors in a proof tree.

Regular logic programming execution extended with the coinductive hypothesis rule is termed *co-logic programming* [13]. The coinductive hypothesis rule works for only those infinite proofs that are *regular* (or *rational*), i.e., infinite behavior is obtained by a finite number of finite behaviors interleaved an infinite number of times. More general implementations of coinduction are possible [13]. More complex examples of coinductive LP can be found elsewhere [14]. Even with the restriction to rational proofs, there are many applications of coinductive logic programming. These include model checking, modeling ω-automata, non-monotonic reasoning, etc [3].

3 Notation and Terminology

In the rest of the paper we follow the notations and terminologies that are used in [7] and [13]. However, in traditional constraint logic programming [7], the only domain considered is the domain of constraints; therefore, defining new function symbols and terms is not allowed. As a result, the terms and atoms that are used in a constraint logic program are limited to what is defined in Σ (defined below), and terms and atoms are interpreted in the domain of constraints. In this paper, we consider not only the domain of constraints, but also the Herbrand universe. Therefore, user-defined terms will be allowed. User-defined functions and predicates will be given the standard interpretation in the Herbrand universe and the Herbrand base. However, constraints will be interpreted using a predefined interpretation (provided by the domain of computation).

A *signature* defines a set of function and predicate symbols with their corresponding arities. If Σ is a signature, a *Σ-structure* $\mathcal{D}$ consists of a set D and an assignment of functions and relations on D to the symbols of Σ which respects the arities of the symbols (a Σ-structure $\mathcal{D}$ is an interpretation of Σ in domain D). Note that D may contain finite as well as infinite objects.

We assume an infinite number of variables; x will denote a variable, h, t will denote terms, p will denote a predicate symbol, a will denote an atom, c will denote a constraint, r will denote a rule, and P will denote a program. $\tilde{x}$ and $\tilde{h}$ denote sequences of terms $x_1, x_2, \ldots, x_n$ and $h_1, h_2, \ldots, h_n$, respectively. $\mathcal{D}$ will denote a structure, D will denote its set of elements. $\exists_{-\tilde{x}}\ \phi$ denotes the full existential closure of the formula ϕ except for the variables $\tilde{x}$, which remain unquantified. $\tilde{\exists}\ \phi$ denotes the full existential closure of the formula ϕ. $\mathcal{D} \models c$ denotes that c is true in $\mathcal{D}$. $\mathcal{D} \models \exists c$ means that the constraint c is satisfiable in $\mathcal{D}$.

A first-order *Σ-formula* is built upon variables, function and predicate symbols of Σ, the logical connectives $\wedge, \vee, \neg, \leftarrow, \rightarrow, \leftrightarrow$ and quantifiers over variables

$\exists, \forall$. A closed formula is a formula in which all variable occurrences are bound by quantifiers. A Σ-*theory* is a collection of closed Σ-formulas.

Analogously to the class of CLP languages, co-CLP($\mathcal{X}$) can be obtained by instantiating the parameter $\mathcal{X}$, where $\mathcal{X}$ is an abbreviation for a 4-tuple $(\Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T})$. Σ is a signature containing the binary predicate symbol $=$ interpreted as identity in D, and a special binary relation $=_u$ which will be used for unification. The expression $x =_u a$ will be interpreted as substituting the term a for variable x. $\mathcal{D}$ is a Σ-structure, $\mathcal{L}$ is a class of Σ-formulas that contains a constraint that is identically true and one that is identically false, and $\mathcal{T}$ is a first-order Σ-theory. $\mathcal{L}$ is assumed to be closed under variable renaming, conjunction and existential quantification. For any signature Σ, a Σ-structure $\mathcal{D}$ (the domain of computation) and a class of Σ-formulas $\mathcal{L}$ (the constraints), the pair $(\mathcal{D}, \mathcal{L})$ is called a *constraint domain*. Sometimes the constraint domain is denoted by $\mathcal{D}$. A Σ-structure $\mathcal{D}$ is a model of a Σ-theory $\mathcal{T}$ if all formulas of $\mathcal{T}$ evaluate to *true* under the interpretation provided by $\mathcal{D}$. We limit our attention to such $\mathcal{T}$'s whose satisfiability of constraints is decidable in $\mathcal{D}$, in other words, for a constraint domain $(\mathcal{D}, \mathcal{L})$ with signature Σ, $\mathcal{T}$ is a decidable theory, i.e., (i) $\mathcal{D}$ is a model of $\mathcal{T}$, (ii) for every constraint $c \in \mathcal{L}$, either $\mathcal{T} \models \exists c$ or $\mathcal{T} \models \neg\exists c$ (i.e., each model of $\mathcal{T}$ is a model of $\exists c$ or each model of $\mathcal{T}$ is a model of $\neg\exists c$). This second property is called *satisfaction completeness*.

A *term* is a tree where every leaf node is labeled with a variable or with some $f/0$ (constant) and every inner node with n children, $n > 0$, is labeled with some f/n. We write $f(t_1, \ldots, t_n)$ for the term with root f and direct subtrees $t_1, \ldots, t_n$. A term t is called *finite* if all paths in the tree t are finite, otherwise it is *infinite*. A term (tree) is *rational* if it only contains finitely many different subterms (subtrees).

We consider an extension of Σ to Σ^D in which there is a constant for every element of D. Terms that are built upon variables and function symbols in Σ^D are denoted by T^D. Let $\mathcal{L}^D$ be a class of Σ^D-formulas [1]. Moreover, let Σ_P^D be the extension of Σ^D with the set of all user defined function and predicate symbols in program P. The set of predicate symbols defined by Σ is denoted by Π_c and the set of predicate symbols defined by a program is denoted by Π_P. Note that Π_c and Π_P are disjoint.

An *atom* is a tree $p(t_1, \ldots, t_n)$, where $p \in \Pi_p$ and $t_1, \ldots, t_n$ are rational terms that are built upon variables and function symbols in Σ_P^D. A *primitive constraint* is an expression of the form $p(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms from T^D and p is a predicate symbol from Π_c. In other words, primitive constraints only contain functors from Σ. This will ensure that the constraint solver will only deal with primitive constraints which it can solve. A *constraint* is a first-order formula built from primitive constraints.

Example 1. Let Σ contain a collection of constants, the binary predicate $=$, a binary function symbol '.' and a special constant *nil*. Let D be the set of finite as well as (rational) infinite lists. Let $\mathcal{D}$ interpret the function symbol '.' of Σ as the list constructor, where '.' maps a flat list L to a binary tree whose root and inner nodes are labeled with '.' and all elements of L appear as leaves of

[1] In the literature Σ^D and $\mathcal{L}^D$ are often denoted as Σ^* and $\mathcal{L}^*$.

the binary tree. The empty list is represented by *nil*. The primitive constraints are equations between terms. Let $\mathcal{L}$ be the class of constraints generated by these primitive constraints. Then $(\mathcal{D}, \mathcal{L})$ is the constraint domain of finite and (rational) infinite lists.

Example 2. Let Σ contain two binary predicate symbols $=$ and $<$, and a binary function symbol $+$. Let D be the set of natural numbers and $\mathcal{D}$ interpret the symbols of Σ as usual (i.e., $+$ is interpreted as addition, etc). Let f be a unary function symbol defined in the program. Then, for example, $7 + 2$ is a term in T^D, but not a formula; however, $7 + 2 < 8$ is a formula in $\mathcal{L}^D$, which is a legal constraint; however, $f(2) < 7$ is not allowed.

3.1 Syntax

Co-CLP's syntax is not identical to that of constraint logic programming, though it is very similar. It extends the syntax of CLP by allowing predicate symbols to be declared as being either inductive or coinductive. In analogy with co-LP [13], the stratification restriction in co-CLP would not allow the inductive and coinductive predicates to be mutually recursive; that is, execution cannot loop between inductive and coinductive calls. This restriction is for preventing ambiguity in the semantics of co-CLP programs. Without this restriction, it would not be possible to assign meaning to the programs that simultaneously exhibit the semantics of both inductive and coinductive constraint logic programming.

A *co-constraint logic program* is a collection of rules of the form $a \leftarrow c, b_1, \ldots, b_n$. a is an atom and called the head of the rule. $c, b_1, \ldots, b_n$ is called the body of the rule; c is the conjunction of constraints in the body and b_i's are atoms. An *annotated program* is a co-constraint logic program extended with a set of declarations. The set of declarations specify for every predicate p in program P, whether p is inductive or coinductive. In the annotated program corresponding to program P, for every inductive predicate q, there is a declaration of the form `:-inductive(q)`, and for every coinductive predicate p, there is a declaration of the form: `:-coinductive(p)`.

Example 3. In the following program the predicate p is a coinductive predicate (note that there is no base case). This predicate generates/accepts (rational) infinite lists of numbers in which, the *difference* between elements of lists with some constant is less than 5.

```
:-coinductive(p).
p([X | T], Y) :- {X - Y =< 5}, p(T, Y).
p([X | T], Y) :- {Y - X =< 5}, p(T, Y).
```

Definition 1. *Let P be an annotated program. We say that a predicate p directly depends on a predicate q if and only if $p = q$ or P contains a clause $p \leftarrow c, b_1, \ldots, b_n$. such that some b_i contains q. The dependency graph of program P has the set of its predicates as vertices, and the graph has an edge from p to q if and only if p directly depends on q.*

Definition 2. *The reduced graph for a co-constraint logic program has vertices consisting of the strongly connected components of the dependency graph of P. There is an edge from vertex u_1 to vertex u_2 in the reduced graph if and only*

if some predicate in u_1 depends directly on some predicate in u_2. A vertex in a reduced graph is said to be coinductive (resp. inductive) if it contains only coinductive (resp. inductive) predicates.

3.2 Declarative Semantics

In this section, we present the declarative semantics of co-constraint logic programming in terms of a greatest fixed-point. Since we are dealing with two types of predicate and function symbols, namely those that are defined by Σ and those that are defined by the user in program P, we must define a new interpretation that handles both. First, we extend the Herbrand universe of a program P to contain the domain D ($D \subset U_P$). We assume that the Herbrand Universe contains both finite and (rational) infinite terms [2]. User-defined terms and atoms will be given the standard interpretation in the Herbrand universe and the Herbrand base of P (U_P and B_P) [9]; while terms and formulas built from elements of Σ are given the interpretation provided by $\mathcal{D}$.

We denote the interpretation of a predicate symbol p defined in program P by $\mathbf{p}$. Similarly, the interpretation of a user-defined function symbol f will be denoted by $\mathbf{f}$. We take $v : V \to U_P$ to be a valuation of variables to ground terms in the Herbrand universe [9], such that if variable x occurs in a constraint, then $v(x) \in D$. We consider the natural extension of v that maps user-defined terms and atoms over the Herbrand universe and Herbrand base, terms in T^D to ground terms, and formulas in $\mathcal{L}^D$ to ground $\mathcal{L}^D$-formulas. We call this valuation a *constraint consistent valuation*. For an atom a of the form $p(t_1, \ldots, t_n)$ in P, $v(a)$ is $\mathbf{p}(v(t_1), \ldots, v(t_n))$. For a user-defined function of the form $f(t_1, \ldots, t_n)$, $v(f)$ is $\mathbf{f}(v(t_1), \ldots, v(t_n))$.

Definition 3. *A $\mathcal{D}$-base of a co-CLP program P over the domain D, denoted by $B_P^{\mathcal{D}}$, is the set*

$\{\boldsymbol{p}(x_1, \ldots, x_n) \mid p$ *is an n-ary user-defined predicate in $P$and each x_i is an element in U_P*$\}$

Definition 4. *A $\mathcal{D}$-interpretation of a formula in $\mathcal{L}^D$ is an interpretation that agrees with $\mathcal{D}$ on the interpretation of the symbols in Σ.*

Definition 5. *A $\mathcal{D}$-interpretation of a fact of the form $p(\tilde{x}) \leftarrow c$, is the set*

$\{\boldsymbol{p}(\tilde{h}) \mid$ *there exists a constraint consistent valuation v, such that $\mathcal{D} \models v(c)$, $v(\tilde{x}) = \tilde{h}$*$\}$

Since the set of predicates in program P is stratified into a collection of mutually disjoint strata of predicates, a vertex in a reduced graph of P is called a stratum. All the vertices in a stratum are of the same kind, either coinductive or inductive. A stratum u depends on stratum u', if there is an edge from u to u' in the reduced graph. A stratum u is said to be lower than stratum u' (u' is higher than u), if there is a path from u to u' in the reduced graph. A stratum u is said to be strictly lower than stratum u' (u' is strictly higher than u), if u is lower than u' and $u \neq u'$.

The model of each stratum, i.e., the model of a vertex in the reduced graph of a co-CLP is defined using the $T_{u,P}^{\mathcal{D}}$ operator below. Since a co-constraint logic program is stratified, its reduced graph is always a DAG. Moreover, all predicates

in the same stratum are the same kind, coinductive or inductive. Therefore, when $T^{\mathcal{D}}_{u,P}$ is used, the atoms defined as true in lower strata are treated as facts when proving atoms containing predicates in the current stratum.

Definition 6. *Let P be a co-constraint logic program over the domain of computation $\mathcal{D}$. Let I be a $\mathcal{D}$-interpretation and $T^{\mathcal{D}}_P : 2^{B^{\mathcal{D}}_P} \rightarrow 2^{B^{\mathcal{D}}_P}$. If R is the union of the models of strata that are strictly lower than stratum u, then:*

$T^{\mathcal{D}}_{u,P}(I) = \ R \ \cup \ \{\boldsymbol{p}(\tilde{h}) \mid p(\tilde{x}) \leftarrow c, b_1, \ldots, b_n$ *is a rule of P, $p \in u$, and there exists a constraint consistent valuation v such that $\mathcal{D} \models v(c), v(\tilde{x}) = \tilde{h}, v(b_i) \in I, i = 1, \ldots, n\}$*

The model of a stratum u of P is the least fixed-pint of $T^{\mathcal{D}}_{u,P}$ if u is inductive, and the greatest fixed-point of $T^{\mathcal{D}}_{u,P}$ if u is coinductive.

Definition 7. *The model of a co-CLP program P over a constraint domain $\mathcal{D}$, written $M^{\mathcal{D}}(P)$, is the union of the models of every stratum of P.*

The model of a co-constraint logic program is built by creating the inductive model for inductive strata, starting the lower strata, and the coinductive model for coinductive strata. This process is repeated from the lowest strata up to the topmost strata. Note that lowest and topmost strata exist since the program is stratified. Note also that the set of $\mathcal{D}$-interpretations forms a complete lattice under the subset ordering. $T^{\mathcal{D}}_{u,P}$ is a monotonic function on a complete lattice. Therefore the least and greatest fixed-points exist for $T^{\mathcal{D}}_{u,P}$.

The model of a co-constraint logic program can be defined using the set of idealized proofs defined below. Intuitively, idealized proofs are trees of ground atoms, such that a parent is deduced from the idealized proofs of its children. An idealized proof is *rational* if it is represented as a rational tree.

Definition 8. *The set of idealized proofs of a stratum u of P over domain $\mathcal{D}$ equals $\mu\Gamma^{\mathcal{D}}_{u,P}$ if u is inductive and $\nu\Gamma^{\mathcal{D}}_{u,P}$ (μ and ν correspond to the least and greatest fixed-points) if u is coinductive, such that if R is the union of the sets of idealized proofs of the strata strictly lower than u then*

$\Gamma^{\mathcal{D}}_{u,P}(S) = R \cup \{node(\boldsymbol{p}(\tilde{h}), T_1, \ldots, T_n) \mid p(\tilde{x}) \leftarrow c, b_1, \ldots, b_n$ *is a rule of P, $p \in u$, and there exists a constraint consistent valuation v such that $\mathcal{D} \models v(c)$, $v(\tilde{x}) = \tilde{h},\ T_i \in S, T_i(\epsilon) = v(b_i),\ i = 1, \ldots, n\}$*

Definition 9. *The set of idealized proofs generated by a co-constraint logic program P over a constraint domain $\mathcal{D}$, denoted by $\Gamma^{\mathcal{D}}_P$, is the union of the sets of idealized proofs of every stratum of P.*

We next define the declarative semantics of co-constraint logic programs in terms of idealized proofs. If $S = \{a \mid \exists T \in \Gamma^{\mathcal{D}}_P, a$ is the root of $T\}$, then $S = M^{\mathcal{D}}(P)$. Any element in the model has an idealized proof and anything that has an idealized proof is in the model. This formulation will be used in soundness and completeness proofs of operational semantics of co-CLP in order to distinguish between the cases when a query has a finite derivation, and when there are only infinite derivations of the query.

3.3 Operational Semantics

The operational semantics given for co-CLP is similar to that of CLP in the sense that it is presented in terms of state transition systems. However, there

are two major differences between these two semantics: (i) While the operational semantics of CLP uses multisets of atoms and constraints along with multisets of constraints as states, co-CLP uses forests of finite trees of atoms along with multisets of constraints as states. (ii) Operational semantics of CLP allows program rules (clauses) along with transition rules that manipulate constraints as transition rules. The operational semantics of co-CLP, in addition to these transitions, allows the coinductive hypothesis rule. The coinductive hypothesis rule is used to obtain finite derivations for goals with (rational) infinite idealized proofs. Intuitively, it states that if, during execution, a call A is encountered and unified with an ancestor call A', and moreover *the set of accumulated constrains is consistent*, then A is deleted from the resolvent and the substitution resulting from unification of arguments of A and A' is applied to the resolvent. Note that each ancestor of a goal constitutes a coinductive hypothesis, which can be accessed in the recursion stack. Note also that in co-CLP, unification must be extended and the "occurs check" removed: unification equations such as $X =_u [1 \mid X]$ are allowed; in fact, such equations will be used to represent (rational) infinite structures in a finite manner.

Definition 10. *Let N^* denote the set of all finite sequences of positive integers, including the empty sequence ϵ. We call these sequences paths. A singleton path is denoted by i for some positive integer i. The concatenation of two paths π and π' is denoted by $\pi.\pi'$. $D \subseteq N^*$ is prefix-closed, if $\pi_1.\pi_2 \in D$ implies $\pi_1 \in D$. A tree of elements of a set S, called an S-tree, is defined as a partial function from paths to elements of S, such that the domain is non-empty and prefix-closed. Each node in the tree is unambiguously denoted by a path. The root of tree t is denoted by $t(\epsilon)$; the node defined by a path π is denoted by $t(\pi)$. A tree t is described by the paths π from the root $t(\epsilon)$ to the nodes $t(\pi)$ of the tree, and the nodes are labeled with elements of S. A child of node π in tree t is any path $\pi.i$ that is in the domain of t, where i is some positive integer. The subtree of t rooted at π, denoted by $t' = t\backslash\pi$, is the partial function $t'(\pi') = t(\pi.\pi')$. node$(L, T_1, \ldots, T_n)$ is a constructor of an S-tree with root labeled L and subtrees T_i, where $L \in S$ and each T_i is an S-tree, such that $1 \leq i \leq n$, node$(L, T_1, \ldots, T_n)(\epsilon) = L$, and node$(L, T_1, \ldots, T_n)(i.\pi) = T_i(\pi)$.*

Definition 11. *A state is a triple (F, E, C), where*

- *F is a finite multiset of finite trees with nodes labeled with atoms,*
- *E is the set of equalities of the form $x =_u t$, in which x is a variable and t is a term and each variable x appears at most once in the left hand side of equations in E,*
- *C is the set of constraints from $\mathcal{L}^D$, it is called the constraint store.*

There is one other state, called the failure state.

Intuitively, the collection of as-yet-unseen atoms is represented by forest F. When an atom labeling a leaf in tree T of forest F is proved to be true, T is modified (using function δ, defined below) by removing the leaf and the maximum number of its ancestors, such that the result is still a tree. If all nodes in a tree are removed, the tree itself is removed. C is the collection of accumulated constraints built from the function and relation symbols of Σ, and E is the system of equations that captures the accumulated results of unification.

Definition 12. *[13] Given a tree of atoms T and a path π in the domain of T, let $\rho(T, \pi.i)$ be the partial function equal to T, except that it is undefined at $\pi.i$, Then*

$$\begin{array}{ll} \delta(T,\pi) = & T \quad \text{, if } \pi \text{ has children in } T \\ \delta(T,\epsilon) = & \emptyset \quad \text{, if } T \text{is empty} \\ \delta(T,\pi) = \delta(\rho(T,\pi),\pi') & \text{, if } \pi = \pi'.i \text{ is a leaf in } T \end{array}$$

The operational semantics of co-CLP relies on three types of transitions: (i) transitions that arise from resolution, (ii) transitions that arise from coinductive hypothesis rules, and (iii) transitions that manipulate the collection of constraints in constraint store. These transitions include adding constraints to constraint store, checking the consistency of current constraints in constraint store and inferring a new collection of constraints from the current collection.

Definition 13. *A transition rule r of a co-constraint logic program P is one of the following four forms:*

- *an instance of a clause in P, with variables consistently renamed for freshness,*
- *a coinductive hypothesis of the form $\eta(\pi, \pi')$: π and π' are paths such that π is a proper prefix of π', π is labeled by $p(t_1, \ldots, t_n)$, π' is labeled by $p(t'_1, \ldots, t'_n)$, $t_1, \ldots, t_n$ and $t'_1, \ldots, t'_n$ are unifiable,*
- *an inference rule for simplifying the current collection of constraints; the function* infer(C, E) *takes the current set of constraints and current substitution and computes a new set of constraints C', provided that $\mathcal{D} \models C \leftrightarrow C'$,*
- *a consistency check rule; the predicate* consistent(C) *expresses a test for consistency of C. It can be defined by: if $\mathcal{D} \models \tilde{\exists} C$ then* consistent(C) *holds.*

Definition 14. *Let T be a tree in forest F. A state (F, E, C) transitions to another state $((F \backslash \{T\}) \cup F', E', C')$ by transition rule r of program P when*

- *r is a definite clause of the form $p(t'_1, \ldots, t'_n) \leftarrow c, b_1, \ldots, b_m$, renamed to new variables, π is a leaf in T, $T(\pi) = p(t_1, \ldots, t_n)$. Let E'' be the substitution obtained from unification of $t_1, \ldots, t_n$ and $t'_1, \ldots, t'_n$, then $E' = E \cup E''$, $C' = C \cup c$ and F' is obtained as follows:*
  - *$m = 0$: $F' = \{\delta(T, \pi)\}$.*
  - *$m > 0$ and p is coinductive: $F' = \{T'\}$ where T' is equal to T except at $\pi.i$ and $T'(\pi.i) = b_i$, for $1 \leq i \leq m$.*
  - *$m > 0$ and p is inductive:*
    - *$\pi = \epsilon$: $F' = \{node(b_i) \mid 1 \leq i \leq m\}$.*
    - *$\pi = \pi'.j$ for some positive integer j : Let T'' be equal to T except at $\pi'.k$ for all $k \geq j$, where T'' is undefined. $F' = \{T'\}$ where T' is equal to T'' except at $\pi'.i$, where $T'(\pi'.i) = b_i$, for $1 \leq i \leq m$, and $T'(\pi'.(j - 1 + m + k)) = T(\pi'.k)$, for $k > j$.*
- *r is of the form $\eta(\pi, \pi')$, p is a coinductive predicate, π is a proper prefix of π', which is a leaf in T, $T(\pi) = p(t'_1, \ldots, t'_n)$, $T(\pi') = p(t_1, \ldots, t_n)$. Let E'' be the substitution obtained from unification of $t_1, \ldots, t_n$ and $t'_1, \ldots, t'_n$, then $E' = E \cup E''$, $C' = C \cup c$ and $F' = \{\delta(T, \pi')\}$.*
- *r is an inference rule, $F' = \{T\}$, $E' = E$ and $C' = infer(C, E)$.*

– *r is a consistency check rule, if consistent*(C) *holds then* $F' = \{T\}$, $E' = E$ *and* $C' = C$, *if* $\neg$*consistent*(C) *holds then* $((F \backslash \{T\}) \cup F', E', C') =$ *failure.*

Note that when a state is transformed by using a coinductive hypothesis, the set of accumulated constraints should be consistent for the coinductive hypothesis to be applied. The state in a transition system can also change by applying an instance of a clause from the program. When a subgoal is proved true, the unneeded nodes will be pruned from the forest. Note that the body of an inductive clause is overwritten on top of the leaf of a tree. When the tree contains only the root, the old tree will be replaced with one or more singleton trees. Coinductive subgoals will be remembered, in the form of a forest, so that infinite proofs can be recognized[2]. The state can also change by applying the inference rule and also by checking the consistency of constraints.

Definition 15. *A transition sequence in a co-CLP program P consists of a sequence of states* $(F_1, E_1, C_1), (F_2, E_2, C_2), \ldots$, *and a sequence of transition rules* $r_1, r_2, \ldots$, *such that* (F_i, E_i, C_i) *transitions to* $(F_{i+1}, E_{i+1}, C_{i+1})$ *by transition rule* r_i.

A transition sequence defines the trace of an execution. Execution terminates when it reaches a final state: either all atoms have been proved or the execution path reaches a state where no further transitions are possible.

Definition 16. *An accepting state is of the form* $(\emptyset, E, C)$, *where* $\emptyset$ *denotes the empty set. A failure state is a non-accepting state which does not have any outgoing transitions. A final state is an accepting state or a failure state.*

Definition 17. *A derivation of a state* (F, E, C) *in a co-constraint logic program P is a state transition sequence where the first state is* (F, E, C). *A derivation is successful if it ends in an accepting state, and a derivation is failed if it ends in a failure state. A goal G of the form* $a_1, \ldots, a_n$, *has a derivation in program P, if* $(\{node(a_i) | 1 \leq i \leq n\}, \emptyset, \emptyset)$ *has a derivation in P. For the goal G with free variables* $\tilde{x}$, *if* $(G, \emptyset, \emptyset)$ *has a successful derivation in P ending at* $(\emptyset, E, C)$, *then* $\exists_{-\tilde{x}} E(C)$ *is called the answer constraint of the derivation. Note that* $E(C)$ *is the expression obtained by applying the substitution E to C.*

Example 4. Assume predicate `p/2` of program P is defined as follows:

```
:-coinductive(p).
p([X | T], Y) :- {X - Y = 2}, p(T, Y).
```

The result of asking the query: `?-p(Z, 3).` is a substitution $E = \{Z =_u [X \mid T], Y =_u 3\}$ and a set of constraints $C = \{X - Y = 2\}$. Substituting 3 for Y in C results in $X = 5$. Since there is no inconsistency, the execution can proceed with X bounded to 5.

Example 5. Let `stream/2` and `number/1` be coinductive and inductive predicates, respectively. The execution of the following program can be used to generate/recognize infinite streams of natural numbers.

```
:-coinductive(stream).
:-inductive(number).
```

[2] Note that the ancestors of a subgoal can be accessed in the recursion stack.

```
stream([H | T], X) :- number(H), stream(T, Y), {Y - X >= 3}.
number(0).
number(s(N)) :- number(N).
```

The following is an execution trace for the query `?- stream([0, s(0), s(s(0)) | R], W).`, which shows the recursion stack and also the set of constraints after each recursive call (substitution is not shown).

1. `stream([0, s(0), s(s(0)) | R], W), C = ∅`
2. `stream([s(0), s(s(0)) | R], U), C = {U - W >= 3}`
3. `stream([s(s(0)) | R], V), C = {U - W >= 3, V - U >= 3}`

The next goal call is `stream(R, Z)`, which unifies with ancestor `(1)` and immediately succeeds, since the set of constraints C is consistent. Hence the original query succeeds with `R = [0, s(0), s(s(0)) | R]` with the answer constraint {`U -W` $\geq$ `3, V - U` $\geq$ `3`}.

3.4 Soundness and Completeness of co-CLP

The correctness results for co-CLP is an extension of the proofs for correctness of co-LP [13], which is presented next.

Lemma 1. *If (a, E_1, C_1) has a successful derivation in a co-constraint logic program P over the domain of constraints $\mathcal{D}$, with final state $(\emptyset, E_2, C_2)$, then (a, E_3, C_3) has a successful derivation in program P, where $\mathcal{D} \models C_3$, $E_2 \subseteq E_3, C_3 \models C_2$.*

Proof: follows by the fact that in the sequence of states in a derivation, the system of equations monotonically increases, also C_3 is consistent and $C_3 \models C_2$.□

Lemma 2. *If atom a has a successful derivation in a co-constraint logic program P over the domain of constraints $\mathcal{D}$, such that the* first *transition of this derivation uses the clause $a' \leftarrow c, b_1, \ldots, b_n$, the substitution obtained after this transition is E and the set of constraints is C, such that $E(a) = E(a')$ and $\mathcal{D} \models E(C)$, then each (b_i, E, C) also has a successful derivation in program P.*

Proof: Assume T is the successful derivation tree for (a, E, C). A derivation for (b_i, E, C) can be obtained by taking each transition that modifies the subtree rooted at b_i in T. However, for transitions obtained by applying a coinductive hypothesis rule of the form $\eta(\pi, \pi')$, this will not work, since the parent a of b_i no longer exists. We consider two cases: (i) $\pi = i.\pi_0$ and $\pi' = i.\pi_1$ for some integer i and paths π_0 and π_1, we apply the transition rule $\eta(\pi_0, \pi_1)$ to the corresponding leaf to which the original derivation have applied $\eta(\pi, \pi')$, (ii) $\pi = \epsilon$, a coinductive hypothesis cannot be applied to the corresponding leaf; however, the transitions of the entire original derivation of a can be recursively taken. □

Lemma 3. *If (a, E, C) has a successful derivation in a co-constraint logic program P over the constraint domain $\mathcal{D}$, which ends at state $(\emptyset, E', C')$, then $E'(E(a))$ is true in program P if $\mathcal{D} \models E'(E(C'))$.*

Proof: The model of P consists of the union of the models of the strata of P. We show that if (a, E, C) has a successful derivation in program P ending

at $(\emptyset, E', C')$, then all groundings of $E'(E(a))$ such that $\mathcal{D} \models E'(E(C'))$ are included in the model for the stratum in which a resides.

The proof proceeds by induction on the height of strata of P. Let Q_u be the set of all groundings over the U_P of $E'(E(a))$ (note that the Herbrand universe contains the domain of constraints $\mathcal{D}$) that satisfy $E'(E(C'))$ which are either in the same stratum u or some lower stratum. We prove by induction on the height of u that Q_u is contained in the model for u. We consider two cases:
(i) the stratum u is coinductive. We show that $Q_u \subseteq \nu T^{\mathcal{D}}_{u,P}$. Let $a' \in Q_u$, then there exist substitutions E_1, E_2, E_3, and set of consistent constraints C_2, C_3 such that $a' = E_1(E_2(E_3(a)))$, where E_1 is a grounding substitution for $E_2(E_3(a))$, (a, E_3, C_3) has a successful derivation ending in $(\emptyset, E_2, C_2)$, and $\mathcal{D} \models C_2$, $C_3 \models C_2$. Now assume $E = E_1 \cup E_2 \cup E_3$ and $C = C_2 \cup C_3$. By lemma 1, (a, E, C) has a successful derivation. This derivation must begin with application of a definite clause rule of the form $a'' \leftarrow c, b_1, \ldots, b_n$, and result in the state $(node(a, [b_1, \ldots, b_n]), E, C)$, where $E(a) = E(a''), \mathcal{D} \models E(C) = E(c)$. By lemma 2, each state (b_i, E, C) has a successful derivation. Let E' be a grounding substitution for the clause $E(a'' \leftarrow c, b_1, \ldots, b_n)$ and $C' = E(C)$, then $\mathcal{D} \models C'$. Let $E^* = E'(E(a'' \leftarrow c, b_1, \ldots, b_n))$, $E'(E(a'')) = a'$ and $E'' = E' \cup E$, then, $E^* = a' \leftarrow E''(C'), E''(b_1), \ldots, E''(b_n)$, $\mathcal{D} \models E''(C') = C''$. By lemma 1, each (b_i, E'', C'') has a successful derivation and and the stratification restriction implies that each $E''(b_i)$ is in a stratum equal to or lower than u. Hence $E''(b_i) \in Q_u$. Therefore, $Q_u \subseteq \nu T^{\mathcal{D}}_{u,P}$.
(ii) the stratum u is inductive. We show that $Q_u \subseteq \mu T^{\mathcal{D}}_{u,P}$. Let $a' \in Q_u$, then there exist substitutions E_1, E_2, E_3, and set of consistent constraints C_2, C_3 such that $a' = E_1(E_2(E_3(a)))$, where E_1 is a grounding substitution for $E_2(E_3(a))$, (a, E_3, C_3) has a successful derivation ending in $(\emptyset, E_2, C_2)$, and $\mathcal{D} \models C_2$, $C_3 \models C_2$. Now assume $E = E_1 \cup E_2 \cup E_3$ and $C = C_2 \cup C_3$. By lemma 1 (a, E, C) has a successful derivation. We consider two cases: (i) if a' is in a lower stratum than u, then it follows by induction, (ii) a' does not occur in a stratum lower than u, i.e., a' is an inductive atom. The proof proceeds by induction on the length of the derivation a'. If the derivation consists of just one transition, then a' must unify with a fact $a \leftarrow c$ in program P such that $\mathcal{D} \models E(C) = E(c)$, therefore $a' \in \mu T^{\mathcal{D}}_{u,P}$, if $\mathcal{D} \models c$. If the derivation is of length $k > 1$, then the derivation begins with an application of a program clause $a'' \leftarrow c, b_1, \ldots, b_n$, where $E(a) = E(a''), \mathcal{D} \models E(C) = E(c)$. By lemma 2 each state (b_i, E, C) has a successful derivation. Let E' be a grounding substitution for the clause $E(a'' \leftarrow c, b_1, \ldots, b_n)$ and $C' = E(C)$, then $\mathcal{D} \models C'$. Let $E^* = E'(E(a'' \leftarrow c, b_1, \ldots, b_n))$, $E'(E(a'')) = a'$ and $E'' = E' \cup E$, then, $E^* = a' \leftarrow E''(C'), E''(b_1), \ldots, E''(b_n)$, $\mathcal{D} \models E''(C') = C''$. By lemma 1, each (b_i, E'', C'') has a successful derivation of length $k' < k$, and the stratification restriction implies that each $E''(b_i)$ is in a stratum equal to or lower than u. We consider two cases: (i) $E''(b_i)$ is in a stratum strictly lower than u, then by induction on strata $E''(b_i) \in \mu T^{\mathcal{D}}_{u,P}$. (ii) $E''(b_i)$ is not in a stratum lower than u, then since it has a derivation of length $k' < k$, by induction on the length of the derivation, $E''(b_i) \in \mu T^{\mathcal{D}}_{u,P}$. Therefore, $a' \in \mu T^{\mathcal{D}}_{u,P}$. □

Theorem 1. *(soundness) If the query $a_1, \ldots, a_n$ has a successful derivation in co-CLP program P over the constraint domain $\mathcal{D}$, which ends at state $(\emptyset, E, C)$, then $E(a_1, \ldots, a_n)$ is true in program P if $\mathcal{D} \models E(C)$.*

Proof: If the goal $a_1, \ldots, a_n$ has a successful derivation in program P ending at state $(\emptyset, E, C)$, then each $E(a_i)$ independently has a successful derivation in program P if $\mathcal{D} \models E(C)$. By lemma 3, each $E(a_i)$ is true in program P. □

Theorem 2. *(completeness) Let $a_1, \ldots, a_n \in M^{\mathcal{D}}(P)$. If each $a_1, \ldots, a_n$ has a rational idealized proof, then the query $a_1, \ldots, a_n$ has a successful derivation in co-constraint logic program P over the domain of constraint $\mathcal{D}$.*

Proof: We only consider the case when $n = 1$, i.e., the query is a single atom, since the case for $n = 0$ is trivial and the case for $n > 1$ follows by composing the individual derivations of each atom of the original query. Let $a \in M^{\mathcal{D}}(P)$ have a rational idealized proof T. The derivation is constructed by a depth-first traversal of the idealized proof tree and recursively applying the clause corresponding to each node encountered to the corresponding leaf in the current state. The traversal stops at the coinductive root $\pi = \pi'.\pi''$ of a subtree that is identical to a subtree rooted at a proper coinductive ancestor π'. Then the derivation applies a transition rule of the form $\eta(\pi', \pi)$ to the leaf corresponding to R in the current state, and finally the depth-first traversal continues traversing starting at a node in the idealized proof tree corresponding to some leaf in the current state of the derivation.

The set of all subtrees of T is finite in cardinality. The stratification restriction prevents a depth-first traversal from encountering the same subtree twice along the same path, if the subtree has an inductive atom at its root. Only subtrees rooted at coinductive atoms can repeat in such a fashion. These imply that the maximum depth of the traversal in the idealized proof is finite. Moreover, all idealized proofs are finitely branching. This implies that the traversal always terminates. Therefore, the constructed derivation is finite.

We need to prove that the final state of the constructed derivation is a success state. We consider two cases that the traversal stops going deeper in the idealized proof tree: (i) the traversal reaches a leaf in the idealized proof tree, (ii) the traversal encounters a subtree identical to an ancestor subtree. In both cases, the derivation removes the corresponding leaf in the current state and maximum number of its ancestors such that the result is still a tree. Therefore, a leaf remains in the state only when its corresponding node in the proof tree has yet to be traversed. Since every node in the idealized proof tree corresponding to a leaf in the state is traversed at some point, the final state's tree contains no leaves, and hence the final state has an empty forest, which is the definition of an accepting state. Therefore, a has a successful derivation in program P. □

4 Conclusions

In this paper we proposed the paradigm of co-CLP which merges constraint logic programming and coinductive logic programming. The paradigm allows us to define inductive and coinductive predicates with constraints imposed on their arguments. We presented both the declarative and the operational semantics

of co-CLP. The declarative semantics allows the universe of terms to contain both finite and (rational) infinite terms. It also allows for the model to contain ground goals that have either finite or infinite idealized proofs. The operational semantics of co-CLP relies on the coinductive hypothesis rule, which is used to obtain finite derivations for goals which would otherwise have infinite derivations.

Co-CLP has interesting real world applications in which (rational) infinite computations are combined with constraints. An example application is a reactor temperature control system [6] in which the controllers run forever and the system requirements are specified as a set of constraints on physical quantities, namely, temperature and time. Many CPS also exhibit these two features. In summary, co-CLP is a new paradigm that can be used for modeling real-time, hybrid and cyber-physical systems [10, 12], that exhibit infinite behavior and also run under some constraints imposed by the environment or by other systems.

5 Acknowledgments

The authors would like to thank Feliks Kluźniak for constructive discussions and for proof-reading the paper.

References

1. J. Barwise and L. Moss. *Vicious circles: on the mathematics of non-wellfounded phenomena*. CSLI, Stanford, CA, USA, 1996.
2. A. Colmerauer. Prolog and infinite trees. In K. L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 231–251. Academic Press, London, 1982.
3. G. Gupta, N. Saeedloei, B. W. DeVries, R. Min, K. Marple, and F. Kluzniak. Infinite computation, co-induction and computational logic. In *CALCO*, pages 40–54, 2011.
4. R. Gupta. Programming models and methods for spatiotemporal actions and reasoning in cyber-physical systems. In *NSF Workshop on CPS*, 2006.
5. C. L. Heitmeyer and N. A. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *IEEE RTSS*, pages 120–131, 1994.
6. T. A. Henzinger and P. hsin Ho. Hytech: The cornell hybrid technology tool. In *Hybrid Systems II, LNCS 999*, pages 265–293. Springer-Verlag, 1995.
7. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.
8. E. A. Lee. Cyber-physical systems: Design challenges. In *ISORC*, May 2008.
9. J. W. Lloyd. *Foundations of logic programming / J.W. Lloyd*. Springer, Berlin, New York, 2nd, extended edition, 1987.
10. N. Saeedloei. *Modeling and Verification of Real-Time and Cyber-Physical Systems*. PhD thesis, University of Texas at Dallas, Richardson, Texas, 2011.
11. N. Saeedloei and G. Gupta. Verifying complex continuous real-time systems with coinductive clp(r). In *LATA*, pages 536–548, 2010.
12. N. Saeedloei and G. Gupta. A logic-based modeling and verification of CPS. *SIGBED Rev.*, 8:31–34, June 2011.
13. L. Simon. *Coinductive Logic Programming*. PhD thesis, University of Texas at Dallas, Richardson, Texas, 2006.
14. L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In *ICALP*, pages 472–483, 2007.
15. L. Sterling and E. Shapiro. *The art of Prolog (2nd ed.): advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1994.