Set Constraints: Results, Applications and Future Directions

Alexander Aiken

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720-1776
aiken@cs.berkeley.edu

Abstract. Set constraints are a natural formalism for many problems that arise in program analysis. This paper provides a brief introduction to set constraints: what set constraints are, why they are interesting, the current state of the art, open problems, applications and implementations.

1 Introduction

Set constraints are a natural formalism for describing relationships between sets of terms of a free algebra. A set constraint has the form $X \subseteq Y$, where X and Y are *set expressions*. Examples of set expressions are 0 (the empty set), α (a set-valued variable), $c(X, Y)$ (a constructor application), and the union, intersection, or complement of set expressions.

Recently, there has been a great deal of interest in program analysis algorithms based on solving systems of set constraints, including analyses for functional languages [AWL94, Hei94, AW93, AM91, JM79, MR85, Rey69], logic programming languages [AL94, HJ92, HJ90b, Mis84], and imperative languages [HJ91]. In these algorithms, sets of terms describe the possible values computed by a program. Set constraints are generated from the program text; solving the constraints yields some useful information about the program (e.g., for type-checking or optimization).

Set constraints have proven to be a very successful formalism. On the theoretical side, rapid progress has been made in understanding the algorithms for and complexity of solving various classes of set constraints. On the practical side, several program analysis systems based either entirely or partially on set constraint algorithms have been implemented. In addition, the use of set constraints has simplified previously known, but rather complicated, program analyses and set constraints have led directly to the discovery of other, previously unknown, analyses.

Much of the work on set constraints is very recent. Consequently, many of the results are not well known outside of the community of researchers active in the area. The purpose of this paper is to provide a brief, accessible survey of the area: what set constraints are, why they are useful, what is and isn't known about solving set constraints, the important open problems, and likely directions for future work. Section 2 gives definitions of the basic set constraint formalism and some illustrative examples. Section 3 presents a survey of results on the satisfiability, complexity, and solvability of various set constraint problems; open problems are also discussed. In Section 4 a brief, informal description of algorithms for solving systems of set constraints is given; this discussion also points out basic trade-offs between expressive power and computational complexity for various classes of set constraint problems. Section 5 surveys applications of set constraints to program analysis. Section 6 concludes with a discussion of current implementations and likely directions for future work.

2 Set Constraints

Let C be a set of constructors and let V be a set of variables. Each $c \in C$ has a fixed arity $a(c)$; if $a(c) = 0$ then c is a constant. The *set expressions* are defined by the following grammar:

$$E ::= \alpha \mid 0 \mid c(E_1, \ldots, E_{a(c)}) \mid E_1 \cup E_2 \mid E_1 \cap E_2 \mid \neg E_1$$

In this grammar, α is a variable (i.e., $\alpha \in V$) and c is a constructor (i.e., $c \in C$). Set expressions denote sets of *terms*. A term is $c(t_1, \ldots, t_{a(c)})$ where $c \in C$ and every t_i is a term (the base cases of this definition are the constants). The set H of all terms is the Herbrand universe. An *assignment* is a mapping $V \rightarrow 2^H$ that assigns sets of terms to variables. The meaning of set expressions is given by extending assignments from variables to set expressions as follows:

$$\begin{aligned}
\sigma(0) &= \emptyset \\
\sigma(c(E_1, \ldots, E_n)) &= \{c(t_1, \ldots, t_n) | t_i \in \sigma(E_i)\} \\
\sigma(E_1 \cup E_2) &= \sigma(E_1) \cup \sigma(E_2) \\
\sigma(E_1 \cap E_2) &= \sigma(E_1) \cap \sigma(E_2) \\
\sigma(\neg E_1) &= H - \sigma(E_1)
\end{aligned}$$

A *system of set constraints* is a finite conjunction of constraints $\bigwedge_i X_i \subseteq Y_i$ where each of the X_i and Y_i is a set expression. A *solution* of a system of set constraints is an assignment σ such that $\bigwedge_i \sigma(X_i) \subseteq \sigma(Y_i)$ is true. A system of set constraints is *satisfiable* if it has at least one solution. The following result was proven first in [AW92]. Simpler proofs have been discovered since [BGW93, AKVW93].

Theorem 1. It is decidable whether a system of set constraints is satisfiable. Furthermore, all solutions can be finitely presented.

It is important to note that the definition of set constraints used here does damage to history. The original formulation of set constraints, due to Heintze and Jaffar [HJ90a], also includes projection operations in the constraint language. However, it is convenient pedagogically to present results as extensions of the definition above. This organization also reflects the manner in which recent research has progressed.

From the definition above, it is easy to see that the set expressions consist only of elementary set operations plus constructors—simply put, it is a set theory of terms. The constraint language is rich enough, however, to describe all of the data types commonly used in programming, and it is this property that makes set constraints a natural tool for program analysis. For example, programming language data type facilities provide "sums of products" data types, which means simply unions of (usually distinct) data type constructors. All such data types can be expressed as set constraints.

Let $X = Y$ stand for the pair of constraints $X \subseteq Y$ and $Y \subseteq X$. Consider the constraint

$$\beta = \mathtt{cons}(\alpha, \beta) \cup \mathtt{nil}$$

If `cons` and `nil` are interpreted in the usual way, then the solution of this constraint assigns to β the set of all lists with elements drawn from α. This example also shows that a special operation for recursion is not required in the set expression language—recursion is obtained naturally through recursive constraints.

The set of non-nil lists (with elements drawn from α) can be defined as $\gamma = \beta \cap \neg\mathtt{nil}$, where β is defined as above. The set γ is useful because it describes the proper domain of the function that selects the first element of a list; such a function is undefined for empty lists. This example also illustrates that set constraints can describe proper subsets of standard sums of products data types.

The final example shows a non-trivial set of constraints where some work is required to derive the solutions. Consider the universe of the natural numbers with one unary constructor `succ` and one nullary constructor `zero`. Let the system of constraints be:

$$\mathtt{succ}(\alpha) \subseteq \neg\alpha \bigwedge \mathtt{succ}(\neg\alpha) \subseteq \alpha$$

These constraints say that if $x \in \alpha$ (resp. $x \in \neg\alpha$) then $\mathtt{succ}(x) \in \neg\alpha$ (resp. $\mathtt{succ}(x) \in \alpha$). In other words, these constraints have two solutions, one where α is the set of even integers and one where α is the set of odd integers. The solutions are described by the following equations:

$$\alpha = \mathtt{zero} \cup \mathtt{succ}(\mathtt{succ}(\alpha))$$
$$\alpha = \mathtt{succ}(\mathtt{zero}) \cup \mathtt{succ}(\mathtt{succ}(\alpha))$$

Note that the two solutions are incomparable; in general, there is no least solution of a system of set constraints.

3 Results and Open Problems

The set constraint language defined in Section 2 is henceforth called the *basic language*. There are several interesting extensions to the basic language, each of which substantially alters the set constraint problem. Three extensions are discussed in this paper: projections, function spaces, and negative constraints.

For every constructor c of arity n, a family of *projections* $c^{-1}, \ldots, c^{-n}$ can be defined such that

$$\sigma(c^{-i}(E)) = \{t_i | \exists t_1, \ldots, t_n . c(t_1, \ldots, t_n) \in \sigma(E)\}$$

To date, projections are used primarily in set constraint analyses for logic programming languages [HJ90b].

A separate extension is adding sets of functions $X \rightarrow Y$ to the set expressions. This is a major change, because it not only enriches the language, but also requires a new domain. The construction of a suitable domain with function spaces is beyond the scope of this paper; somewhat surprisingly, however, given such a domain, set constraint techniques still apply. In an appropriate domain, the meaning of $X \rightarrow Y$ is

$$X \rightarrow Y = \{f | x \in X \Rightarrow f(x) \in Y\}$$

Function spaces are used primarily in the analysis of functional programming languages [AW93, AWL94].

Finally, negative constraints are strict containments $X \not\subseteq Y$. Negative constraints can express the set of non-solutions of a system of positive constraints:

$$\neg \bigwedge_i (X_i \subseteq Y_i) = \bigvee_i X_i \not\subseteq Y_i$$

Since conjunctions of positive constraints correspond to an existential property (i.e., is any assignment a solution of the constraints) disjunctions of negative constraints can express universal properties (i.e., is every assignment a solution of the constraints) [AKW93, GTT93].

Four proofs of decidability of the satisfiability problem for the basic language are known [AW92, GTT92, BGW93, AKVW93]. Remarkably, each proof is based on completely different techniques. A particularly elegant proof is due to Bachmair, Ganzinger, and Waldmann [BGW93]; their result shows set constraints are equivalent to the *monadic class*, the class of first order formulas with arbitrary quantification but only unary predicates and no function symbols. In addition to satisfiability, constraint resolution algorithms are known that construct explicit representations of the solutions of systems of set constraints for the basic language.

The situation with the various extensions is less clear. Table 1 summarizes the current state of knowledge. The decidability of the satisfiability of set constraints with projections was open for several years [HJ90a] and has only very recently been resolved [CP94b]. Constraint resolution algorithms for restricted forms of the general problem are known [HJ90a, Hei92]; the current state of the art permits the full basic language and restricts only projections [BGW93].

Work on set constraints extended with negative constraints has been motivated in part because it is an intermediate step toward handling projections. To see this, consider the expression $c^{-1}(c(X, Y))$. Note that if $Y = 0$, then

$c(X, Y) = 0$, since constructors function as cross products. Therefore, the meaning of this expression can be characterized as

$$c^{-1}(c(X, Y)) = \begin{cases} 0 \text{ if } Y = 0 \\ X \text{ if } Y \neq 0 \end{cases}$$

Thus, even a restricted form of projection implicitly involves negative constraints ($Y \neq 0$ in the right-hand side above). Three independent proofs of the decidability of set constraints with negative constraints have been discovered [AKW93, GTT93, CP94a]; currently there is only one reported proof of the decidability of set constraints with projections [CP94b]. These are decision procedures only, however, and do not characterize the solution sets.

Problem	*Satisfiability*	*Constraint Resolution*
basic	yes	yes
basic with projections	yes	with restrictions
basic with function spaces	yes	with restrictions
basic with negative constraints	yes	?

Table 1. Status of set constraint problems.

Set constraints extended with function spaces have been used to develop very expressive subtype inference systems for functional languages. Currently, constraint solving algorithms for a fairly general class of set constraints with function types are known [AW93, AWL94]. Damm has proven the surprising result that satisfiability of set constraints with function spaces is decidable [Dam94].

Set constraint resolution algorithms can be computationally expensive in general. For the basic problem, deciding satisfiability is NEXPTIME-complete [BGW93] and even if the language is restricted to the set operations over constants satisfiability remains NP-complete [AKVW93]. By restricting the set operations (instead of the arity of constructors) it is possible to achieve polynomial time algorithms for interesting classes of constraints [JM79, MR85, Hei92].

4 Algorithms

At the current time, the literature on set constraint algorithms is very diverse in many dimensions, with a wide variety of notation and algorithmic techniques in use. Unfortunately, no reference provides a systematic introduction to more than a small portion of the body of existing work. This section gives a very brief and relatively informal overview of the basic algorithmic issues in solving systems of set constraints. For a more detailed treatment of the various algorithms, the interested reader should consult sources listed in the bibliography.

All set constraint resolution algorithms have the same basic structure. An initial system of constraints is systematically transformed until the constraints reach a particular syntactic *solved form*. In most cases, the solved form is equivalent to one or more regular tree grammars. More precisely, the final result is a set of equations

$$\alpha = c(X_1, \ldots, X_n) \cup \ldots \cup d(Y_1, \ldots, Y_m)$$

which can viewed equivalently as the productions of a grammar

$$\alpha ::= c(X_1, \ldots, X_n) \mid \ldots \mid d(Y_1, \ldots, Y_m)$$

The language generated by the tree grammar then describes the solution of the constraints.

Unfortunately, this simple explanation of the solutions of set constraints is a bit oversimplified. In reality, set constraints are more general than tree grammars. In the solutions of set constraints, this extra generality appears as "free" variables in the solved form equations. A free variable is one that does not appear on the left-hand side of any equation. Thus, a more accurate description of the solutions of set constraints is that they are tree grammars that may include free variables.

At their core, all set constraint algorithms have two characteristic forms of constraints: transitive constraints and structural constraints. Transitive constraints arise from combining upper and lower bounds on variables:

$$X \subseteq \alpha \ \wedge \ \alpha \subseteq Y \Rightarrow X \subseteq Y$$

Because of the need to resolve transitive constraints, most interesting set constraint problems have at least $\mathcal{O}(n^3)$ time complexity.

Structural constraints are constraints between constructor expressions:

$$c(X_1, \ldots, X_n) \subseteq c(Y_1, \ldots, Y_n)$$

In general, there may be many incomparable solutions of such a constraint. For example, because the semantics of a constructor is essentially a cross product, a constructor expression is 0 if any component is 0, and therefore the constraint is satisfied if $X_i = 0$ for any i. Of course, the constraint is also satisfied if $X_i \subseteq Y_i$ for all i. Thus, the complete set of solutions is

$$c(X_1, \ldots, X_n) \subseteq c(Y_1, \ldots, Y_n) \Leftrightarrow X_1 = 0 \vee \ldots \vee X_n = 0 \vee (X_1 \subseteq Y_1 \wedge \ldots \wedge X_n \subseteq Y_n)$$

Searching for a solution of such a constraint requires guessing a disjunct that can be satisfied. This non-deterministic choice increases the complexity of set constraint problems above the complexity of the corresponding tree automata problems. For example, deciding whether the language of one tree automata is a subset of another is complete for EXPTIME [Sei90]; solving a general system of set constraint inclusions is complete for NEXPTIME.

If it is known that the system of constraints under consideration has a least solution and the goal is to compute only the least solution, then it is easy to see that the cases $X_i = 0$ need not be considered and the last case can be chosen deterministically. Thus, more efficient algorithms are possible in the special case that a system of constraints has a least solution.

Finally, the set operators $\cap, \cup$, and $\neg$ play roles very similar to their roles in other logics. There are some distributive laws involving constructors, but these are not surprising:[1]

$$\begin{aligned}
c(X_1, \ldots, X_n) \cap c(Y_1, \ldots, Y_n) &= c(X_1 \cap Y_1, \ldots, X_n \cap Y_n) \\
c(X_1 \cup Y_1, Z_2, \ldots, Z_n) &= c(X_1, Z_2, \ldots, Z_n) \cup c(Y_1, Z_2, \ldots, Z_n) \\
\neg c(X_1, \ldots, X_n) &= c(\neg X_1, 1, \ldots, 1) \cup \ldots \cup c(1, \ldots, 1, \neg X_n) \cup \\
&\quad \bigcup_{d \neq c} d(1, \ldots, 1)
\end{aligned}$$

For set constraint problems with restricted set operations and where the constraints have least solutions, it is possible to design polynomial time algorithms to compute the least solution; for examples, see [JM79, MR85, Hei92, Hei94]. If the set operations are not restricted, then it becomes possible to describe some complex sets of terms very succinctly with set expressions, which raises the computational complexity of constraint resolution to exponential time.

5 Applications

Set constraints have a long history and, in fact, less general formalisms predate the term "set constraints" by many years. The basic language of set constraints is now known to be equivalent to the monadic class of logical formulas [BGW93]; the first decision procedure for the monadic class was given by Löwenheim in 1915 [L1̈5]. Within the realm of computer science, Reynolds was the first to develop a resolution algorithm for a class of set constraints [Rey69]. Reynolds was interested in the analysis and optimization of Lisp programs. In this application, set constraints were used to compute a conservative description of the data structures in use at a program point. Using this information, a Lisp program could be optimized by, for example, eliminating run-time type checks where it was provably safe to do so.

Independently of Reynolds, Jones and Muchnick developed a different analysis system for Lisp programs based on solving systems of set equations [JM79]. This analysis was used not only to eliminate dynamic type checks but also to reduce reference count operations in automatic memory management systems based on reference counting. Recently Wang and Hilfinger have proposed another analysis method for Lisp based on set equations [WH92].

[1] As written, the law for negation appears to require that the set of all constructors d such that $d \neq c$ can be enumerated and thus the set of constructors must be finite. In fact, this restriction is not necessary, and it is a simple matter to implement negation for infinite sets of constructors.

A different set of applications provide type inference algorithms for functional languages that verify the type correctness of a larger class of programs than the standard Hindley/Milner type system. Mishra and Reddy described a type system based on a set constraint resolution algorithm that could handle considerably more complex constraints than previous algorithms [MR85]. Thatte introduced *partial types* [Tha88], the type inference problem for which, while substantially different from earlier systems, is also reducible a set constraint resolution problem. The most recent work in this area is due to Wimmers and the author [AW93, AWL94], who provide a type inference system that generalizes the results in [MR85, Tha88]. An implementation of this last system is publicly available (see Section 6).

A natural application area for set constraints is the analysis of logic programs. The idea was first explored by Mishra [Mis84]; more recently, this line of work has been well developed in a series of papers by Jaffar and Heintze [HJ90b, HJ90a, HJ92], as well as in Heintze's thesis [Hei92]. Many of the techniques developed in [Hei92] have been fruitfully applied to compile time analysis in other areas, especially the compile-time analysis of ML programs [Hei94].

6 Conclusions and Directions

Interest in set constraints originally arose from the needs of researchers working in program analysis. Currently, there is a lively, continuing interplay between the theoretical and practical efforts in the area. Future work is most likely to proceed along three lines. First, the open problems in Table 1 may be resolved. Second, efforts to apply set constraints to new problems will lead to additional variations on the basic language. Third, there will be additional effort devoted to the efficient implementation of set constraint resolution algorithms. This is likely to include not only new engineering techniques, but also exploration of restricted classes of constraints for which good worst-case complexity results can be obtained.

Besides a number of prototype or special purpose systems, there are currently two substantial, complete set constraint resolution implementations, one by Nevin Heintze at CMU [Hei92] and one by the author and colleagues at IBM. The latter implementation is available by anonymous ftp and comes with a type inference system for a functional language based on solving systems of set constraints [AWL94]. To get this system, retrieve `pub/personal/aiken/Illyria.tar.Z` from the machine `s2k-ftp.cs.berkeley.edu`.

References

[AKVW93] A. Aiken, D. Kozen, M. Vardi, and E. Wimmers. The complexity of set constraints. In *Computer Science Logic '93*, Swansea, Wales, September 1993. To appear.

[AKW93] A. Aiken, D. Kozen, and E. Wimmers. Decidability of systems of set constraints with negative constraints. Research Report RJ 9421, IBM, 1993.

[AL94] A. Aiken and T.K. Lakshman. Directional type checking of logic programs. In *Proceedings of the 1st International Static Analysis Symposium*, September 1994. To appear.

[AM91] 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.

[AW92] A. Aiken and E. Wimmers. Solving systems of set constraints. In *Symposium on Logic in Computer Science*, pages 329–340, June 1992.

[AW93] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 1993.

[AWL94] A. Aiken, E. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, Portland, Oregon, January 1994.

[BGW93] L. Bachmair, H. Ganzinger, and U. Waldmann. Set constraints are the monadic class. In *Symposium on Logic in Computer Science*, pages 75–83, June 1993.

[CP94a] W. Charatonik and L. Pacholski. Negative set constraints wtih equality: An easy proof of decidability. In *Symposium on Logic in Computer Science*, July 1994. To appear.

[CP94b] W. Charatonik and L. Pacholski. Set constraints with projections are in NEXPTIME. In *Foundations of Computer Science*, 1994. To appear.

[Dam94] F. M. Damm. Subtyping with union types, intersection types and recursive types. In *Proceedings of the International Symposium on Theoretical Aspects of Computer Software*. Springer-Verlag, April 1994. To appear.

[GTT92] R. Gilleron, S. Tison, and M. Tommasi. Solving systems of set constraints using tree automata. In *Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science*, pages 505–514, 1992.

[GTT93] R. Gilleron, S. Tison, and M. Tommasi. Solving Systems of Set Constraints with Negated Subset Relationships. In *Foundations of Computer Science*, pages 372–380, November 1993.

[Hei92] N. Heintze. *Set Based Program Analysis*. PhD thesis, Carnegie Mellon University, 1992.

[Hei94] N. Heintze. Set-based analysis of ML programs (extended abstract). In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, June 1994. To appear.

[HJ90a] N. Heintze and J. Jaffar. A decision procedure for a class of Herbrand set constraints. In *Symposium on Logic in Computer Science*, pages 42–51, June 1990.

[HJ90b] 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.

[HJ91] N. Heintze and J. Jaffar. Set-based program analysis. Draft manuscript, 1991.

[HJ92] N. Heintze and J. Jaffar. An engine for logic program analysis. In *Symposium on Logic in Computer Science*, pages 318–328, June 1992.

[JM79] 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.

[Lö15] L. Löwenheim. Über möglichkeiten im relativkalkül. *Math. Annalen*, 76:228–251, 1915.

[Mis84] P. Mishra. Towards a theory of types in PROLOG. In *Proceedings of the First IEEE Symposium in Logic Programming*, pages 289–298, 1984.

[MR85] P. Mishra and U. Reddy. Declaration-free type checking. In *Proceedings of the Twelfth Annual ACM Symposium on the Principles of Programming Languages*, pages 7–21, 1985.

[Rey69] J. C. Reynolds. *Automatic Computation of Data Set Definitions*, pages 456–461. Information Processing 68. North-Holland, 1969.

[Sei90] H. Seidl. Deciding equivalence of finite tree automata. *SIAM Journal of Computing*, 19(3):424–437, June 1990.

[Tha88] S. Thatte. Type inference with partial types. In *Automata, Languages and Programming: 15th International Colloquium*, pages 615–629. Springer-Verlag Lecture Notes in Computer Science, vol. 317, July 1988.

[WH92] E. Wang and P. N. Hilfinger. Analysis of recursive types in Lisp-like languages. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 216–225, June 1992.

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