Set-Based Failure Analysis for Logic Programs and Concurrent Constraint Programs

Andreas Podelski[1] Witold Charatonik[1⋆] Martin Müller[2]

[1] Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
{podelski;witold}@mpi-sb.mpg.de
[2] Programming Systems Lab, Universität des Saarlandes
66041 Saarbrücken, Germany
mmueller@ps.uni-sb.de

Abstract. This paper presents the first approximation method of the finite-failure set of a logic program by set-based analysis. In a dual view, the method yields a type analysis for programs with ongoing behaviors (perpetual processes). Our technical contributions are (1) the semantical characterization of finite failure of logic programs over infinite trees and (2) the design and soundness proof of the first set-based analysis of logic programs with the greatest-model semantics. Finally, we exhibit the connection between finite failure and the inevitability of the 'inconsistent-store' error in fair executions of concurrent constraint programs where no process suspends forever. This indicates a potential application to error diagnosis for concurrent constraint programs

Keywords: abstract interpretation, set-based program analysis, types, logic programs, concurrent constraint programs, finite failure, fairness

1 Introduction

Set-based program analysis dates back to Reynolds [35] and Jones and Muchnick [27] and forms a well-established research topic by now (see [1, 24, 34] for overviews and further references). It has direct practical applications to type inference, optimization and verification of imperative, functional, logic and, as we will see in this paper, also concurrent programs.

In set-based analysis, the problem of reasoning about runtime properties of programs is transferred to the problem of solving set constraints. The design of a specific analysis involves two steps: (1) define a mapping from a class of programs P to set constraints φ_P and show the soundness of the abstraction of P by a distinguished solution of φ_P, and (2) single out a corresponding subclass of set constraints and devise an efficient algorithm for computing the distinguished solution. For instance, Heintze and Jaffar defined a set-based analysis for logic

⋆ On leave from University of Wrocław, Poland. Partially supported by Polish KBN grant 8T11C02913.

programs with the least model semantics in [22]. Their analysis is an approximation method for the *success set* of a logic program, i.e. for the set of initial queries for which a successfully terminating execution exists.

In this paper, we consider the *finite failure set* of a logic program, i.e. the set of initial queries for which all fair executions terminate with failure. In order to give a *sound* prediction of finite failure ('if predicted, it will occur'), we need a characterization of finite failure in terms of program semantics. Classical results from logic programming, however, only yield the converse, i.e. a characterization of the greatest-model semantics in terms of finite failure (see Remark 1). Fortunately, for programs over the domain of infinite trees we can characterize finite failure through the greatest-model semantics (more precisely, its complement; see Theorem 1). Since the analysis we design computes an abstraction of that semantics, we obtain an approximation method for the finite-failure set of a logic program over infinite trees (see Theorem 3). More precisely, the emptiness of the computed abstract value for the predicate p indicates the finite failure of every predicate call $p(x)$. At the same time, this method can predict finite failure of a logic program over rational trees, or over finite trees (see Remarks 3 and 5).

In the least-model analysis in [22], Heintze and Jaffar use *definite* set constraints; they give a corresponding constraint solving algorithm in [21] (see [9] for further results). Our analysis uses *co-definite* set constraints, which bear their name in duality to definite set constraints due to the fact that every satisfiable constraint in this class has a greatest solution. This fact is crucial for our analysis. Algorithms for solving co-definite set constraints are given in [4, 16]. In this paper, we focus on the definition of the analysis and the soundness of the abstraction, which is: the greatest solution of the co-definite set constraint φ_P inferred from the program P is a safe approximation of the greatest-model semantics for P (see Theorem 2).

In a different reading, our abstraction method is a *type analysis* of logic programs with *ongoing behavior*. Such programs are investigated under the denomination *perpetual processes* in [28]. There, the semantics of such a program P is defined by the greatest-fixpoint semantics over the domain of infinite trees. Our analysis computes the abstraction of this semantics in the form of the greatest solution of the inferred co-definite set constraint (the greatest-fixpoint semantics is equal to the greatest model of P's completion [10]). This solution assings to every program variable x a set of infinite trees that can be viewed as the *type* of x. This type describes a safe approximation (i.e. a superset) of the set of all possible runtime values for x in ongoing program executions.

Finally, we consider a potential application to *concurrent constraint* programs (see e.g. [36, 37]). We carry over the approximation method of the greatest model to cc programs. This yields a type analysis for cc programs in the same sense as above. It also yields a failure analysis. In cc programs, an inconsistent constraint store (viz., failure) is considered a runtime error. (This is in contrast to logic programming where failure is part of the backtracking mechanism.) Our analysis computes an approximation of the execution states of cc programs for which failure is inevitable in fair executions unless a process (i.e. a predicate call) suspends forever (see Theorem 4). The global suspension of a process is not necessarily a programming error. That a process *must* suspend forever in order to avoid a runtime error is, however, a problem worth diagnosing and reporting.

Related Work. To our knowledge, set-based analysis for logic programming (see e.g. [5, 18, 13, 14, 22, 23, 30]) has previously only been designed to approximate the success set (which can be characterized by the least model semantics). Mishra's analysis [30] is often cited as the historically first one here. Heintze and Jaffar [23] have shown that Mishra's analysis is less accurate than theirs in two ways, due to the choice of the greatest solution for the class of set constraints he considers (see Remark 4) and due to the choice of the non-standard interpretation of non-empty *path-closed* sets of finite trees, respectively. Using the techniques in this paper, we are able to show that Mishra's approximation is so weak that it even approximates the greatest model. Mishra proves that '$p(x)$ will never succeed' if the set constraint ψ_P he derives is unsatisfiable. Our results yield that '$p(x)$ will finitely fail' if ψ_P is unsatisfiable over the domain of non-empty path-closed sets of *infinite* trees (see Remark 6).

Regarding the analysis of concurrent constraint programs, various techniques based on abstract interpretation have been used (see e.g. [17]) but none that is related to set-based analysis. A first formal calculus for (partial) correctness of cc programs is developed in [15]. The proof methods there are more powerful than ours but not automatic. The necessity to consider greatest-fixed point semantics for the analysis of reactive systems has been observed by other authors and in the context of different programming paradigms (see e.g. [11, 19]). None of these analyses is set-based.

Finally, we want to mention that the idea to derive necessary conditions for the inevitability of a runtime error by static analysis stems from the work of Bourdoncle [3] on *abstract debugging*.

2 Logic Programs

Preliminaries. We assume a ranked alphabet Σ fixing the arity $n \geq 0$ of its function symbols $f, g, \ldots$ and constant symbols $a, b, \ldots$, and an infinite set Var of variables $x, y, z, \ldots$. We write $\bar{x}$ for finite sequences of variables, and use analogous sequence notation for other syntactic entities. We also write $f(\bar{x})$ for flat terms, where we assume implicitly that the arity of f equals the length of $\bar{x}$. A term without variables is called a *ground term*. The set of *infinite trees* over Σ is denoted by T^{∞}_{Σ}. Note that an infinite tree can have finite paths (ending with a constant symbol); a finite tree is a special case. The set of terms over Σ and Var is denoted by $T^{\infty}_{\Sigma}(\mathsf{Var})$. For an arbitrary formula Φ, we write $\exists_{-x}\Phi$ for the existential closure of Φ with respect to all variables in Φ but x. We also assume a set Pred of predicate symbols. The *Herbrand Base* $\mathcal{B}$ is the set of all ground atoms over Pred and T^{∞}_{Σ}, i.e., $\mathcal{B} = \{p(t) \mid p \in \mathsf{Pred}, t \in T^{\infty}_{\Sigma}\}$.[1]

[1] What we call Herbrand Base is sometimes called *Complete Herbrand Base* [28] in order to distinguish it from the classical notion for finite trees.

Logic Programs. A *logic program* defines predicates through *clauses* of the form

$$p(t) \;\leftarrow\; p_1(t_1), \ldots, p_n(t_n)$$

where $p(t)$ is called the *head* and $p_1(t_1), \ldots, p_n(t_n)$ is called the *body* of the clause. A clause with an empty body is called a *fact*. A complete program has the form

$$\bigwedge_{p \in \mathsf{Pred}} \bigwedge_{i=1}^{n_p} p(t_i) \;\leftarrow\; \bigwedge_{j=1}^{n_{i,p}} p_{ij}(t_{ij}) \,.$$

where i ranges over the number n_p of clauses in the definition of predicate p, and j ranges over the number $n_{i,p}$ of queries in the i^{th} clause of predicate p. For better readability, we assume that all predicates are unary; the results can easily be extended to the case without this restriction (for example, by requiring the signature to contain at least one binary function symbol).

If we consider the logical semantics of a program of the form above, we take the *completion of* P [10], which is given by the following formula.

$$compl(P) \quad\equiv\quad \bigwedge_{p \in \mathsf{Pred}} \forall x \;\; p(x) \leftrightarrow \bigvee_{i=1}^{n_p} \exists_{-x}(\, x = t_i \wedge \bigwedge_{j=1}^{n_{i,p}} p_{ij}(t_{ij}) \,).$$

A *query* s is a conjunction $\bigwedge_k p_k(t_k)$ where the t_k are terms. We here allow infinite terms like $f(x, f(x, \ldots))$ in order to model execution states with cyclic unifiers such $y \mapsto f(x, y)$. Such terms can be finitely represented by equations, e.g. $y = f(x, y)$, or by syntact annotations as in [2].

A *ground query* is a query $\bigwedge_k p_k(t_k)$ such that all t_k are ground (i.e. without variables). We use the predicate constant *true* as the neutral element for conjunction: i.e., $s = s \wedge true$. In particular, the 'empty' query is written as *true*.

An *interpretation* ρ (sometimes called a *model*) is a subset of the Herbrand Base, $\rho \subseteq \mathcal{B}$. Interpretations are ordered by subset inclusion.

We identify an interpretation $\rho \subseteq \mathcal{B}$ with a valuation $\rho : \mathsf{Pred} \to 2^{T_\Sigma^\infty}$, i.e. a mapping of predicate symbols to sets of trees such that

$$\rho(p) = \{t \in T_\Sigma^\infty \mid p(t) \in \rho\}.$$

A *model of the program* P is a valuation $\rho : \mathsf{Pred} \to 2^{T_\Sigma^\infty}$ such that the formula $compl(P)$ is valid in the usual logical sense.

The greatest model of $compl(P)$, denoted by $gm(P)$, always exists. Using our convention of identifying the interpretation $gm(P)$ with a valuation, we use the notation $gm(P)(p)$ for the denotation of the predicate p by the greatest-model semantics, i.e.

$$gm(P)(p) = \{t \in T_\Sigma^\infty \mid p(t) \in gm(P)\}.$$

Operational Semantics. The logic program P defines a *fair transition system* $\mathcal{T}_P = \langle \mathcal{S}, \tau_P \rangle$. The fairness of the transition system is defined by the fairness of the non-deterministic *selection rule* (in the classical sense [28]: a selection rule is fair if every query atom in a state s gets selected eventually, in every execution starting in s). The non-determinism of the selection rule means that conjunction corresponds to parallel composition with the interleaving semantics; disjunction corresponds to non-deterministic choice.

The set $\mathcal{S}$ of states of the transition system $\mathcal{T}_P$ consists of all queries (including *true*) and the *failure* state *false*,

$$\mathcal{S} \quad = \quad \{\bigwedge_k p_k(t_k) \mid \forall k \;\; p_k \in \mathsf{Pred},\; t_k \in T^{\infty}_{\Sigma}(\mathsf{Var})\} \cup \{\mathit{false}\}$$

The transition relation $\tau_P \subseteq \mathcal{S} \times \mathcal{S}$ is defined according to the standard rewriting semantics under a fair selection rule. When a *selected* query atom $p(t)$ in a state $s \in \mathcal{S}$ of the form $s = s_{rest} \wedge p(t)$ unifies with the head of a clause $p(t_i) \leftarrow \bigwedge_i p_{ij}(t_{ij})$, then the state s' obtained as the instantiation of $s_{rest} \wedge \bigwedge_i p_{ij}(t_{ij})$ under the most general unifier of t and t_i is a possible successor state of s. We say that $p(t)$ is *applied* in the transition step from s to s'. When a selected query atom $p(t)$ does not unify with any of the heads of the clauses of p, then the successor state is *false*.

Similarly, P defines a fair *ground* transition system $\mathcal{T}^g_P = \langle \mathcal{S}, \tau^g_P \rangle$. We obtain the transition relation τ^g_P by modifying the one of $\mathcal{T}_P$: after every transition step of $\mathcal{T}^g_P$, all variables in the successor state are instantiated with ground terms (i.e. infinite trees). Note that ground queries are a special case of queries.

We say that a derivation *finitely fails* if if it ends in the state *false*. A query $p(x)$ is *finitely failed* (and belongs to the set FF) if every $\mathcal{T}_P$ derivation starting with query $p(x)$ finitely fails.

$$FF \quad = \quad \{p(x) \;\mid\; p(x) \text{ is finitely failed}\}$$

Similarly, a ground query $p(t)$ is called *ground finitely failed* (and belongs to the set GFF) if every T^g_P derivation starting from $p(t)$ finitely fails.

$$GFF \quad = \quad \{p(t) \;\mid\; p(t) \text{ is ground finitely failed}\}$$

We will now characterize the finite failure set of a program P over the domain T^{∞}_{Σ} of infinite trees through the greatest model of $compl(P)$. Since we have not found this observation in the literature, we will give its proof, drawing from several results that are classical in the theory of logic programming.

Theorem 1 (Characterization of finite failure over infinite trees). Given a logic program P over infinite trees, the query $p(x)$ is finitely failed if and only if the value of p in the greatest model of $compl(P)$ over the domain T^{∞}_{Σ} of infinite trees is the empty set; i.e.,

$$p(x) \in FF(P) \;\text{ if and only if }\; gm(P)(p) = \emptyset.$$

Proof. The only-if direction is a classical result (namely, the 'algebraic soundness of finite failure', see [28, 25]).

For the other direction, first note that equations over infinite trees have the *saturation property*, that is, an infinite set of constraints is satisfiable if every of its finite subsets is [28, 26, 33].

Now assume that $p(x) \notin FF(P)$. Since (see [28, 26])

$$gm(P)(p) = \{t \mid p(t) \notin GFF(P)\},$$

it is sufficient to show that there exists an infinite tree t such that $p(t) \notin GFF(P)$ (i.e., $p(t)$ is not in the ground finite failure set; note that in general, the ground finite failure of a call does not imply finite failure of some ground instance of this call.)

By assumption, there exists an execution starting in the state $p(x)$ that does not lead to the failure state. That is, there exists a transition sequence $s_0, s_1, s_2, \ldots$ starting in $s_0 = p(x)$ such that the constraint store φ_i of every state s_i is satisfiable (in the terminology of constraint logic programming [25], a state $\bigwedge_k p_k(t_k)$ is written as the pair $\langle \bigwedge_k p_k(x_k), \varphi \rangle$ where the constraint store φ is a conjunction of equations that is equivalent to $\bigwedge_k x_k = t_k$ over the domain of infinite trees). Since φ_i is stronger than φ_{i-1} for $i \geq 1$, φ_n is equivalent to $\bigwedge_{i=0}^n \varphi_i$.

Thus, we have a sequence of constraints $\varphi_0, \varphi_1, \varphi_2, \ldots$ such that $\bigwedge_{i=0}^n \varphi_i$ is satisfiable for all n. The saturation property yields that also the infinite conjunction $\bigwedge_{i\geq 0} \varphi_i$ is satisfiable. Let α be a solution of $\bigwedge_{i\geq 0} \varphi_i$. Then the transition sequence $s'_o, s'_1, s'_2, \ldots$ that we obtain by instantiating the states s_i by the valuation α is a ground transition sequence that does not lead to the fail state. Hence, if $\alpha(x) = t$, then $p(t) \notin GFF(P)$ and $GFF(P)(p)$ is nonempty. $\square$

Remark 1. Palmgren [33] has shown that a constraint logic program over a constraint domain with the saturation property is canonical. That is, $gfp(T_P) = T_P{\downarrow}^\omega$ (where $_P{\downarrow}^\omega = \bigcap_{i=1}^\omega T_P^i(\mathcal{B})$ holds; for the definition of T_P see Section 4.) Since $gfp(T_P) = \mathcal{B} \backslash GFF(P)$ holds for canonical programs (see [25]), this is sufficient to characterize *ground* finite failure over infinite trees. Canonicity is not sufficient for finite failure of non-ground queries.

For example, consider the program $p(f(x)) \leftarrow p(x)$ over the structure of *finite* trees. This program is canonical (over finite trees). Its greatest model over finite trees assigns p the empty set (in accordance with the fact that $p(t) \in GFF(P)$ for all *finite* trees t), but $p(x)$ is not finitely failed.

Similarly, Jaffar and Stuckey [26] have shown that for programs over infinite trees, $T_P \downarrow \omega$ equals the complement of $[FF(P)]$, where $[FF(P)]$ is the set of *ground instances* of elements of $FF(P)$. This is a characterization of the denotational semantics through the operational semantics; our characterization is the converse.

Remark 2. The statement of Theorem 1 holds for constraint logic programs over every constraint system with the saturation property ('an infinite set of constraints is satisfiable if every of its finite subsets is').

Remark 3. Since the structure of *rational* trees and the structure of infinite trees are elementarily equivalent [29] (in particular, the test of satisfiability of constraints is the same), we can take the operational semantics of programs over rational trees in Theorem 1 (but we must consider the logical semantics over infinite trees; note that ratinal tree constraints do not have the saturation property). The modified statement is:
Given a logic program P over rational *trees, the query $p(x)$ is finitely failed if and only if the value of p in the greatest model of $compl(P)$ over the domain T_Σ^∞ of* infinite *trees is the empty set.*

3 Co-definite Set Constraints

Syntax. A (general) *set expression* e is built from first-order terms, union, intersection, complement, and the projection operator [21]:

$$e \quad ::= \quad x \mid f(\overline{e}) \mid e \cup e' \mid e \cap e' \mid e^c \mid f_{(k)}^{-1}(e)$$

The projection $f_{(k)}^{-1}(e)$ is only defined if k is a positive integer smaller than the arity of f. If e does not contain the complement operator, then e is called a *positive* set expression. A (general) *set constraint* is a conjunction of inclusions of the form $e \subseteq e'$.

A *definite* set constraint [21] is a conjunction of inclusions $e_l \subseteq e_r$ between positive set expressions, where the set expressions e_r on the right hand side of $\subseteq$ are furthermore restricted to contain only variables, constants and function symbols and the intersection operator (i.e., no projection or union).

Definition 1. A *co-definite* set constraint φ is a conjunction of inclusions $e_l \subseteq e_r$ between positive set expressions, where the set expressions e_l on the left-hand side of $\subseteq$ are further restricted to contain only variables, constants, unary function symbols and the union operator (that is, no projection, intersection or terms with a function symbol of arity greater than one).

$$e_l \quad ::= \quad x \mid a \mid f(e) \qquad e_r \quad ::= \quad x \mid f(\overline{e}) \mid e \cup e' \mid e \cap e' \mid f_{(k)}^{-1}(e)$$

Semantics. We interpret set constraints over $2^{T_\Sigma^\infty}$, the domain of sets of trees over the signature Σ. That is, variables denote sets of trees, and a (set) valuation is a mapping $\alpha : \mathsf{Var} \to 2^{T_\Sigma^\infty}$. Tree constructors are interpreted as functions over sets of trees: the constant a is interpreted as $\{a\}$, and the function symbol f is interpreted as the function which maps sets $S_1, \ldots, S_n$ into the set

$$\{f(t_1, \ldots, t_n) \mid t_1 \in S_1, \ldots, t_n \in S_n\}\,.$$

The application of the projection operator for a function symbol f and the k-th argument position on a set S of trees is defined by

$$f_{(k)}^{-1}(S) = \{t \mid \exists t_1, \ldots t_n : \ \ t_k = t,\, f(t_1, \ldots, t_k, \ldots, t_n) \in S\}\,.$$

The set operators union $\cup$ and intersection $\cap$, as well as inclusion $\subseteq$ are interpreted as usual. Define the union of set valuations $\bigcup_i \alpha_i$ on variables as the pointwise union on the images of all variables; i.e., $(\bigcup_i \alpha_i)(x) = \bigcup_i \alpha_i(x)$.

The following properties hold for co-definite set constraints (see also [4]). These properties are essential for our proof in the following section to work, which shows soundness of abstraction.

Proposition 1 (Properties of co-definite set constraints).

1. Solutions of co-definite set constraints are closed under arbitrary unions. That is, the valuation $\bigcup_i \alpha_i$ is a solution if the valuations α_i, $i \in I$, are.
2. If satisfiable, every co-definite set constraint φ has a greatest solution, noted $gSol(\varphi)$.
3. Every co-definite set constraint without inclusions of the form $a \subseteq x$ is satisfiable.

Proof. The first claim is proved by case-distinction over the possible set inclusions. The second is an immediate corollary from the first one. (Note that the restriction to constants and monadic function symbols on the left hand side of an inclusion is crucial here. For instance, the set constraint $f(x,y) \subseteq f(a,a) \cup f(b,b)$ does not have a greatest solution; it has two maximal but incomparable ones.) In order to verify the third claim notice that the valuation which maps every variable into the empty set is a solution of co-definite set constraints without inclusions of the form $a \subseteq e$. $\Box$

Remark 4. Mishra [30] uses a class of set constraints with a non-standard interpretation over non-empty *path-closed* sets of finite trees to approximate the success set of a logic program. (A set of trees is path-closed if it can be recognized by a deterministic top-down tree automaton [20].) Set constraints over non-empty path-closed sets also have the properties 1. and 2. above. Due to the non-standard interpretation, this holds even if n-ary constructor terms are allowed on the left side of the inclusion. For example, the constraint $f(x,y) \subseteq f(a,a) \cup f(b,b)$ has a greatest solution over path-closed sets (which assigns both variables x and y the set $\{a,b\}$).

4 Set-based Analysis

We will next describe the inference of a co-definite set constraint φ_P from a logic program P. The intuition is as follows. A clause of the form $p(t_i) \leftarrow p_j(t_{ij})$ can be written equivalently as $p(x_i) \leftarrow x_i = t_i \wedge t_{ij} = x_{ij} \wedge p(x_{ij})$. Following the abstract interpretation framework, we abstract the semantics-defining fixpoint operator T_P by replacing the constraint $x_i = t_i \wedge t_{ij} = x_{ij}$ in its definition by the co-definite set constraint $x_i \subseteq t_i \wedge \Phi(t_{ij} \subseteq x_{ij})$; the operator Φ is defined below. The fixpoint equation for the abstract operator $T_P^{\#}$ is essentially the inferred set constraint φ_P. The soundness of the abstraction follows directly. The schema of our method (whose ingredients are Propositions 1 and Lemma 1 below) is described in an abstract setting in [12].

We next introduce the operator Φ that assigns an inclusion of the form $t \subseteq x$ a co-definite set constraint. For example, $\Phi(f(x,y) \subseteq f(a,a) \cup f(b,b))$ is essentially the conjunction of $x \subseteq f^{-1}_{(1)}(f(a,a) \cup f(b,b))$ and $y \subseteq f^{-1}_{(2)}(f(a,a) \cup f(b,b))$ which is equivalent to the conjunction of $x \subseteq a \cup b$ and $y \subseteq a \cup b$.

We introduce a fresh variable z_t for each subterm t appearing in the formula and then define the constraint $\Phi(t \subseteq x)$ for a term t and a variable x by induction on the depth of t.

$$\Phi(y \subseteq x) = y \subseteq x$$

$$\Phi(t \subseteq x) = \begin{pmatrix} z_t \subseteq x \wedge z_{t_1} \subseteq f^{-1}_{(1)}(z_t) \wedge \Phi(t_1 \subseteq z_{t_1}) \\ \dots \\ \wedge\, z_{t_n} \subseteq f^{-1}_{(n)}(z_t) \wedge \Phi(t_n \subseteq z_{t_n}) \end{pmatrix} \quad \text{for } t = f(t_1, \dots, t_n)$$

Lemma 1. If a tree valuation $\alpha : \mathsf{Var} \to T^{\infty}_{\Sigma}$ satisfies the equality $x = t$, then the set valuation $\sigma_\alpha : \mathsf{Var} \to 2^{T^{\infty}_{\Sigma}}$ defined by $\sigma_\alpha(x) = \{\alpha(x)\}$ satisfies the co-definite set constraints $x \subseteq t$ and $\Phi(t \subseteq x)$. □

We define the co-definite constraint φ_P inferred from P as follows. Here, we assume that the different clauses are renamed apart (if not, we apply α-renaming to quantified variables).

$$\varphi_P \quad \equiv \quad \bigwedge_{p \in \mathsf{Pred}} \; p \subseteq \bigcup_i^{n_p} t_i \;\wedge\; \bigwedge_i^{n_p} \bigwedge_j^{n_{i,p}} \Phi(t_{ij} \subseteq p_{ij})$$

Both, symbols $p \in \mathsf{Pred}$ and $x \in \mathsf{Var}$ act here as second-order variables ranging over sets of trees. In the following, when we compare an interpretation ρ of a logic program with a valuation σ of a set constraint, $\rho \subseteq \sigma$ means that $\rho(p) \subseteq \sigma(p)$ for all $p \in \mathsf{Pred}$.

Theorem 2 (Soundness of Abstraction).
For a logic program P, the greatest model of P's completion is smaller than the greatest solution of φ_P, formally $gm(P) \subseteq gSol(\varphi_P)$.

Proof. We first define an abstraction $T^{\#}_P$ of the T_P operator, and we prove that $gfp(T_P) \subseteq gfp(T^{\#}_P)$, using Lemma 1. In the second part we show that $gfp(T^{\#}_P) \subseteq gSol(\varphi_P)$, using here Proposition 1.

1. $\boldsymbol{gfp(T_P) \subseteq gfp(T^{\#}_P)}$. The T_P operator maps an interpretation ρ to another one $T_P(\rho)$ where, for all $p \in \mathsf{Pred}$,

$$T_P(\rho)(p) \quad = \quad \left\{ t \in T_\Sigma \;\middle|\; \begin{array}{r} \exists \alpha : \mathsf{Var} \to T_\Sigma \;\; \exists i : \;\; t = \alpha(t_i), \\ T^{\infty}_{\Sigma}, \alpha \models \bigwedge_j t_{ij} \in \rho(p_{ij}) \end{array} \right\} .$$

As usual, we write $\mathcal{M}, \alpha \models F$ if the formula F is valid under the interpretation with the valuation α on the structure (with the domain) $\mathcal{M}$. The greatest-model semantics and the greatest-fixpoint semantics of a program P coincide; i.e., the greatest model of P's completion is the greatest fixpoint of the operator T_P, formally $gm(P) = gfp(T_P)$ (see e.g. [28]).

The $T_P^\#$ operator maps an interpretation ρ to the interpretation $T_P^\#(\rho)$ where, for all $p \in \mathsf{Pred}$,

$$T_P^\#(\rho)(p) \;=\; \left\{ t \in T_\Sigma \;\middle|\; \begin{array}{l} \exists \sigma : \mathsf{Var} \to 2^{T_\Sigma^\infty},\;\; \exists i : \;\; t \in \sigma(t_i), \\ \qquad 2^{T_\Sigma^\infty}, \sigma \models \; \bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \wedge x_{ij} \subseteq \rho(p_{ij}) \end{array} \right\}.$$

Here, we use new variables x_{ij} as placeholders for p_{ij}. The variables $x \in \mathsf{Var}$ now range over sets of trees. The formula above is a co-definite set constraint with additional constants noted $\rho(p_{ij})$. The constant $\rho(p_{ij})$ is interpreted as the set $\rho(p_{ij})$.

Let $\rho' = T_P(\rho)$ and $\rho'' = T_P^\#(\rho)$. Then $\rho'(p) \subseteq \rho''(p)$ holds for all $p \in \mathsf{Pred}$. This can be seen as follows. For every tree valuation α satisfying the condition in the set comprehension for ρ', the set valuation σ_α defined by $\sigma_\alpha(x) = \{\alpha(x)\}$ satisfies the condition in the set comprehension for ρ''. Clearly, $\sigma_\alpha(t_{ij}) \subseteq \rho(p_{ij})$; we replace the inclusion $t_{ij} \subseteq \rho(p_{ij})$ by the equivalent conjunction $t_{ij} = x_{ij} \wedge x_{ij} \subseteq \rho(p_{ij})$. If σ_α satisfies the equality $t_{ij} = x_{ij}$ then also $\Phi(t_{ij} \subseteq x_{ij})$ by Lemma 1.

Hence, $T_P^\#$ is indeed an abstraction of T_p, and, thus, $gfp(T_P) \subseteq gfp(T_P^\#)$. This concludes the first part of the proof.

2. $\boldsymbol{gfp(T_P^\#) \subseteq gSol(\varphi_P)}$. In order to show that $gfp(T_P^\#) \subseteq gSol(\varphi_P)$, we first reformulate the definition of $T_P^\#$ as follows.

$$T_P^\#(\rho)(p) \;\;=\;\; \bigcup_{\sigma:\mathsf{Var}\to 2^{T_\Sigma^\infty}} \;\; \bigcup_i \{\sigma(t_i) \mid 2^{T_\Sigma^\infty}, \sigma \models \bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \;\wedge\; x_{ij} \subseteq \rho(p_{ij})\}$$

Fix ρ and let $\rho'' = T_P^\#(\rho)$.

We next exploit the fact that the solutions of co-definite set constraints are closed under arbitrary unions (Proposition 1). Hence, we can replace the union of solutions in the formula above by the greatest solution. We obtain that

$$\rho''(p) \;=\; \bigcup_i \sigma_i(t_i) \quad \text{where} \quad \sigma_i \;=\; gSol(\bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \wedge x_{ij} \subseteq \rho(p_{ij})).$$

Since all program variables are renamed apart, we have $\rho''(p) = \bigcup_i \sigma(t_i)$ where

$$\sigma \;\;=\;\; gSol(\bigwedge_i \bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \wedge x_{ij} \subseteq \rho(p_{ij})).$$

Thus, we have $\rho''(p) = \sigma(p)$ where

$$\sigma \;\;=\;\; gSol(p = \bigcup_i t_i \wedge \bigwedge_i \bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \wedge x_{ij} \subseteq \rho(p_{ij})).$$

Again, since all program variables are renamed apart,

$$\rho'' \quad = \quad gSol(\bigwedge_{p \in \mathsf{Pred}} p = \bigcup_i t_i \wedge \bigwedge_i \bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \wedge x_{ij} \subseteq \rho(p_{ij})).$$

Here, we equate the interpretation $\rho'' : \mathsf{Pred} \to 2^{T_\Sigma^\infty}$ with a valuation σ interpreting a formula with predicate symbols $p \in \mathsf{Pred}$ and tree variables $x \in \mathsf{Var}$ both ranging over sets of trees, and with constants of the form $\rho(p_{ij})$ standing for the corresponding sets. We omit any further formalization of this setting.

Let ρ_0 be any fixpoint of $T_P^\#$, i.e., $T_P^\#(\rho_0) = \rho_0$. This means that ρ_0 is a solution (the greatest one, in fact) of

$$\bigwedge_{p \in \mathsf{Pred}} p = \bigcup_i t_i \wedge \bigwedge_i \bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \wedge x_{ij} \subseteq \rho_0(p_{ij}).$$

That is, ρ_0 is a solution of φ_P. Hence, ρ_0 is smaller than the greatest solution of φ_P. This is true in particular if ρ_0 is chosen as the *greatest* fixpoint of $T_P^\#$. This concludes the second part of the proof. □

Theorem 3 (Set-based failure analysis for logic programs). The query $p(x)$ is finitely failed in every fair execution of the logic program P if the value of p in the greatest solution (over sets of infinite trees) of the co-definite set constraint φ_P derived from P is the empty set; i.e., for all predicates $p \in \mathsf{Pred}$, if $gSol(\varphi_P)(p) = \emptyset$ then $p(x) \in FF(P)$.

Proof. We combine Theorems 2 and 1. □

A more precise formulation of the statement above is: the emptiness of the computed value for an argument variable in the i-th clause of p entails the finite failure of every predicate call of p with that clause.

Remark 5. Since the domains of infinite and rational trees are equivalent wrt. to finite failure, and failure over infinite trees implies failure over finite trees, we have the following two statements.
Given a logic program P over rational *trees [over* finite *trees], the query $p(x)$ is finitely failed if the value of p in the greatest solution over sets of* infinite *trees of the co-definite set constraint φ_P derived from P is the empty set.*

Remark 6. Essentially, the set constraint derived from a logic program P in the 'least-model' analysis of Mishra [30] is of the form

$$\psi_P \quad \equiv \quad \bigwedge_{p \in \mathsf{Pred}} p = \bigcup_i^{n_p} t_i \ \wedge \ \bigwedge_i^{n_p} \bigwedge_j^{n_{i,p}} t_{ij} \subseteq p_{ij}.$$

Instead of Lemma 1, we have the obvious fact that the set valuation σ_α defined by $\sigma_\alpha(x) = \{\alpha(x)\}$ satisfies the set constraint $x = t$ (which is equivalent to $t = x$) if the tree valuation α satisfies the tree constraint $x = t$. Since we also have the existence of greatest solutions over the domain of non-empty path-closed sets of (finite or infinite) trees (see Remark 4), the proof of Theorem 2 goes through also for ψ_P instead of φ_P, and the statements in this and the next section hold in the appropriate adaptation. One can prove that $gSol(\varphi_P) \leq gSol(\psi_P)$ (see [5]), i.e. the analysis using path-closed constraints is less accurate than the one with co-definite set constraints. Solving path-closed constraints is still an open problem (both, for least and for greatest solutions).

5 Concurrent Constraint Programs

We consider *concurrent constraint (*cc*) programs* (see e.g. [36, 37]) in a normalized form such that we can employ a Prolog-style clausal syntax. This is a notational convention which is convenient to establish a connection to logic programming.

Furthermore, we consider only the case where constraints C are term equations $t_1 = t_2$ interpreted over infinite trees, as in the cc programming language and system Oz [32, 37]. Hence, we can adopt a Prolog-like syntax and assume that every procedure p is defined either by a single fact or by several *guarded clauses* of the form

$$p(x) \;\leftarrow\; x = t \;[\!]\; p_1(t_1), \ldots, p_n(t_n).$$

In such a guarded clause, we call $x = t$ the *guard* and $p_1(t_1), \ldots, p_n(t_n)$ the *body*.

The operational semantics of a cc program P is defined through a fair transition system $\mathcal{T}_P^{\mathsf{cc}}$ as $\mathcal{T}_P$ for logic programs (again with the non-deterministic fair selection rule), with one important difference: A selected query $p(t)$ can only be applied if amongst the guarded clauses of predicate p there is one, the i^{th} one with body $x = t_i \;[\!]\; \bigwedge_j p_{ij}(t_{ij})$, say, such that $x = t$ entails $\exists_{-x}\, x = t_i$; if this is the case in a state S, then the successor state will be $S \wedge \bigwedge_j p_{ij}(t_{ij})$ under the most general unifier of t and t_i (for a more precise definition, see e.g. [36, 37]). Notice that a logic program is a special case of a cc program where all guards are trivially true, e.g. $x = x$.

Failure of cc *programs.* We next apply the approximation method of the previous section to logic programs abstracting cc programs in order to predict the behavior of the latter.

Define the logic program $\tilde{P}$ *abstracting* the cc program P by replacing the guard $[\!]$ with conjunction. It is an abstraction in the following sense.

Proposition 2. If the query $p(t)$ finitely fails in the logic program $\tilde{P}$ abstracting the cc program P then failure is inevitable in fair executions of the cc program P unless a process (i.e. a predicate call) suspends forever.

Proof. Observe that every (finite or infinite) fair computation in P in which no process suspends forever induces a fair computation in $\tilde{P}$. Namely, whenever a selected query $p(t)$ is applied with a guarded clause in P it can also be applied with the associated unguarded clause in $\tilde{P}$. This proves the claim by contraposition. □

Proposition 3 (Prediction of failure behavior of cc programs). Failure is inevitable in fair executions of the cc program P unless a process suspends forever, if the value of p in the greatest model of $compl(\tilde{P})$ over the domain T_Σ^∞ of infinite trees is the empty set.

Proof. We combine Proposition 2 and Theorem 1. □

Theorem 4 (Set-based failure analysis for cc programs). Failure is inevitable in fair executions of the cc program P unless a process suspends forever if the value of p in the greatest solution (over sets of infinite trees) of the co-definite set constraint $\varphi_{\tilde{P}}$ derived from $\tilde{P}$ is the empty set.

Proof. We combine Theorem 3 and Proposition 3. □

6 Examples

We will give some examples to illustrate how our method of approximating greatest models with co-definite set constraints tests the inevitability of certain runtime errors. Consider the following simple stream program.

$$\begin{array}{l}\mathsf{stream}([X,Y|S]) \leftarrow Y = s(s(X)),\ \mathsf{computation}(X),\ \mathsf{stream}([Y|S]).\\ \mathsf{main}(Z) \leftarrow \mathsf{stream}([Z|T]).\end{array}$$

Suppose we know that the predicate computation makes sense only for (trees representing) odd numbers, whereas no such restriction is known for main and stream. This invariant can be expressed by the following set constraint, which may have been derived from another code fragment or externally provided by a program annotation.

$$\mathsf{computation} \subseteq s(0) \cup s(s(\mathsf{computation}))\ . \qquad (1)$$

Further, we can approximate the set of non-failed computations of the program with the constraint

$$\begin{array}{l}\mathsf{stream} \subseteq cons(X, cons(Y,S)) \wedge \\ X \subseteq \mathsf{computation} \wedge X \subseteq s_{(1)}^{-1}(s_{(1)}^{-1}(Y)) \wedge \\ Y \subseteq cons_{(1)}^{-1}(\mathsf{stream}) \wedge S \subseteq cons_{(2)}^{-1}(\mathsf{stream}) \wedge \\ \mathsf{main} \subseteq cons_{(1)}^{-1}(\mathsf{stream})\ .\end{array} \qquad (2)$$

It is not difficult to see that the greatest solution of the conjunction of (1) and (2) assigns to the variable main (as well as to X, Y, and computation) the set of odd numbers. We obtain from this fact that, for example, the query main(0) inevitably leads to a state where computation is called with a wrong argument.

We illustrate now the necessity to consider infinite trees by another example. Consider the reactive logic program P defined by

$$p(f(x)) \leftarrow p(x).$$

The execution of the query $p(x)$ does not fail, whether the program is defined over the domain of finite or infinite trees. We derive the co-definite set constraint $\varphi_P \equiv p \subseteq f(x) \wedge x \subseteq p$. When interpreted over sets of finite trees, φ_P has as greatest solution the valuation assigning the empty set to p (and x). In the infinite tree case the greatest solution assigns to p the singleton set containing the infinite tree $f(f(f(\ldots)))$. That is, an interpretation of the derived co-definite set constraint over sets of finite trees does not admit the prediction of finite failure.

7 Conclusion

We have presented a set-based analysis of logic programs with ongoing behavior (i.e. with the greatest-fixpoint semantics). We have given a characterization of finite failure of logic programs over rational or infinite trees through the greatest model over infinite trees, and we have exhibited a connection between the inevitability of 'inconsistent-store' runtime error for cc programs and finite failure for logic programs, thus indicating a potential application to error diagnosis for cc programs.

Our 'greatest-model' set-based analysis of logic programs is interesting in its own right, as a particular instance of static analysis, and also in comparision with the 'least-model' set-based analyses of classical logic programs e.g. by Mishra [30] or by Heintze and Jaffar [22].

The practicability of our approach depends on the efficiency of the constraint solving. Succeeding the technical report [8] on which this paper and [4] are based, Devienne, Talbot and Tison [16] have given a strategy for solving co-definite set constraints which may achieve an exponential speedup. The realization of this set-based analysis for the Oz system, and its extension to reactive Oz programs with non-cc features such as cells and higher-order features is part of ongoing work. We have implemented a prototype version (with an incomplete constraint solver); experiments seem to indicate its potential usefulness for finding bugs.

One question arising from this work and the work by Cousot and Cousot in [12] is whether this set-based analysis is an instance of an abstract interpretation, i.e., whether our constraint-solving process is isomorphic to the iteration of an abstraction of the T_P fixpoint operator.

References

1. A. Aiken. Set constraints: Results, applications and future directions. In *Proceedings of the Workshop on Principles and Practice of Constraint Programming*, LNCS 874, pages 326–335. Springer-Verlag, 1994.

2. H. Aït-Kaci and A. Podelski. Functions as passive constraints in LIFE. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16 (3):1–40, 1994.
3. F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI'93)*, pages 46–55. ACM Press, 1993.
4. W. Charatonik and A. Podelski. Co-definite set constraints. In T. Nipkow, editor, *9th International Conference on Rewriting Techniques and Applications*, pages 211–225, Springer-Verlag. 1998.
5. W. Charatonik and A. Podelski. Directional Type Inference for Logic Programs. In G. Levi, editor, *Proceedings of the International Symposium on Static Analysis.* LNCS, pages 278–294, Springer-Verlag. 1998.
6. W. Charatonik, D.McAllester, D. Niwiński, A. Podelski and I. Walukiewicz. The Horn Mu-calculus. In V. Pratt, editor, *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, pages 58–69, IEEE Press. 1998.
7. W. Charatonik and A. Podelski. Set-based Analysis of Reactive Infinite-state Systems. In B. Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. LNCS 1384, pages 358–375, Springer-Verlag. 1998.
8. W. Charatonik and A. Podelski. Set constraints for greatest models. Technical Report MPI-I-97-2-004, Max-Planck-Institut für Informatik. 1997.
9. W. Charatonik and A. Podelski. Set constraints with intersection. In G. Winskel, editor, *Twelfth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 362–372, IEEE Press. 1997.
10. K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.
11. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Proc. POPL '92*, pages 83–94. ACM Press, 1992.
12. P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Record of FPCA '95 - Conference on Functional Programming and Computer Architecture*, pages 170–181, La Jolla, California, USA, 25-28 June 1995. SIGPLAN/SIGARCH/WG2.8, ACM Press, New York, USA.
13. J. P. Gallagher, D. Boulanger, and H. Saglam. Practical model-based static analysis for definite logic programs. In *Proceedings of the 1995 International Symposium on Logic Programming*, pages 351–368.
14. J. P. Gallagher and L. Lafave. Regular approximation of computation paths in logic and functional languages. In *Proceedings of the Dagstuhl Workshop on Partial Evaluation*, pages 1–16. Springer-Verlag, February 1996.
15. F. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages (POPL)*, pages 98–108. ACM Press, 1994.
16. P. Devienne, J.-M. Talbot, and S. Tison. Solving classes of set constraints with tree automata. In G. Smolka, editor, *Proceedings of the Third International Conference on Principles and Practive of Constraint Programming*, pages 62–76, volume 1330 of *Lecture Notes in Computer Science.* Springer-Verlag, 1997.
17. M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Compositional Analysis for Concurrent Constraint Programming. In *Proceedings of the Eight Annual IEEE Symposium on Logic in Computer Science*, pages 210–221, 1993.

18. T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 300–309, July 1991.
19. K. L. S. Gasser, F. Nielson, and H. R. Nielson. Systematic realisation of control flow analyses for CML. In *Proceedings of ICFP'97*, pages 38–51. ACM Press, 1997.
20. F. Gécseg and M. Steinby. *Tree Automata*. Akademiai Kiado, 1984.
21. N. Heintze and J. Jaffar. A decision procedure for a class of set constraints (extended abstract). In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 42–51, 1990.
22. N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 197–209, January 1990.
23. N. Heintze and J. Jaffar. Semantic types for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 141–156. MIT Press, 1992.
24. N. Heintze and J. Jaffar. Set constraints and set-based analysis. In *Proceedings of the Workshop on Principles and Practice of Constraint Programming*, LNCS 874, pages 281–298. Springer-Verlag, 1994.
25. J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *The Journal of Logic Programming*, 19/20:503–582, 1994.
26. J. Jaffar and P. J. Stuckey. Semantics of infinite tree logic programming. *Theoretical Computer Science*, 46:141–158, 1986.
27. N. D. Jones and S. S. Muchnick. Flow analysis and optimization of lisp-like structures. In *Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 244–256, January 1979.
28. J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation. Springer-Verlag, Berlin, Germany, second, extended edition, 1987.
29. M.J. Maher. Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees. In *Proceedings of the Symposium of Logic in Computer Science*, pages 348–457, IEEE Press, 1988.
30. P. Mishra. Towards a theory of types in Prolog. In *IEEE International Symposium on Logic Programming*, pages 289–298, 1984.
31. M. Müller. *Type Analysis for a Higher-Order Concurrent Constraint Language*. PhD thesis, Universität des Saarlandes, Technische Fakultät, D-66041 Saarbrücken, Germany, 1998.
32. The Oz System. Programming Systems Lab, Universität des Saarlandes. Available through the WWW at `http://www.ps.uni-sb.de/oz`. 1997.
33. E. Palmgren. Denotational semantics of Constraint Logic Programming – a non-standard approach. In B. Mayoh, E. Tyugu, and J. Penjam, editors, *Constraint Programming*, volume 131 of *NATO ASI Series F: Computer and System Sciences*, pages 261–288. Springer-Verlag, Berlin, Germany, 1994.
34. L. Pacholski and A. Podelski. Set Constraints: A Pearl in Research on Constraints (Tutorial Abstract). In G. Smolka, editor, *Proceedings of the Third International Conference on Principles and Practive of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
35. J. Reynolds. Automatic Computation of Data Set Definitions. *Information Processing*, 68, 1969.
36. V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *18^{th} Symposium on Principles of Programming Languages*, pages 333–352. ACM Press, Jan. 1991.
37. G. Smolka. The Oz Programming Model. In *Volume 1000 of Lecture Notes in Computer Science*. Springer-Verlag, 1995.