# CLP(SC): Implementation and Efficiency Considerations

Jeffrey S. Foster
Computer Science Department
Cornell University
Ithaca, NY 14850
jfoster@cs.cornell.edu

**Abstract**

CLP(SC) is a constraint logic programming language over set constraints proposed by Kozen [7]. In this paper, we describe a complete C++ implementation of CLP(SC). We describe the data structures used to represent systems of set constraints and an efficient algorithm, a modification of one given in [7], for unifying constraints. In addition, we investigate two further techniques for increasing efficiency: keeping track of variable equalities and doing PROLOG-style unification.

## 1 Introduction

A *set constraint* is an inclusion or equality between two expressions representing sets of ground terms over a finite alphabet. CLP(SC) is a constraint logic programming language over set constraints proposed by Kozen [7]. This paper describes a complete C++ implementation of CLP(SC), filling in the details necessary to go from the high-level algorithmic descriptions of [7] to a working program. Although we do not currently have enough empirical evidence to state that our system is highly efficient, preliminary tests show that it may indeed be so.

CLP(SC) allows positive inclusions and equalities between arbitrary set expressions composed with union, intersection, and complement. It is known [1] that solving systems of such constraints is *NEXPTIME*-complete. Our main contributions are to show that in practice, systems of constraints can be solved fairly efficiently, and that the hypergraph representation used in [1, 7] is in fact as practical for implementations as other techniques [2, 5, 6]. In order to gain this efficiency, we developed an algorithm for conjoining new constraints to a pre-existing constraint system without doing the expensive cross-product–like operation suggested in [7].

In addition, we describe two other methods to increase efficiency that may be of interest in the context of general constraint solving: using syntactic analysis (PROLOG-style unification) and taking advantage of variable equalities. In fact, our implementation is designed to be modular. It should be easy to replace the hypergraph-based constraint solver with another style of solver, or, alternately, to extract the solver from the program and use it in another context. Although the interpreter only responds "yes" or "no" to a query clause, in fact the solver does build a complete representation of all solutions to the generated constraints.

This paper is organized as follows. In section 2, we give a grammar for CLP(SC). In section 3, we describe the algorithms used to solve set constraints. In section 4, we give two additional measures, syntactic analysis and variable equalities, to increase efficiency. In section 5, we describe the results of this work, and in section 6 we discuss future work and some open problems.

## 2 The Language

The syntax and operation of CLP(SC) is nearly identical to PROLOG, except that goals may be set constraints and solving set constraints replaces unification. The grammar given here is exactly the grammar given by Kozen [7].

The objects in the language represent sets of ground terms. Ground terms themselves are promoted to singleton sets. In this way, there is no syntactic distinction between terms and sets, and membership can be written with an inclusion. Arbitrary regular sets [7] of ground terms, including infinite ones, can be computed. See [3] for an example of a different language, or [7] for comparisons with other languages.

CLP(SC) works much the same as a standard PROLOG interpreter. Program clauses are stored in a database. Resolution uses the usual top-down, left-to-right rule. Where PROLOG would try to unify terms, CLP(SC) calls the constraint solver.

## 2.1 Grammar

Here is the grammar for CLP(SC), as proposed in [7]. We have replaced non-typeable characters by more computer-friendly symbols. A CLP(SC) program consists of a series of *program clauses*, and running a program corresponds to asking whether a *query clause* is satisfiable. These clauses are of the form

| | | |
|---|---|---|
| $A\ :-$ | $B_1, \ldots, B_n.$ | Program clause |
| $?-$ | $B_1, \ldots, B_n.$ | Query clause |

where $A$ is an atomic formula and the $B_i$ are either atomic formulae or set constraints. An *atomic formula* is of the form

$$pred(S_1, \ldots, S_k)$$

where the $S_i$ are set expressions and *pred* denotes a predicate. A *set constraint* is one of

| | |
|---|---|
| $S_1$ = $S_2$ | $S_1$ and $S_2$ are equal as sets |
| $S_1$ <= $S_2$ | $S_1$ is a subset of $S_2$ |

where $S_1$ and $S_2$ are set expressions.

Finally, a *set expression* is one of

| | |
|---|---|
| 0 | Empty set |
| 1 | Set of all ground terms |
| *variable-id* | Variable |
| *functor*$(S_1, \ldots, S_n)$ | Functor |
| $S_1$ + $S_2$ | Union of $S_1$ and $S_2$ |
| $S_1$ * $S_2$ | Intersection of $S_1$ and $S_2$ |
| $\sim S$ | Complement of $S$ |
| $(S)$ | $S$ |

To run a program, one asks whether a query clause is satisfiable or not. The interpreter responds "yes" or "no." Although not currently done, it should be possible to construct sample satisfying assignments to the variables. This may be somewhat problematical, however, since solutions can be infinite sets. As a compromise, in a debugging mode offered by our implementation, we output the actual internal representation of the system.

As in PROLOG, variables are capitalized, while functors are procedures are not. Functors and procedure names are distinguished from each other syntactically. We use a generalized DeMorgan law [7] to handle negated set expressions. Applying this law requires that the set of all functors is known. Accordingly, the interpreter for CLP(SC) has special predicates to declare the existence of functors.

## 2.2 Example

The following is an actual CLP(SC) program that runs under our interpreter. These are exactly the sample predicates given in [7].

```
?- const(a).
?- const(b).
```

```
?- unary(f).
?- unary(g).

// sng(X) == X is a singleton set
sng(a).
sng(b).
sng(f(X)) :- sng(X).
sng(g(X)) :- sng(X).

empty(0).

nonempty(X) :- Y <= X, sng(Y).

equal(X, X).

unequal(X, Y) :- nonempty(X*(~Y)+Y*(~X)).

// dbl(X) == X is a doubleton set
dbl(Y+Z) :- unequal(Y, Z), sng(Y), sng(Z).

// atleast2(X) == X has at least 2 elements
atleast2(X) :- Y<=X, dbl(Y).
```

The built-in predicates `const` and `unary` declare that $a$ and $b$ have arity 0 while $f$ and $g$ have arity 1. We currently do not allow greater arities.

# 3 Solving Set Constraints

## 3.1 Atomic Form and Hypergraphs

In [7], Kozen describes a normal form for set constraints called *atomic form* and shows how set constraints in this form can be interpreted as a hypergraph. CLP(SC) solves constraints by building this hypergraph and testing for closure, the equivalent of satisfiability. As set constraints are added and removed from the system, the hypergraph is updated. The following definition is taken directly from [7].

Let $\Sigma$ be a finite ranked alphabet. Denote by $\Sigma_n$ the $n$-ary elements of $\Sigma$. We refer to the elements of $\Sigma$ as functors.

**Definition:** A system of set constraints is in *atomic form* if

- the variables are partitioned into two disjoint sets $U$ and $X$, called the *atoms* and *primary variables*
- there is a subset $E_f(\bar{u}) \subseteq U$ for each $f \in \Sigma_n$ and $\bar{u} \in U^n$
- there is a subset $P(x) \subseteq U$ for each $x \in X$

such that the system consists of constraints

$$\begin{aligned}
\bigcup_{u \in U} u &= 1 \\
u \cap v &= 0, \text{ for distinct } u, v \in U \\
f(\bar{u}) &\subseteq \bigcup_{u \in E_f(\bar{u})} u \\
x &= \bigcup_{u \in P(x)} u, \text{ for } x \in X
\end{aligned}$$

where any $f(\bar{u})$ appears on at most one left hand side of a constraint.

Using this definition, the tuple $(U, X, E, P)$ describes a system of set constraints in atomic form. The elements of $U$, which can be thought of as maximal conjunctions of the primary variables $X$ or their negations, form the nodes of the hypergraph. The elements of $E$ are the edges. $X$ are the variables given in the CLP(SC) program, and $P$ describes the relationship between $X$ and $U$. Although the elements of $U$ are variables, we will refer to them exclusively as atoms or as nodes, and when we use the term variables, we will mean elements of $X$.

## 3.2 Data Structures

### 3.2.1 Further Considerations

Before describing in detail the data structures and algorithms, we should point out that the mechanisms used in the CLP(SC) language – resolution and backtracking – require that more than the basic algorithms are implemented. Given an existing system of set constraints, stored as a hypergraph, it must be possible to both add new constraints (to support resolution) and to undo previous additions of constraints (to support backtracking).

In addition, in order to make the algorithms more reasonable, we only allow constant and unary functors in our implementation. The algorithms are easily, but messily, extended to binary or greater functors. In fact, in our implementation, everything except the graph object and the edge building and closure routines support general $n$-ary functors.

### 3.2.2 Internal Representations

A system of set constraints $(U, X, E, P)$ in atomic form is represented internally by several data structures. The vertices of the graph contain assignments to the variables. For each variable $x$, two sets $P(x)$ and $\sim P(x)$ are maintained; $P(x)$ is the set of vertices in which $x$ is assigned true, and $\sim P(x)$ is the set of vertices in which $x$ is assigned false. $P(x) \cup \sim P(x) = U$, where $U$ is the set of all vertices. Although storing this information in two places is redundant, it makes the program more efficient.

For each vertex, we maintain a set of in edges and out edges. Because only unary and constant functors are allowed, the hypergraph is actually just a graph. For each constant functor $a \in \Sigma_0$ we maintain a set $E_a \subseteq U$. Constraints on the unary functors are represented by edges in the graph.

### 3.2.3 Backtracking

When failure occurs while searching for a path through the database, and there exists another potential path (a backtrack point), some number of the most recent modifications to the constraint system must be undone. This requires a limited form of *persistent* data structures [4]. Although for some applications, such as an incremental compiler, it may be desirable to have full persistence (the ability to maintain and update arbitrary, distinct versions of the data structures), for our purposes it is sufficient to support a simple kind of partial persistence. Specifically, (1) Modifications to a structure are only required of the current version, and (2) If we modify a previous version, it is safe to discard all changes made since that version.

Each new version of the data structure has a distinct *version number*. For each low-level operation on a data structure (for example, inserting an edge in a graph or removing an element from a set $E_a$), we push a record of the operation on a stack and increment the current version number. Then the actual change is made to the structure. To undo modifications, we need only pop the change records off the stack, doing the operations in reverse until we have reached the desired version.

This is not, of course, the only method possible for implementing backtracking. Another equally straightforward method would be to, after each update, make a copy of the whole constraint system. However, we felt that this would be too expensive, since it may potentially duplicate a lot of information. As mentioned in the results section, the method we chose, which does not duplicate any unnecessary information, works well in practice.

## 3.3 Building the Hypergraph

### 3.3.1 Partial Reduction to Atomic Form

The hypergraph is constructed from the original set of constraints using the algorithm in [7], modified substantially to be more efficient and to allow the addition of constraints one by one. This first step, partial reduction to atomic form, is exactly as outlined in [7], except that some effort is made to avoid introducing excess variables.

Given a set of constraints $C$, we first create two new sets of constraints, $F$ and $B$. The constraints $F$ correspond to edges in the hypergraph; the constraints $B$ describe the atoms that make up the vertices of the graph. A *literal* is a variable or a negated variable. Most of the work of the first phase is done in the flattening step, which separates out constraints on the functors into a set $F'$.

**Algorithm: Flattening**. Given a set expression $e$, convert $e$ to an expression containing only literals and constants.

1. If $e$ is a constant or literal, do nothing.
2. If $e$ is of the form $f(t_1, \dots, t_n)$, flatten $t_1, \dots, t_n$, replace the expression $e$ with a new variable $x$, and add the constraint $f(l_1, \dots, l_n) = x$ to $F'$, where the $l_i$ are the flattened $t_i$.
3. Otherwise, flatten each subexpression of $e$.

Using the flattening procedure, the system of constraints $C$ is separated into $F$ and $B$:

**Algorithm: Partial Atomic Form**.

1. For each constraint $x \subseteq y$ or $x = y$ in $C$, flatten the expressions $x$ and $y$. Any functors have been shifted into $F'$, so the flattened constraint is purely boolean. Add it to $B$.
2. Each constraint in $F'$ is now in the form $f(l_1, \dots, l_n) = x$, where the $l_i$ are literals or constants. We replace each equality by two inclusions. For each of these constraints,
   (a) Add the constraint $f(l_1, \dots, l_n) \subseteq x$ to $F$.
   (b) For each $g \neq f$, add the constraint $g(1, \dots, 1) \subseteq \sim x$ to $F$.
   (c) For $1 \leq i \leq n$, add the constraints $f(1, \dots, 1, \sim l_i, 1, \dots, 1) \subseteq \sim x$ to $F'$, where $\sim l_i$ is the $i$th argument.

This last step uses the generalized DeMorgan law [7] to replace $\sim f(l_1, \dots, l_n)$. It is here that knowing $\Sigma$ is essential to the algorithm. After applying this algorithm, all constraints have been separated into boolean constraints, $B$, or constraints of the form $f(x_1, \dots, x_n) \subseteq y$, where $x_1, \dots, x_n$, and $y$ are literals or constants. We refer to this as partial atomic form.

### 3.3.2 Updating the Hypergraph

After putting the constraints $C$ into partial atomic form $(B, F)$, the next step is building the hypergraph. Two complications make this the most difficult part of the interpreter.

- Because the interpreter searches for a path through the database, it must be possible to add constraints to an already existing hypergraph and to undo the last few updates to the hypergraph.
- The vertices of the hypergraph represent distinct assignments to the variables that satisfy the boolean constraints. Given the boolean constraints $B$, constructing these satisfying assignments is an *NP*-complete problem.

Kozen [7] gave a method for building separate systems of set constraints from scratch and then conjoining them using a cross-product–like operation. That technique did not address the issue of finding satisfying assignments to the variables or the need for backtracking. In addition, it may require a minimization step afterward to keep the size of the systems manageable.

Instead, we chose a related but substantially different method. We maintain only one hypergraph, always minimized, at all times. Backtracking is handled as described above. In order to find satisfying assignments, we introduce new variables one by one, which in practice eliminates many nodes from consideration. We can do this because, given new constraints $C$ over variables $Y$, we know that the previous set of variables $X$ is a subset of $Y$. This is a consequence of the language.

**Algorithm: Updating a Constraint System.** Given an already-constructed constraint system $(U, X, E, P)$, a set of constraints $B$ and $F$ in partial atomic form, and a set of new variables $Y$ for $B$ and $F$, construct a new constraint system $(U', Y, E', P')$ that is equivalent to the old system conjoined with $B$ and $F$. Assume $X \subseteq Y$.

1. Erase any assignment $u \in U$ that does not satisfy the constraints $B$.
2. If the constraint system is empty (not equivalent to $U = \emptyset$, since $U = \emptyset$ may indicate an unsatisfiable system of constraints) and $Y \neq \emptyset$,
   (a) Pick some $y$ in $Y$.
   (b) Create the two assignments $\sigma_t = [y \rightarrow true]$ and $\sigma_f = [y \rightarrow false]$.
   (c) Create appropriate node(s) for whichever assignments satisfy $B$.
   (d) Fully connect the resulting graph and let $X = \{y\}$.
3. For each $y \in Y - X$ and for each $u \in U$
   (a) Let $\sigma$ be the assignment stored in $u$. Construct the two new assignments $\sigma_t = \sigma[y \rightarrow true]$ and $\sigma_f = \sigma[y \rightarrow false]$.
   (b) If exactly one of $\sigma_t$ and $\sigma_f$ satisfies $B$, replace $\sigma$ in $u$ by the satisfying assignment. Add $u$ to the appropriate set $P(y)$ or $\sim P(y)$.
   (c) If both satisfy $B$, duplicate node $u$ and store $\sigma_t$ in $u$ and $\sigma_f$ in the copy. Add $u$ to $P(y)$ and the duplicate node to $\sim P(y)$. To duplicate node $u$, create a new node, and copy all edges and self-loops to the new node.
   (d) If neither satisfy $B$, delete node $u$.

Note that $\sigma_t$ and $\sigma_f$ may only be partial functions from $Y$ to $\{0, 1\}$. If the algorithm encounters an unassigned variable, it simply assumes that the variable could have either value. Thus, expressions can evaluate to 0, 1, or don't-know.

Let $(U', Y, P', E)$ be the constraint system after this process. This system represents the old system conjoined with the constraints $B$. The next step is to add the constraints $F$. Recall that all the elements of $F$ are of the form $f(x_1, \ldots, x_n) \subseteq y$, where $x_1, \ldots, x_n, y$ are literals or constants.

Define the following map $\tau$ from literals and constants to $2^{U'}$:

$$
\begin{aligned}
\tau(0) &= \emptyset \\
\tau(1) &= U' \\
\tau(x) &= P(x) \\
\tau(\sim x) &= \sim P(x)
\end{aligned}
$$

For each constraint $f(x_1, \ldots, x_n) \subseteq y$, replace the set $E_f(\tau(x_1) \times \cdots \times \tau(x_n))$ with its intersection with $\tau(y)$. For constant functors $a$, this is equivalent to $E_a = E_a \cap \tau(y)$. For unary functors $f$, this is equivalent to $E_f(\tau(x_1)) = E_f(\tau(x_1)) \cap \tau(y)$.

The new system, $(U', Y, P', E')$ represents the old constraints conjoined with both $B$ and $F$.

### 3.4 Testing Satisfiability

Satisfiability of a constraint system $C$ put into atomic form $(U, X, P, E)$ is equivalent to closure of the hypergraph [7]. To test satisfiability, the interpreter first computes the largest closed induced subgraph of the graph $(U, E)$, applying the following step until no longer possible:

- Find a node $u \in U$ such that, for some unary functor $f$, $E_f(u) = \emptyset$. Delete node $u$.

This step discards nodes that cannot possibly contribute to satisfying the constraint system. Afterwards, the system is satisfiable iff (1) $U \neq \emptyset$; and (2) For all constants $a$, $E_a \neq \emptyset$.

## 4 Shortcuts

The method just described fully satisfies all the demands made by the interpreter and does so in a fairly efficient manner. The flattening procedure doesn't flatten anything it doesn't need to. The method for finding new satisfying assignments avoids potential exponential blow up of the algorithms, at least in our tests. We have also experimented with two other techniques to increase efficiency.

### 4.1 Variable Equalities

A constraint system $(U, X, E, P)$ must have $|U| \leq 2^{|X|}$, since there are at most $2^{|X|}$ assignments to $|X|$ variables. Looked at another way, each new variable added to a constraint system can potentially double the number of nodes in the graph. Thus, it seems that an essential trick to making CLP(SC) run efficiently is to somehow limit the number of variables.

As a consequence of the language, the interpreter will generate many constraints of the form $x = y$, where $x$ and $y$ are variables. For example, such a constraint will be generated in matching the goal *foo(x)* with the clause

$$foo(y) \quad :- \quad y <= a, bar(y).$$

In this situation, there is absolutely no reason to create a new variable $y$, since it must be identical to the variable $x$. To capture this equivalence and to prevent the creation of a new variable, we store a union-find forest of equivalent variables. Each tree of the forest contains a set of variables known to be equivalent. When the solver is called with new constraints, we first search through the constraints for ones of the form $x = y$, and join together the appropriate trees in the forest.

Unfortunately, although trial runs have shown that indeed this discovers many equivalences, we have not been able to make good use of this information. The updating algorithm does not have any obvious places where such equivalences are useful. The one place we do use it is to avoid testing satisfying assignments. Given a variable $y \in Y - X$, if $y$ is equivalent to a variable already assigned, we just assign it to the value of that variable, bypassing the evaluation step.

### 4.2 Syntactic Analysis

No matter how efficient, running set constraints through the full-blown constraint solver is potentially quite expensive. In many cases the full power of the solver is unnecessary. For example, the constraints $0 = 1, a = b, f(a) = f(a)$ are clearly unsatisfiable, unsatisfiable, and satisfiable, respectively.

Because set expressions are defined over an Herbrand domain, in certain, limited cases the standard unification procedure of PROLOG can solve set constraints. In our system, we make use of the following lemma, which can be proven using the axioms for termset algebra as given in [7].

**Lemma:** The constraint $f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)$ is satisfiable iff $n = m$, $f = g$, and for $1 \leq i \leq n$, $s_i = t_i$ is satisfiable.

**Algorithm: Syntactic Analysis.** For each constraint $c \in C$, if $c$ is of the form $f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)$, check that $n = m$ and $f = g$. If they don't, declare failure. Otherwise, replace $c$ by the constraints $s_i = t_i, 1 \leq i \leq n$. If $c$ is not of this form, ignore it. Continue until no more constraints of the form $f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)$ exist.

We apply this algorithm before running the given constraints $C$ through the full constraint solver. It is safe to halt the algorithm and declare failure because an unsatisfiable system can never be made satisfiable by conjoining additional constraints. That in itself may save unnecessary work. In addition, this technique can potentially yield a very large savings, since the flattening step adds new variables for each functor expression it encounters, and this algorithm bypasses flattening.

# 5 Results

Our implementation of CLP(SC) allows programs with any number of constant and unary functors, and arbitrary combinations of unions, intersections, and complements. With these allowances, the algorithms are *EXPTIME*-complete [1] (not *NEXPTIME*, since we only allow constant and unary functors). However, preliminary tests show that, in practice, the running time of the program is tractable.

On a SPARC10 using SOLARIS, running the query *dbl(f(a)+a)* takes only two seconds, yet it builds up a constraint system with 14 variables. Although there could be a tremendous number of nodes in the graph, there are actually only 32. Even with 32 nodes, the graph could be dense, but instead all the nodes have only two out edges. We take advantage of this sparsity by using edge lists in the graph.

Our straightforward implementation of backtracking works well and quickly. It adds only constant extra overhead to each operation, and preliminary tests show that the interpreter spends most of the time in running the constraint solver, rather than in backtracking.

Syntactic analysis is not always useful. For the query *dbl(f(a)+a)*, syntactic analysis leads to no simplifications. However, for the query *sng(f(a))*, only syntactic analysis is used. We haven't yet found an example in the middle ground.

The variable equalities seem to be extremely useful, and programs would probably run faster if we made better use of them. For example, in the case of the query *dbl(f(a)+a)*, 264 out of 364 times, a variable was known to be equivalent to a variable already assigned a value.

Although this evidence is incomplete, partially because we do not yet have a "real" CLP(SC) program, it gives us reason to believe that our system is efficient and warrants further study.

# 6 Future Work

There is a problem with the example program. Consider the predicate *sng*. *sng(x)* succeeds *if* $x$ is a singleton set. But if $x$ is *not* a singleton set, then the predicate does not halt. The reason for this is that there are potentially an infinite number of sets that $x$ needs to be compared to before deciding it is not a singleton set. We are currently trying to implement *sng* as a built-in predicate.

This raises several questions. If something as simple as *sng* needs to be built-in, are there equally fundamental programs that cannot be expressed in the language? Is this limitation important?

The algorithm for updating the variable assignments stored in the nodes of the graph does not specify the order in which the variables should be chosen. In our implementation, for several practical reasons, the variables are ordered according to their position in the clauses. Is there a better ordering? How much difference does that make? It seems clear that the notion of equivalent variables could be better used. How could we do that?

Because we allow negations of set expressions, the interpreter needs to know $\Sigma$. Is there any way in which we could avoid this? One technique would be to scan the database and extract information about all the functors used in the program. That is insufficient, however, because it may be that some element of $\Sigma$ just happens to not be mentioned. An ideal solution would allow functors to be incrementally added to a given constraint system in the same way that set constraints are.

## Software

We have implemented CLP(SC) in C++. The source code is available at `ftp.cs.cornell.edu/pub/jfoster`. The program is modular in design, and it should be easy to either replace our constraint solver with another, or to extract our solver and use it in another context. Any C++ compiler that supports the Standard Template Library and exception handling should be able to compile the code. We have successfully built the software using GCC 2.7.2 under the SOLARIS operating system and using CodeWarrior 8 on a Macintosh.

## Acknowledgments

We are indebted to Dexter Kozen for many valuable ideas and suggestions, and for making this whole project possible. We would also like to thank Alex Aiken for helpful comments, and fellow CS486 students, especially Joon Lee, for work on an earlier prototype of this system. The support of the National Science Foundation under grant CCR-9317320 is gratefully acknowledged.

## References

[1] A. Aiken, D. Kozen, M. Vardi, and E. Wimmers. The complexity of set constraints. In *Proc. 1993 Conf. Computer Science Logic (CSL '93)*, volume 832 of *Lecture Notes in Computer Science*, pages 1–17. Eur. Assoc. Comput. Sci. Logic, Springer-Verlag, September 1994.

[2] A. Aiken and E. Wimmers. Solving systems of set constraints. In A. Scedrov, editor, *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 329–340. IEEE, June 1992.

[3] A. Dovier, E. Pontelli, E. Omodeo, and G. Rossi. Embedding finite sets in a logic programming language. In E. Lamma and P. Mello, editors, *Proc. Third International Workshop Extensions of Logic Programming (ELP '92)*, volume 660 of *Lecture Notes in Artificial Intelligence*, pages 150–167. Springer-Verlag, February 1992.

[4] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. Making data structures persistent. In *Proceedings of the Eigteenth Annual ACM Symposium on Theory of Computing*, pages 109–121. ACM, May 1986.

[5] R. Gilleron, S. Tison, and M. Tommasi. Solving systems of set constraints using tree automata. In P. Enjalbert, A. Finkel, and K. W. Wagner, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS '93)*, volume 665 of *Lecture Notes in Computer Science*, pages 505–514. Springer-Verlag, February 1993.

[6] N. Heintze. *Set Based Program Analysis.* PhD dissertation, Carnegie Mellon University, Department of Computer Science, October 1992.

[7] D. Kozen. Set constraints and logic programming. Technical Report TR94-1467, Cornell University, Computer Science Department, November 1994.