Set-based Error Diagnosis of Concurrent Constraint Programs

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

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

Abstract. This paper presents a method for the automated detection of errors in concurrent constraint programs, the first one to our knowledge. We use a new set-based analysis for this purpose. The method is based on our observation that one can understand the inevitability of deadlock or failure in concurrent constraint programs through finite failure in logic programming. The technical contributions are a characterization of finite failure of logic programs over infinite trees, and the proof of the soundness of the abstraction in our set-based analysis.

Keywords: program analysis, set constraints, logic programming, reactive systems, concurrent constraint programming

1 Introduction

The concurrent constraint (cc) paradigm is a powerful concurrent programming paradigm in which the – superficially unrelated – concepts of concurrency and constraints fruitfully interact (for entry points to the vast literature, see [30, 31, 12, 10]). On the one hand, concurrency can be a useful programming abstraction for the hard task of writing constraint solvers. On the other hand, the intrinsicly complex synchronization of concurrent processes can be transferred to high-level, conceptually simpler logical constraint problems. The cc paradigm makes it possible to turn these two reciprocal dependencies into a useful programming tool. By now, several implementations exist; we are interested in particular in contributing to the development of the freely available Oz System [25, 31].

It is evident that any kind of static analysis is desirable that can detect errors or increase confidence in the correctness of programs. This holds in general, and even more so in the traditionally untyped setting of concurrent constraint programming. More generally, we believe that concurrent constraint languages have a high potential for automated verification because of the close connection with logic: computation states are described by constraints and concurrent composition corresponds to conjunction.

In this paper, we first present a set-based analysis for concurrent constraint programs and for logic programs that specify reactive infinite-state systems;

that is, programs which are intended to run forever. The states of such a system are, in logic programming terminology, possibly non-ground queries (*i.e.*, calls with partially instantiated arguments). In the terminology of constraint logic programming, a state consists of concurrent procedure calls and the constraint store. We call such (cc or logic) programs *reactive*.[3]

We then apply our set-based analysis to the automated and static prediction of an important kind of errors in cc programs, namely deadlock or failure. The application of set-based analysis for this purpose is original, and we obtain the first automated method for the detection of programming errors in cc programs.

In set-based analysis (see, *e.g.*, [1, 18, 28] and references therein), the problem of reasoning about runtime properties of programs is delegated to the problem of solving set constraints. The design of a system addressing a particular program analysis problem (for a particular class of programs) involves two steps: (1) define a mapping from 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 solving them. For instance, Heintze and Jaffar defined a set-based analysis for logic programs with the least model semantics in [16]. In this analysis, they used so-called *definite* set constraints (which are called definite because every satisfiable constraint in this class has a *least* solution), and they showed that their constraint solving problem is decidable [15].

Our analysis uses *co-definite* set constraints as defined in [6]. Co-definite set constraints bear their name in duality to definite set constraints due to the fact that every satisfiable constraint in this class has a greatest solution.[4] This property is crucial for our analysis of reactive systems. In this paper we focus on the definition of the analysis and the soundness proof of the abstraction. An algorithm for solving co-definite set constraints is given in [6].

We give our analysis in two steps. First, we consider logic programs, and then we turn to concurrent constraint programs. The denotational semantics of a reactive *logic* program P is defined by the greatest fixpoint of the immediate consequence operator T_P associated with P, which also is the greatest model $gm(P)$ of P's completion [22, 8]. We associate a co-definite set constraint φ_P with every reactive logic program P such that the greatest solution $gSol(\varphi_P)$ of φ_P approximates the greatest model of P's completion. In formulas:

$$gm(P) \quad \subseteq \quad gSol(\varphi_P)$$

We can relate this approximation of the denotational semantics to the standard operational semantics of logic programs [22, 19], because we can give a characterization of finite failure over infinite trees through the greatest model (see

[3] Reactive logic programs are called "perpetual processes" in [22]; the term "reactive" coined in [27] has become standard for referring to non-terminating behavior.

[4] Apart from the name duality, definite and co-definite set constraints are not dual, neither with respect to their syntax nor with respect to the constraint solving problem (although they have the same complexity characterizations). See [6] for details.

Theorem 1). This allows us to infer finite failure of the reactive logic program from emptiness of some variable in $gSol(\varphi_P)$.

Now we consider *concurrent constraint* programs. We can carry over the approximation result of the greatest model to cc programs. The interpretation of $gSol(\varphi_P)$ in operational terms, however, needs some more care due to the possibility of deadlock. For cc programs without deadlock the characterization of finite failure through the greatest model still holds. Generally, we cannot statically exclude deadlock. Hence we show that emptiness in $gSol(\varphi_P)$ implies failure in every fair execution of a cc program P unless a deadlock occurs (Theorem 8).

In cc programs, an inconsistent constraint store (*viz.*, failure) is considered a run-time error. This is in contrast to logic programming where failure is part of the backtracking mechanism. A cc program has certainly an error if every possible fair execution leads to failure. In our programming experience with Oz, a cc program with the possibility to deadlock should also be considered erroneous. In other words, emptiness under the greatest solution of φ_P correctly approximates the inevitability of a run-time error (namely, deadlock or failure) in a program P.

Finally, our set-based analysis of reactive logic programs is interesting in its own right, as a particular instance of static analysis, type inference or approximation of runtime values (as we show in the section on related work), and also in comparision with the existing fundamental studies of set-based analysis of sequential (terminating) logic programs which are based on the least model semantics (see, *e.g.*, [23, 16, 17, 18]).

Overview. We consider some examples in the following section. Section 3 mentions related work. Section 4 defines logic programs and concurrent constraint programs. In Section 7 we define and investigate co-definite set constraints. Section 8 defines our analysis and proves it correct. Section 9 concludes.

2 Examples

We illustrate by an example 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.

$$\mathsf{stream}([X, Y|S]) :- \; Y = s(s(X)),\ \mathsf{computation}(Y),\ \mathsf{stream}([Y|S]).$$
$$\mathsf{main}(Z) :- \; \mathsf{stream}([Z|T]).$$

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.

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

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

$$\begin{aligned}
&\mathsf{stream} \subseteq cons(X, cons(Y, S)) \wedge \\
&X \subseteq \mathsf{computation} \ \wedge \ X \subseteq s^{-1}_{(1)}(s^{-1}_{(1)}(Y)) \wedge \\
&Y \subseteq cons^{-1}_{(1)}(\mathsf{stream}) \ \wedge \ S \subseteq cons^{-1}_{(2)}(\mathsf{stream}) \wedge \\
&\mathsf{main} \subseteq cons^{-1}_{(1)}(\mathsf{stream}) \, .
\end{aligned} \tag{2}$$

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 $p(f(x)) :- \ p(x)$. The execution of the non-ground query $p(x)$ does not fail, independent of whether the program is defined over the domain of finite or infinite trees. From P, 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 a conclusion about finite failure of non-ground queries.

3 Related Work

The history of set-based program analysis dates back to Reynolds [29] and Jones and Muchnick [21], and forms a well-established research topic by now (see, *e.g.*, [2, 3, 5, 16, 17, 23], and [1, 18, 28] for overviews and further references). Its research combines theoretical investigations ranging from expressiveness and decidability to program semantics and domain theory. It has direct practical applications to type inference, optimization and verification of imperative, functional, logic and – as we will see in this paper – also reactive programs.

Up to now, set-based analysis for logic programming has considered the least model semantics. Mishra's analysis [23] is often cited as the historically first one here. A comparison with our analysis sheds a new light on this.

Mishra has used a class of set constraints with a non-standard interpretation over non-empty *path-closed* sets of finite trees to approximate the least model of a logic program. Set constraints over path-closed sets also have the property that every satisfiable constraint has a greatest solution.

Due to the non-standard interpretation, this property holds even if n-ary constructor terms are allowed left of the inclusion. For example, the constraint $f(x, y) \subseteq f(a, a) \cup f(b, b)$ has a greatest model over path-closed sets (which assigns both variables x and y the set $\{a, b\}$) while it has two maximal but incomparable ones over the standard set domain. Therefore, constructor terms with an arity greater than 2 have been excluded from co-definite set constraints [6].

Heintze and Jaffar [17] have shown that Mishra's analysis is less accurate than theirs in two ways, due to the choice of the greatest solution and due to the choice of the non-standard interpretation, respectively. The proof of the soundness of our abstraction carries readily over if we take Mishra's set constraints instead of co-definite ones. This means that Mishra's approximation is so weak that it even approximates the greatest model. Mishra was interested only in terminating executions and hence considered logic programs with respect to the least model semantics. He proves that "$p(x)$ will never succeed" if p is analysed to be necessarily empty. For the case of *finite* trees, this is indeed all one can show. Our remarks above imply that one can strengthen this consequence by inferring that "$p(x)$ will always fail", provided that one takes non-empty path-closed sets of *infinite* instead of finite trees.

For the analysis of concurrent constraint programs, various techniques based on abstract interpretation habe been used (see, *e.g.*, [12]) but none related to set-based analysis. A first formal calculus for (partial) correctness of cc programs is developed in [10]. 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.*, [9, 14]). Again, none of them is a set-based analysis.

4 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^∞. The set of terms over Σ and Var is denoted by $T_\Sigma(\mathsf{Var})$. Both trees and terms are denoted by t. For an arbitrary formula Φ, we write as $\exists_{-x}\Phi$ the existential closure of Φ with respect to all variables in Φ but x. We also assume a set Pred of predicate symbols p. The *Herbrand Base* $\mathcal{B}$ is the set of all ground atoms over Pred and T_Σ^∞, *i.e.*, $\mathcal{B} = \{p(t) \mid p \in \mathsf{Pred}, t \in T_\Sigma^\infty\}$.[5]

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

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

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

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

[5] What we call Herbrand Base is sometimes called *Complete Herbrand Base* [22] when it needs to be distinguished from the set of ground atoms over Pred and *finite* trees.

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).

As the denotational semantics for a program of the form above we assume the *completion of* P [8] 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. A *ground query* is a query $\bigwedge_k p_k(t_k)$ such that all t_k are ground. We use the predicate constant *true* as the neutral element for conjunction: *i.e.*, $s = s \wedge \mathit{true}$. 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 ρ with a valuation $\rho : \mathsf{Pred} \to 2^{T_\Sigma^\infty}$ from predicate symbols to sets of trees (where $\rho(p) = \{t \in T_\Sigma^\infty \mid p(t) \in \rho\}$). A *model of the program* P is a valuation such that the formula $compl(P)$ is valid (in the usual sense of logic). The greatest model of $compl(P)$, denoted by $gm(P)$, always exists.

Operational Semantics. The logic program P defines a *fair transition system* $\mathcal{T}_P$ whose one-step transition relation is defined as usual for derivations of logic programs or constraint logic programs [22, 19], with the fair non-deterministic selection rule.

The states of $\mathcal{T}_P$ are possibly non-ground queries (including *true*), and the *failure* state *false*:[6]

$$\{\bigwedge_k p_k(t_k) \mid \forall k \;\; p_k \in \mathsf{Pred}, \; t_k \in T_\Sigma(\mathsf{Var})\} \cup \{\mathit{false}\}$$

When a selected query $p(t)$ in a state S unifies with the head of a clause $p(t_i) \text{:-} \bigwedge_i p_{ij}(t_{ij})$, then a successor state is $S \wedge \bigwedge_i p_{ij}(t_{ij})$ under the most general unifier of t and t_i. We say that $p(t)$ is *applied*. When a selected query $p(t)$ does not unify with any of the heads of the clauses of p, then the successor state is *false*. The non-determinism of the selection rule means that conjunction has an interleaving operational semantics (*i.e.*, conjunction corresponds to parallel composition); disjunction corresponds to non-deterministic choice. The fairness of the selection rule is exactly the fairness of the transition system.

[6] There is a slightly subtle point here. The notion of queries considered here is not general enough to model arbitrary execution states of a reactive logic program over infinite trees. Since *terms* are finite, we can use them only to model *non-cyclic* constraint stores. This can be repaired by considering, as in [19], queries of the form $\bigwedge_k p(t_k) \leftarrow C$ where C is a conjunction of term equations. We gloss over this detail here since it is not essential to make our point (see however [24]).

Similarly, P defines a *fair ground transition system* $\mathcal{T}_P^g$ which modifies $\mathcal{T}_P$ such that after every step of $\mathcal{T}_P$ all variables in the obtained state are substituted with ground terms. The states of this transition system are (possibly empty) ground queries and *false*.

$$\{\bigwedge_k p_k(t_k) \mid \forall k \;\; p_k(t_k) \in \mathcal{B}\} \cup \{false\}$$

We say that a derivation *succeeds* [*fails*] if if it ends in the state *false*. A query $p(x)$ *succeeds* if there exists a derivation leading to the state *true*; $p(x)$ is *finitely failed* if every $\mathcal{T}_P$ derivation starting with query $p(x)$ leads to failure after finitely many steps.

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

Similarly, a ground query $p(t)$ is called *ground finitely failed* if every T_P^g derivation starting from $p(t)$ reaches failure after finitely many steps.

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

We can 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 short proof, which draws from several well-known results.

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.*, $p(x) \in FF(P)$ iff $gm(P)(p) = \emptyset$.

Proof. The only-if direction is a standard result (the algebraic soundness of finite failure [22, 19]).

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 [22, 20, 26].

Now assume that $p(x) \notin FF(P)$. Since $gm(P)(p) = \{t \mid p(t) \notin GFF(P)\}$, 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 ground finite failure; note that in general, the ground finite failure of a call does not imply finite failure of some ground instance of this call.)

There exists an execution starting in the state $s = p(x)$ that does not lead to the failure state. That is, there exists a transition sequence $s = s_0, s_1, s_2, \ldots$ such that (in the terminology of constraint logic programming [19]) the constraint store φ_i of every state s_i is satisfiable. Since φ_i is stronger than φ_{i-1} for $i \geq 1$, this implies that $\bigwedge_{i=0}^n \varphi_i$ is satisfiable for all n. The saturation property yields that also $\bigwedge_{i\geq 0} \varphi_i$ 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 that does not lead to the fail state. Hence, if $\alpha(x) = t$, then $p(t) \notin GFF(P)$. □

Remark. Palmgren [26] has shown that a constraint logic program over a constraint domain with the saturation property is canonical. That is, $gfp(T_P) = T_P{\downarrow}^{\omega} = \bigcap_{i=1}^{\infty} T_P^i(\mathcal{B})$ holds. (For the definition of T_P see Section 8.) Since $gfp(T_P) = \mathcal{B}\backslash GFF(P)$ holds for canonical programs, this is only sufficient to characterize *ground* finite failure over infinite trees. For illustration, consider the program $p(f(x)) :- p(x)$ which is canonical even over finite trees. Its greatest model over finite trees assigns p the empty set and $p(t) \in GFF(P)$ for all *finite* tree t; but $p(x) \notin FF(P)$.

A similar remark applies to the Jaffar and Stuckey's result [20] that $T_P \downarrow \omega$ equals $\mathcal{B}\backslash[FF(P)]$ where $[FF(P)]$ is the set of *ground instances* of elements of $[FF(P)]$. Based on this result, one can characterize the finite failure of *ground* queries only.

5 Concurrent Constraint Programs

We consider *concurrent constraint (*cc*) programs* (see, *e.g.*, [30, 31]) in a normalized form such that we can employ a Prolog-style clausal syntax. This is convenient to establish a direct 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 [25, 31]. Hence, we can replace every constraint conjunct C (*i.e.*, each *tell* operation in cc terminology) by calls of new predicates each of which is defined by a fact. More precisely, we transform every program as follows. First, we add to the program a new fact of the form $p_t(t)$ for every occurring term t. Second, we replace every query $p(t)$ and, third, every equation $t_1 = t_2$ in bodies as follows.

$$\begin{array}{lcll} p(t) & \rightsquigarrow & p(x),\ p_t(x) & x \text{ fresh} \\ t_1 = t_2 & \rightsquigarrow & p_{t_1}(t_2),\ p_{t_2}(t_1) & \end{array}$$

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

$$p(x) :- \ x = t \parallel p_1(t_1), \ldots, p_n(t_n).$$

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 $x = t_i \parallel \bigwedge_j p_{ij}(t_{ij})$, 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.*, [30, 31]). Notice that a logic program is a special case of a cc program where all guards are trivially true, *e.g.*, $x = x$.

6 Failure or Deadlock

We say that an execution sequence *deadlocks* if it contains a query $p(t)$ that is never applied because none of the corresponding guards is ever entailed. We observe the analogy with failure: an execution sequence *(finitely) fails* if it contains a query $p(t)$ that is never applied because it does not unify with any of the heads of the clauses of p. Failure is considered a run-time error in cc systems. In our programming experience, programs which deadlock should also be considered erroneous. (Of course, our method cannot directly be applied to a program where deadlock is a desired feature of its execution behavior, and not a bug.)

We want to give a conservative approximation of all execution states of cc programs for which either failure or deadlock is inevitable. Define the logic program $\tilde{P}$ *abstracting* the cc program P by replacing the guard $\parallel$ 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 it either deadlocks or fails in P. If the query $p(t)$ succeeds in P then it also succeeds in $\tilde{P}$.

Proof. Observe that every (finite or infinite) fair computation in P which neither fails nor deadlocks induces a computation in $\tilde{P}$ which does not fail or deadlock, either. This can be made formal by a simulation argument which exploits that 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 second claim immediately, and also the first one by contraposition. □

Corollary 3 Characterization of finite failure or deadlock. Given a concurrent constraint program P over infinite trees, the query $p(x)$ is either finitely failed or it deadlocks 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.

Proof. We combine Proposition 2 and Theorem 1. □

7 Co-definite Set Constraints

A (general) *set expression* e is built from first-order terms, union, intersection, complement, and the projection operator [15]:

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

The projection $f^{-1}_{(k)}(e)$ 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 [15] 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 4. 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).

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

We interpret set constraints over $2^{T^\infty_\Sigma}$, the domain of sets of trees over the signature Σ. That is, variables denote sets of trees, and a (set) valuation is a mapping $\alpha : \mathsf{Var} \to 2^{T^\infty_\Sigma}$. 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

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

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 [6]). These properties are essential for our proof in the following section to work, which shows soundness of abstraction.

Proposition 5. 1. The solutions of co-definite set constraints are closed under arbitrary union.
2. Every co-definite set constraint has a greatest solution if satisfiable.
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. 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 $a \subseteq e$. □

We write the greatest solution of a satisfiable co-definite set constraint φ as $gSol(\varphi)$.

8 Set-based Analysis

We will next describe the inference of a co-definite set constraint φ_P from a logic program P. We assume that the different clauses are renamed apart (if not, we apply α-renaming to quantified variables). 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.

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

We define the constraint φ_P inferred from P as follows.

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

Here, we treat both symbols $p \in \mathsf{Pred}$ and $x \in \mathsf{Var}$ as second-order variables which range over sets of trees. In the following, when we compare an interpretation ρ 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 6 Soundness of Abstraction. For any logic program P, the greatest model of P's completion is smaller than the greatest solution of φ_P: $gm(P) \subseteq gSol(\varphi_P)$.

Proof. We first define an abstraction $T_P^\#$ of the T_P operator, and we prove that $gfp(T_P) \subseteq gfp(T_P^\#)$. Here, we extend valuations σ over trees to valuations α_σ over sets of trees by $\alpha(x) = \{\sigma(x)\}$. In the second part we show that $gfp(T_P^\#) \subseteq gSol(\varphi_P)$ where we use Proposition 5.

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

$$\mathcal{T}_P(\rho)(p) \quad = \quad \left\{ t \in T_\Sigma \;\middle|\; \begin{array}{r} \exists \alpha : \mathsf{Var} \to T_\Sigma \; \exists i : \; t = \alpha(t_i), \\ T_\Sigma^\infty, \alpha \models \; t_{ij} \in \rho(p_{ij}) \end{array} \right\}. \tag{3}$$

The greatest-model semantics and the greatest-fixpoint semantics of a program P coincide. That is, the greatest model of P's completion is the greatest fixpoint of the operator T_P, $gm(P) = gfp(T_P)$.

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

$$T_P^\#(\rho)(p) \;=\; \left\{ t \in T_\Sigma \;\middle|\; \begin{array}{r} \exists \sigma : \mathsf{Var} \to 2^{T_\Sigma^\infty}, \; \exists i : t \in \sigma(t_i), \\ 2^{T_\Sigma^\infty}, \sigma \models \; \bigwedge_j \Phi(t_{ij} \subseteq x_{ij}) \wedge x_{ij} \subseteq \rho(p_{ij}) \end{array} \right\}. \tag{4}$$

Here, we use new variables x_{ij} as placeholders for p_{ij} (for better *legibility*: otherwise, p_{ij} would appear as a variable on which both, σ and ρ are applied). The variables $x \in \mathsf{Var}$ now range over sets of trees. 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 formula F above consists of co-definite set constraints in conjunction with inclusions between variables and constants $\rho(p_{ij})$ (interpreted as the corresponding set).

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 $\sigma(x) = \{\alpha(x)\}$ 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 $t_{ij} = x_{ij} \wedge x_{ij} \subseteq \rho(p_{ij})$, and if σ_α satisfies the equality $t_{ij} = x_{ij}$ then also the weaker constraint $\Phi(t_{ij} \subseteq x_{ij})$.

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. $gfp(\boldsymbol{T_P^{\#}}) \subseteq gSol(\boldsymbol{\varphi_P})$**.** 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 $\rho'' = T_P^{\#}(\rho)$.

Next, we exploit the fact that the solutions of co-definite set constraints are closed under (arbitrary) union (Proposition 5). This fact extends to formulas containing inclusions with set constants on the right hand side, such as $x_{ij} \subseteq \rho(p_{ij})$. Note that for all i, the formula in question is satisfiable. 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,

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

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 7 Set-based analysis of finite failure over infinite trees.
The query $p(x)$ is finitely failed in every fair execution of the reactive 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 6 and 1. □

Theorem 8 Set-based analysis of erroneous cc programs.
The procedure p in the cc program P finitely fails or deadlocks 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 7 and Corollary 3. □

9 Conclusion

We have presented a set-based analysis of reactive logic programs (over infinite trees with the greatest-fixpoint semantics). We have given a characterization of finite failure of logic programs over infinite trees through the greatest model, and we seen a connection between an error in cc programs and finite failure.

We have applied our analysis to obtain the first automated method for the static prediction of deadlock or failure in concurrent constraint programs used for the specification of reactive infinite-state systems. To our knowledge, this is the first set-based analysis of reactive programs (with greatest-fixpoint semantics).

We now need to evaluate the practicability of our approach, and the efficiency of constraint solving on real programs. In succession to the technical report [7] on which part of this paper and [6] is based, Devienne, Talbot and Tison [11] have given a strategy for solving co-definite set constraints which may achieve an exponential speedup. However, as the authors point out, their setup uses

bottom-up tree automata and thus applies to the case of finite trees only, while the case of infinite trees is the relevant one for analyzing cc programs.

The realization of this framework for the Oz system, and its extension to reactive Oz programs with non-cc features such as cells and higher-order features is a challenging research question and part of ongoing work. We have implemented a prototype version (with an incomplete constraint solver); we have used it in experiments which demonstrate its usefulness for finding bugs.

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. A. Aiken and B. Murphy. Static type inference in a dynamically typed language. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 279–290, January 1991.
3. A. Aiken, E. L. Wimmers, and T. Lakshman. Soft typing with conditional types. In *Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, Jan. 1994.
4. A. Arnold and M. Nivat. Formal computations of non deterministic recursive program schemes. *Mathematical Systems Theory*, 13:219–236, 1980.
5. S. K. Biswas. A demand-driven set-based analysis. In *The 24th ACM Symposium on Principles of Programming Languages POPL '97*, pages 372–385, Paris, France, 15–17 Jan. 1997.
6. W. Charatonik and A. Podelski. Co-definite set constraints. Submitted for publication. www.mpi-sb.mpg.de/~podelski/papers/union.html.
7. W. Charatonik and A. Podelski. Co-definite set constraints. Technical Report MPI-I-97-2-004, Max-Planck-Institut für Informatik, 1997.
8. K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.
9. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Proc. POPL '92*, pages 83–94. ACM Press, 1992.
10. 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.
11. 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*, volume 1330 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
12. 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.
13. 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.
14. 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.

15. 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.
16. 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.
17. N. Heintze and J. Jaffar. Semantic types for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 141–156. MIT Press, 1992.
18. 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.
19. J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *The Journal of Logic Programming*, 19/20:503–582, May-July 1994.
20. J. Jaffar and P. J. Stuckey. Semantics of infinite tree logic programming. *Theoretical Computer Science*, 46:141–158, 1986.
21. 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.
22. J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation. Springer-Verlag, Berlin, Germany, second, extended edition, 1987.
23. P. Mishra. Towards a theory of types in Prolog. In *IEEE International Symposium on Logic Programming*, pages 289–298, 1984.
24. M. Müller. *Type Analysis for a Higher-Order Concurrent Constraint Language*. PhD thesis, Universität des Saarlandes, Technische Fakultät, D-66041 Saarbrücken, Germany, expected 1997.
25. The Oz System. Programming Systems Lab, Universität des Saarlandes. Available through the WWW at `http://www.ps.uni-sb.de/oz`. 1997.
26. 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.
27. A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In J. de Bakker, W.-P. de Roever, and G.p Rozenberg, editors, *Current Trends in Concurrency*, Lecture Notes in Computer Science 224, pages 510–584. Springer-Verlag, 1986.
28. 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.
29. J. Reynolds. Automatic Computation of Data Set Definitions. *Information Processing*, 68, 1969.
30. 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.
31. G. Smolka. The Oz Programming Model. In *Volume 1000 of Lecture Notes in Computer Science*. Springer-Verlag, 1995.

This article was processed using the LaTeX macro package with LLNCS style