Set Constraints and Logic Programming

Dexter Kozen

Computer Science Department, Cornell University, Ithaca, NY 14853-7501, USA

Set constraints are inclusion relations between expressions denoting sets of ground terms over a ranked alphabet. They are the main ingredient in set-based program analysis [4,5,15,16,19,23,24,26]. In this paper we describe a constraint logic programming language CLP(SC) over set constraints in the style of Jaffar and Lassez [17]. The language subsumes ordinary logic programs over an Herbrand domain. We give an efficient unification algorithm and operational, declarative, and fixpoint semantics. We show how the language can be applied in set-based program analysis by deriving explicitly the monadic approximation of the collecting semantics of Heintze and Jaffar [15,16].

1 Introduction

Set constraints are inclusion relations between expressions denoting sets of ground terms over a ranked alphabet Σ. The language of set constraints contains the usual Boolean operators along with a set operator f for each n-ary $f \in \Sigma$ with interpretation

$$f(A_1, \ldots, A_n) = \{f(t_1, \ldots, t_n) \mid t_i \in A_i,\ 1 \le i \le n\}\ .$$

In *set-based program analysis* [4,5,15,16,19,23,24,26], set constraints are used to represent *monadic* properties of program variables; all interdependencies are ignored. Although information is lost, enough is retained to allow useful program optimization and type inference, and the resulting systems remain decidable [2,3,6,7,9,13,14,27].

Heintze and Jaffar [16] and Heintze [15] apply set-based program analysis in both the imperative and logic programming settings. They first give a least fixpoint characterization of the sets of valuations of program variables that can occur at each point in a program during execution; this is called the *collecting semantics*. These sets are of course nonrecursive. They then give a *monadic approximation* to the collecting semantics in which variable dependencies are

Preprint submitted to Elsevier Preprint *19 October 1995*

ignored. This gives a superset of the actual set of values, but one can still derive useful inferences about program behavior, and the sets of values obtained are recursive. The monadic approximation has a least fixpoint characterization almost identical to the characterization of the collecting semantics, except that the basic operators are interpreted as set operators.

One might desire a language in which algorithms in set-based program analysis can be easily expressed. In this paper we introduce a logic programming language CLP(SC) for this purpose. The language CLP(SC) is a constraint logic programming language in the style of Jaffar and Lassez [17] using set constraints over an Herbrand domain.

Sets of ground terms satisfy many nice algebraic properties. An axiomatization of these properties was proposed in [20] (see §2.1 below). Models of these axioms are called *termset algebras*. The axioms of termset algebra are reminiscent of the Clark axioms for Herbrand domains; in fact, constraint logic programming over set constraints and conventional logic programming over Herbrand domains have much in common. In many ways, one can think of CLP(SC) as an intermediate stage between logic programming over an Herbrand domain and constraint logic programming in general.

The language CLP(SC) subsumes ordinary logic programming over an Herbrand domain, since ground terms can be identified with singleton sets, and singleton sets are definable in CLP(SC).

There have been several previous approaches to augmenting logic programming languages with sets. Jayaraman and Plaisted [18] present a language in the equational programming style which combines relational, subset, and equational assertions. Operational and fixpoint semantics are given. A *collect all* property is posed as part of the semantics, which plays the same role as minimal models or least fixpoints in logic programming. Kuper [22] presents a language with two types of objects, individuals and sets, and a membership predicate. Program clauses

$$A :- \forall x_1 \in X_1 \ \ldots \forall x_n \in X_n \ B_1, \ \ldots, \ B_m.$$

are allowed, where the X_i are terms denoting finite sets. Kuper mentions a suitable treatment of negation as an important open problem. Dovier *et al.* [10] present a language with membership and equality predicates for finite sets and a constructor *with* for adding new elements to sets. Constraints are used in the unification process. Stolzenburg [28,29] introduces a logic programming language with finite sets in which membership is dealt with via constraints. These approaches concentrate on the set unification problem.

Our approach differs from these in several ways. We have only one type of object, namely sets of ground terms, and no explicit membership predicate. Single ground terms are identified with singleton sets, and the membership predicate is encoded using the subset predicate. The domain of computation consists of all regular sets of ground terms, including infinite regular sets. Any such set can be uniquely specified by a finite collection of set constraints. All Boolean operations, including negation, are allowed. Negations are dealt with using a generalized DeMorgan law.

Frühwirth *et al.* [12] have also shown how to express the monadic approximation using logic programs. However, their approach is quite different: they transform a given logic program into another logic program such that the latter computes exactly the monadic approximation of the former. They work with a conventional logic programming language over an Herbrand domain and do not discuss set constraints.

The present paper is organized as follows. In §2, we review the basic theory of set constraints. In §3, we describe the syntax of the language CLP(SC) and give three equivalent semantics: operational, fixpoint, and declarative. In §4, we discuss techniques for solving set constraints, including the definition of a useful normal form. In §5, we give a unification algorithm based on the constraint satisfaction algorithm of [3], as well as some heuristics which may improve performance. Finally, in §6, we show how the language can be applied in set-based program analysis by deriving explicitly the monadic approximation to the collecting semantics of Heintze and Jaffar [15,16].

2 Set Expressions and Set Constraints

Let Σ be a finite ranked alphabet consisting of symbols f, each with an associated arity. Symbols in Σ of arity 0, 1, 2, 3, and n are called *nullary, unary, binary, ternary*, and *n-ary*, respectively. Nullary elements are often called *constants*. The set of elements of Σ of arity n is denoted Σ_n. The use of any expression of the form $f(x_1, \ldots, x_n)$ in the sequel carries the implicit assumption that f is of arity n.

The set of ground terms over Σ is denoted T_Σ. This is the smallest set such that if $t_1, \ldots, t_n \in T_\Sigma$ and $f \in \Sigma_n$, then $f(t_1, \ldots, t_n) \in T_\Sigma$. If $X = \{x, y, \ldots\}$ is a set of variables, then $T_\Sigma(X)$ denotes the set of terms over Σ and X, considering the elements of X as symbols of arity 0.

Let $\mathsf{B} = (\cup, \cap, \sim, 0, 1)$ be the usual signature of Boolean algebra. Other Boolean operators such as $-$ (set difference) and $\oplus$ (symmetric difference) are defined from these as usual. Let $\Sigma + \mathsf{B}$ denote the signature consisting of the disjoint union of Σ and B. A *set expression* over variables X is any element of

$T_{\Sigma+\mathsf{B}}(X)$. The following is a typical set expression:

$$f(g(x \cup y), {\sim} g(x \cap y)) \cup a$$

where $f \in \Sigma_2$, $g \in \Sigma_1$, $a \in \Sigma_0$, and $x, y \in X$. A *Boolean expression* over X is any element of $T_{\mathsf{B}}(X)$.

A *positive set constraint* is a formal inclusion $s \subseteq t$, where s and t are set expressions. We also allow equational constraints $s = t$, although inclusions and equations are interdefinable: $s \subseteq t$ is equivalent to $s \cup t = t$, and $s = t$ is equivalent to $s \oplus t \subseteq 0$. A *negative set constraint* is the negation of a positive set constraint: $s \not\subseteq t$ or $s \neq t$.

We interpret set expressions over the powerset 2^{T_Σ} of T_Σ. This forms an algebra of signature $\Sigma + \mathsf{B}$, where the Boolean operators have their usual set-theoretic interpretations and elements $f \in \Sigma_n$ are interpreted as functions

$$\begin{aligned} f &: (2^{T_\Sigma})^n \to 2^{T_\Sigma} \\ f(A_1, \ldots, A_n) &= \{f(t_1, \ldots, t_n) \mid t_i \in A_i,\ 1 \leq i \leq n\}\ . \end{aligned} \tag{1}$$

Later, we will restrict our attention to the subalgebra Reg_Σ of regular subsets of T_Σ.

A *set valuation* is a map $\sigma : X \to 2^{T_\Sigma}$ assigning a subset of T_Σ to each variable in X. Any set valuation σ extends uniquely to a $(\Sigma + \mathsf{B})$-homomorphism $\sigma : T_{\Sigma+\mathsf{B}}(X) \to 2^{T_\Sigma}$ by induction on the structure of set expressions in the usual way. We say that the set valuation σ satisfies the positive constraint $s \subseteq t$ if $\sigma(s) \subseteq \sigma(t)$, and satisfies the negative constraint $s \not\subseteq t$ if $\sigma(s) \not\subseteq \sigma(t)$. We write $\sigma \models \varphi$ if the set valuation σ satisfies the constraint φ. A system $\mathcal{C}$ of set constraints is *satisfiable* if there is a set valuation σ that satisfies all the constraints in $\mathcal{C}$; in this case we write $\sigma \models \mathcal{C}$ and say σ is a *solution* of $\mathcal{C}$.

2.1 Axioms of Termset Algebra

In [20], the following axiomatization of the algebra of sets of ground terms was introduced:

$$f(\ldots, x \cup y, \ldots) = f(\ldots, x, \ldots) \cup f(\ldots, y, \ldots) \tag{2}$$

$$f(\ldots, x - y, \ldots) = f(\ldots, x, \ldots) - f(\ldots, y, \ldots) \tag{3}$$

$$\bigcup_{f \in \Sigma} f(1, \ldots, 1) = 1 \tag{4}$$

$$f(1, \ldots, 1) \cap g(1, \ldots, 1) = 0\ , \quad f \neq g \tag{5}$$

$$f(x_1, \ldots, x_n) = 0 \rightarrow \bigvee_{i=1}^{n} (x_i = 0) \tag{6}$$

and the axioms of Boolean algebra. The ellipses in (2) and (3) indicate that the explicitly given arguments occur in corresponding places, and that implicit arguments in corresponding places agree. Models of these axioms are called *termset algebras.*

The standard interpretation 2^{T_Σ} forms a model of these axioms. Another model is given by the subalgebra Reg_Σ of regular subsets of T_Σ.

Some immediate consequences of these axioms are

$$\begin{aligned}
f(\ldots, 0, \ldots) &= 0 && (7)\\
f(\ldots, \sim x, \ldots) &= f(\ldots, 1, \ldots) - f(\ldots, x, \ldots) && (8)\\
f(\ldots, x \oplus y, \ldots) &= f(\ldots, x, \ldots) \oplus f(\ldots, y, \ldots) && (9)\\
f(\ldots, x \cap y, \ldots) &= f(\ldots, x, \ldots) \cap f(\ldots, y, \ldots) && (10)\\
x \subseteq y \Rightarrow f(\ldots, x, \ldots) &\subseteq f(\ldots, y, \ldots) \,. && (11)
\end{aligned}$$

One particularly important consequence is the *generalized DeMorgan law*:

$$\sim f(x_1, \ldots, x_n) = \bigcup_{g \neq f} g(1, \ldots, 1) \cup \bigcup_{i=1}^{n} f(\underbrace{1, \ldots, 1}_{i-1}, \sim x_i, \underbrace{1, \ldots, 1}_{n-i}) \,. \tag{12}$$

This law is useful in pushing occurrences of the negation operator $\sim$ down to the leaves of a term. This law can be justified intuitively as follows. The expression $f(x_1, \ldots, x_n)$ denotes the set of all ground terms with head symbol f and i^{th} subterm satisfying x_i. A term is *not* of this form if either its head symbol is not f (hence the first clause on the right hand side of (12)) or its head symbol is f, but its i^{th} subterm does not satisfy x_i for some i (hence the second clause on the right hand side). Formally, the law can be derived from the termset algebra axioms by purely equational reasoning.

3 CLP(SC)

In this section we describe a logic programming language CLP(SC), a constraint logic programming language in the style of Jaffar and Lassez [17] over set constraints. We describe the syntax of the language and give three equivalent semantics: operational, declarative or model-theoretic, and fixpoint. The equivalence of these three semantics follows from standard results and techniques of constraint logic programming [17].

3.1 *Examples*

Before describing the syntax and semantics of the language CLP(SC), here are some sample programs to whet the intuition.

- Consider the clauses

  $$sng(a).$$
  $$sng(f(x_1,\ldots,x_n))\ \text{:-}\ sng(x_1),\ \ldots,\ sng(x_n).$$

  for all constants $a \in \Sigma$ and function symbols $f \in \Sigma$ of arity $n \geq 1$. The goal $sng(x)$ succeeds iff x is a singleton set.
- For the goal $empty(x)$ to succeed iff x is the empty set:

  $$empty(0).$$

- For the goal $nonempty(x)$ to succeed iff x is not the empty set:

  $$nonempty(x)\text{:-}\,y \subseteq x,\ sng(y).$$

- For the goal $equal(x,y)$ to succeed iff x and y are equal as sets:

  $$equal(x,x).$$

- For the goal $unequal(x,y)$ to succeed iff x and y are unequal as sets:

  $$unequal(x,y)\text{:-}\,nonempty(x \oplus y).$$

- For the goal $dbl(x)$ to succeed iff x is a doubleton set:

  $$dbl(y \cup z)\text{:-}\,unequal(y,z),\ sng(y),\ sng(z).$$

- For the goal $atleast2(x)$ to succeed iff x contains at least two elements:

  $$atleast2(x)\text{:-}\,y \subseteq x,\ dbl(y).$$

Ordinary logic programming over the Herbrand domain T_Σ is subsumed, since ground terms can be identified with singleton sets, which are definable using $sng(x)$. The membership predicate is encoded using the subset predicate. Negative constraints are also obviated by the use of $sng(x)$, using the fact that a set is nonempty iff it includes a singleton subset (although this in itself does not give a decision procedure for negative constraints).

3.2 *Syntax of* CLP(SC)

Let $\Pi = \{p,q,r,\ldots\}$ be a ranked alphabet of relation symbols not containing $=$ or $\subseteq$, each with a fixed finite arity. Let Π_n denote the set of elements of Π

of arity n. An *atomic formula* is an expression of the form $p(\bar{u})$, where $p \in \Pi_n$ and $\bar{u} = u_1, \ldots, u_n$ is an n-tuple of set expressions. A *program clause* is either

$A.$
A :- $B_1, \ldots, B_n.$

where A is an atomic formula and the B_i are either atomic formulas or positive set constraints. A *program* π is a set of program clauses. A *query* is an expression of the form

?- $B_1, \ldots, B_n.$

where the B_i are either atomic formulas or positive set constraints.

3.3 Regular Sets

A subset of T_Σ is *regular* if it is described by a finite tree automaton; equivalently, if it is some set x_1 described by a system of simultaneous set equations of the form

$$\begin{aligned} x_1 &= s_1(x_1, \ldots, x_m) \\ x_2 &= s_2(x_1, \ldots, x_m) \\ &\vdots \\ x_m &= s_m(x_1, \ldots, x_m) \end{aligned} \tag{13}$$

in which each variable x_i occurs on the left hand side of exactly one equation and each right hand side is a disjunction of set expressions of the form $f(y_1, \ldots, y_n)$, where $f \in \Sigma_n$ and $y_i \in \{x_1, \ldots, x_m\}$, $1 \leq i \leq n$. It can be proved by induction on the depth of terms that any such system has a unique solution (see [11]). The family of regular sets over Σ is denoted Reg_Σ. For example, the system

$$x = a \cup g(y) \qquad y = g(x) \tag{14}$$

has the unique regular solution

$$\sigma(x) = \{g^n(a) \mid n \text{ even}\} \qquad \sigma(y) = \{g^n(a) \mid n \text{ odd}\} \ .$$

Gilleron *et al.* [13] have shown that every satisfiable system of set constraints has a regular solution, *i.e.* one in which all variables are interpreted as regular sets. We give an alternative proof of this fact below (Theorem 7).

For our domain of computation we take the family Reg_Σ of regular subsets of T_Σ. We contend that this domain in the present context is analogous to the Herbrand universe in ordinary logic programming. One might alternatively consider the sets represented by the family of ground set expressions, *i.e.* elements of $T_{\Sigma+\mathsf{B}}$. However, this set is too small, because there are satisfiable systems of set constraints with no solution in $T_{\Sigma+\mathsf{B}}$: (14), for example. On the other hand, the entire power set of T_Σ is too big, since there are subsets of T_Σ that are not represented by any finite system of set constraints.

The choice of the regular sets as domain of computation allows us to think conveniently in terms of a generalized notion of *substitution*: if A is any expression involving the set variables $\bar{x} = x_1, \ldots, x_n$, and if $\bar{d} = d_1, \ldots, d_n$ is an n-tuple of regular sets described uniquely by a finite system $\mathcal{C}$ of set constraints of the form (13), then the "substitution instance" $A[\bar{x}/\bar{d}]$ can be expressed syntactically by conjoining $\mathcal{C}$ and A.

The domain of regular sets also satisfies the two fundamental desiderata for constraint logic programming languages as set forth in [17], namely:

- Every element of the domain is the unique solution of a finite or infinite family of constraints. In fact, every regular set is the unique solution of a finite family of constraints of the form (13).
- Every element not satisfying a constraint C satisfies some constraint C' such that the conjunction C, C' is unsatisfiable. This property follows immediately from the fact that every regular set is the unique solution of a single constraint obtained by combining the constraints (13):

$$\bigcup_{i=1}^{m}(x_i \oplus s_i(x_1, \ldots, x_m)) = 0 \ .$$

3.4 Operational Semantics

In the following, $\mathcal{C}, \mathcal{C}'$ denote finite systems of set constraints; $\mathcal{B}, \mathcal{B}'$ finite lists of atomic formulas; p an element of Π_n; $\bar{s}, \bar{t}$ n-tuples of set expressions; and π a program.

Following [17], our operational semantics involves sequences of one-step derivations of the form

$$p(\bar{s}),\ \mathcal{B},\ \mathcal{C} \xrightarrow[\pi]{1} \bar{s} = \bar{t},\ \mathcal{B},\ \mathcal{B}',\ \mathcal{C},\ \mathcal{C}' \tag{15}$$

which reduces the goal on the left hand side to the goal on the right hand side whenever

– there is a fresh instantiation

$$p(\bar{t}) :\!- \mathcal{B}',\ \mathcal{C}'.$$

of a program clause in π obtained by substituting new variables; and
– the constraint system $\bar{s} = \bar{t},\ \mathcal{C},\ \mathcal{C}'$ is satisfiable.

There is no implied ordering of the atomic formulas in a goal; any one may be chosen for expansion at any time.

We say that the query

$$?\!- \mathcal{B},\ \mathcal{C}. \tag{16}$$

succeeds if there is a sequence

$$\mathcal{B},\ \mathcal{C} \xrightarrow[\pi]{*} \mathcal{C}' \tag{17}$$

of such one-step derivations eliminating all atomic formulas, and $\mathcal{C}'$ is satisfiable. Here $\xrightarrow[\pi]{*}$ denotes the reflexive transitive closure of $\xrightarrow[\pi]{1}$. If σ is a set valuation, we say that the query (16) *succeeds with* σ if there is a derivation (17) with $\sigma \models \mathcal{C}'$. Note that σ also satisfies the original constraint system $\mathcal{C}$.

3.5 Declarative Semantics

Let

$$\Delta = \{p(\bar{d}) \mid n \geq 0,\ p \in \Pi_n,\ \bar{d} \in \mathrm{Reg}_\Sigma^n\}\ .$$

The set Δ corresponds to the *Herbrand base* of ordinary logic programming.

We consider first-order structures $\mathcal{M}$ with carrier Reg_Σ, set operations and relations $\cup, \cap, \sim, 0, 1, =, \subseteq$ with their usual interpretations, $f \in \Sigma$ with set-theoretic interpretation (1), and interpretations of relation symbols in Π specified by some subset $\Delta^{\mathcal{M}}$ of Δ. If $\sigma : X \rightarrow \mathrm{Reg}_\Sigma$, we write

$$\mathcal{M}, \sigma \models \varphi$$

if $\mathcal{M}$ satisfies the first-order formula φ under valuation σ in the ordinary sense of first-order logic. We write $\mathcal{M} \models \pi$ if $\mathcal{M}$ satisfies the clauses in the program π, considered as universally quantified Horn clauses of first-order logic.

3.6 Fixpoint Semantics

For $\Gamma \subseteq \Delta$, let $T_\pi(\Gamma)$ be the set of all $p(\bar{d}) \in \Delta$ such that there exists a program clause

$$A \text{:-}\, B_1,\ \ldots,\ B_m,\ \mathcal{C}.$$

in π and a set valuation $\sigma : X \to \mathrm{Reg}_\Sigma$ such that

- $B_i[\bar{x}/\sigma(\bar{x})] \in \Gamma$, $1 \leq i \leq m$
- $\sigma \models \mathcal{C}$, and
- $p(\bar{d}) = A[\bar{x}/\sigma(\bar{x})]$.

The map $T_\pi : 2^\Delta \to 2^\Delta$ is monotone with respect to set inclusion, therefore by the Knaster-Tarski Theorem has a least fixpoint Δ_π. Let $\mathcal{M}_\pi$ be the model specified by Δ_π as described in §3.5; *i.e.*, $\Delta^{\mathcal{M}_\pi} = \Delta_\pi$.

The following results assert the equivalence of these three semantics. The proofs are standard, using results and techniques of logic programming and constraint logic programming [17].

Lemma 1 *The set $\Delta^{\mathcal{M}}$ is a prefixpoint of T_π (i.e., $T_\pi(\Delta^{\mathcal{M}}) \subseteq \Delta^{\mathcal{M}}$) if and only if $\mathcal{M} \models \pi$.*

By the Knaster-Tarski Theorem, the least prefixpoint of T_π is also its least fixpoint. It follows that $\mathcal{M}_\pi$ is the minimal model of π.

Theorem 2 *Let $\mathcal{B}$ be a finite list of atomic formulas, $\mathcal{C}$ a finite system of set constraints, $\bar{d} = d_1, \ldots, d_m \in \mathrm{Reg}_\Sigma$, σ a partial set valuation such that $\sigma(x_i) = d_i$, $1 \leq i \leq m$, where $\bar{x} = x_1, \ldots, x_m$ is a list of variables including all those occurring in $\mathcal{B}$ and $\mathcal{C}$, and $\mathcal{D}$ a system of set constraints of the form (13) defining the substitution $[\bar{x}/\bar{d}]$ uniquely.*

The following statements are equivalent:

(i) $\mathcal{M}_\pi, \sigma \models \mathcal{B} \wedge \mathcal{C}$;
(ii) the query ?–$\mathcal{B}, \mathcal{C}$. succeeds with some extension σ' of σ;
(iii) the query ?–$\mathcal{B}, \mathcal{C}, \mathcal{D}$. succeeds;
(iv) $\sigma \models \mathcal{C}$, *and for every clause B_i in $\mathcal{B}$, $B_i[\bar{x}/\bar{d}] \in \Delta_\pi$.*

4 Efficient Constraint Solving

4.1 Atomic Form and Hypergraphs

In this section we describe a convenient normal form for systems of constraints called *atomic form.* This normal form corresponds to the combinatorial method of [2,3,20] involving hypergraphs. It is also strongly related to the automata-theoretic approach of [13,14] and to the approach of [7] involving finite models of monadic logic.

Definition 3 *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*, respectively,*
- *there is a subset $E_f(\bar{u}) \subseteq U$ for each $f \in \Sigma_n$ and $\bar{u} \in U^n$, and*
- *there is a subset $P(x) \subseteq U$ for each $x \in X$,*

such that the system consists of constraints

$$\bigcup_{u \in U} u = 1 \tag{18}$$

$$u \cap v = 0 \ , \qquad \text{for distinct } u, v \in U \tag{19}$$

$$f(\bar{u}) \subseteq \bigcup_{u \in E_f(\bar{u})} u \tag{20}$$

$$x = \bigcup_{u \in P(x)} u \ , \qquad x \in X \tag{21}$$

where any $f(\bar{u})$ appears on at most one left hand side of a constraint of the form (20). We take $E_f(\bar{u}) = U$ for expressions $f(\bar{u})$ not appearing on the left hand side of any constraint (20); this implicitly asserts the redundant constraint $f(\bar{u}) \subseteq 1$.

The tuple (U, X, E, P) specifies a system of set constraints in atomic form, where U is the set of atoms, X the set of primary variables, E specifies the maps $E_f : U^n \rightarrow 2^U$, and P gives the sets $P(x)$.

The clauses (18) and (19) say that the atoms form a finite partition of T_Σ. As in [2,3], we can regard such a system as a hypergraph on vertices U with hyperedge relations

$$E_f : U^n \rightarrow 2^U \ ,$$

one for each $f \in \Sigma_n$. For constants $a \in \Sigma_0$, E_a is a subset of U, unary $g \in \Sigma_1$ give rise to ordinary binary edge relations, binary $f \in \Sigma_2$ give rise to ternary hyperedge relations, *etc.* This structure can also be regarded as a nondeterministic finite tree set automaton [13,14].

Definition 4 ([2]) *The hypergraph corresponding to a system of set constraints in atomic form is said to be* closed *if every $E_f(\bar{u})$ is nonempty. The hypergraph is said to have a* closed induced subhypergraph *if there is a subset $V \subseteq U$ such that for every $f \in \Sigma_n$ and every n-tuple $\bar{u} \in V^n$, the set $E_f(\bar{u})$ intersects V.*

The notion of closure is captured axiomatically by (6) [20].

Definition 5 *A* run *is a map $\theta : T_\Sigma \to U$ such that for all $f(t_1, \ldots, t_n) \in T_\Sigma$,*

$$\theta(f(t_1, \ldots, t_n)) \in E_f(\theta(t_1), \ldots, \theta(t_n)) \ . \tag{22}$$

The run θ corresponds to an infinite run of a tree set automaton in the automata-theoretic approach of [13,14].

The following theorem was proved in [2].

Theorem 6 ([2]) *Let $\mathcal{C} = (U,\ X,\ E,\ P)$ be a system of set constraints in atomic form considered as a hypergraph as described above. The following three statements are equivalent:*

(i) $\mathcal{C}$ has a closed induced subhypergraph;
(ii) there exists a run $\theta : T_\Sigma \to U$;
(iii) $\mathcal{C}$ is satisfiable.

Proof sketch. (i) → (ii) The existence of a closed induced subhypergraph on atoms V allows us to assign an atom $\theta(t) \in V$ to each ground term $t \in T_\Sigma$ inductively such that (22) holds.

(ii) → (iii) Given a run θ, a set valuation σ satisfying $\mathcal{C}$ can be obtained by setting

$$\sigma(x) = \theta^{-1}(P(x)) \qquad \sigma(u) = \theta^{-1}(\{u\}) \ . \tag{23}$$

(iii) → (i) Given valuation σ satisfying $\mathcal{C}$, take $V = \{u \in U \mid \sigma(u) \neq \varnothing\}$. □

If there is a closed induced subhypergraph not containing some atom u, then u is not needed to construct a run θ, and its removal does not affect satisfiability.

We will often (but not always) want to annihilate such atoms. This is done formally by imposing the extra set constraint $u = 0$, then using property (7) and Boolean algebra to construct an equisatisfiable system in atomic form in which the atom u does not appear. For each occurrence of u on the left hand side of a constraint (20), by (7) that constraint is immediately satisfied and may be deleted. Any other occurrence of u may then be deleted, since it only appears in disjunctions. We are left with a smaller system in atomic form.

4.2 Reduction to Atomic Form

Every system of set constraints can be put into atomic form effectively with at most an exponential increase in size. Here is an algorithm, which is essentially the same as the normal form algorithm of [2].

Let X be the set of variables appearing in the original system. These are the *primary variables.*

Algorithm 1 *(i) Replace any subexpression $f(t_1, \ldots, t_n)$ by x and add constraints*

$$\begin{aligned} x &= f(y_1, \ldots, y_n) \\ y_i &= t_i \ , \quad 1 \leq i \leq n \ , \end{aligned} \tag{24}$$

where $x, y_1, \ldots, y_n$ are new auxiliary variables. This is called flattening. *Repeat until the system consists of purely Boolean constraints and constraints of the form (24).*

(ii) Replace each constraint of the form (24) by two inclusions

$$f(y_1, \ldots, y_n) \subseteq x \qquad \sim f(y_1, \ldots, y_n) \subseteq \sim x \ . \tag{25}$$

(iii) Apply the generalized DeMorgan law (12) to the left hand side of (25) to get the equivalent inclusion

$$\bigcup_{\substack{g \in \Sigma \\ g \neq f}} g(1, \ldots, 1) \cup \bigcup_{i=1}^{n} f(\underbrace{1, \ldots, 1}_{i-1}, \sim y_i, \underbrace{1, \ldots, 1}_{n-i}) \subseteq \sim x \ ,$$

then rewrite this as separate inclusions

$$\begin{aligned} g(1, \ldots, 1) &\subseteq \sim x \ , \qquad g \neq f \\ f(\underbrace{1, \ldots, 1}_{i-1}, \sim y_i, \underbrace{1, \ldots, 1}_{n-i}) &\subseteq \sim x \ , \qquad 1 \leq i \leq n \ . \end{aligned}$$

All constraints are now either purely Boolean or of the form

$$f(x_1, \ldots, x_n) \subseteq x \tag{26}$$

where $x, x_1, \ldots, x_n$ are positive or negative literals or the constant 1.

(iv) *Let Y be the set of variables in use at this point. This includes the primary variables X and all auxiliary variables added in step (i). Let $\mathcal{B}$ be the set of purely Boolean constraints on Y constructed in step (i). Introduce a new set of variables U called* atoms, *one for each atom of the free Boolean algebra on generators Y modulo $\mathcal{B}$; equivalently, one for each truth assignment to Y satisfying $\mathcal{B}$. For $x \in Y$, let $P(x)$ be the set of all $u \in U$ such that the truth assignment corresponding to u satisfies x. Replace the constraints $\mathcal{B}$ with the constraints (18), (19), and (21) for each $x \in Y$.*

(v) *In constraints of the form (26), replace each positive literal x with $\bigcup_{u \in P(x)} u$, each negative literal $\sim x$ with $\bigcup_{u \notin P(x)} u$, and each occurrence of the constant 1 with $\bigcup_{u \in U} u$. Apply (2) to express each left hand side as a union of expressions of the form $f(u_1, \ldots, u_n)$. Separate each resulting constraint*

$$\bigcup_{\bar{u} \in A} f(\bar{u}) \subseteq \bigcup_{u \in E} u$$

into a finite collection of constraints

$$f(\bar{u}) \subseteq \bigcup_{u \in E} u \ , \qquad \bar{u} \in A \ .$$

(vi) *Collect all constraints with the same left hand side,*

$$f(\bar{u}) \subseteq \bigcup_{u \in E} u \ , \qquad E \in \mathcal{E} \ ,$$

and let $E_f(\bar{u}) = \bigcap \mathcal{E}$. Replace these constraints with the single equivalent constraint (20).

(vii) *Remove all constraints of the form (21) for auxiliary variables, i.e. those in $Y - X$. They are no longer needed (and trivially satisfiable if the rest of the system is).*

The resulting system is in atomic form and is equivalent to the original.

One can still reduce the size of the system by annihilating atoms u that are inaccessible in the automata-theoretic sense, since they will never be chosen in the construction of the run θ in Theorem 6. Formally,

(viii) *Let W be the smallest set closed under the following operation: if $\bar{u} \in W^n$ then $E_f(\bar{u}) \subseteq W$. Annihilate all atoms $u \in U - W$. If U has a closed induced subhypergraph on atoms V, then the induced subhypergraph on atoms $V \cap W$ is also closed, therefore by Theorem 6 the new system is satisfiable iff the old one was.*

4.3 Testing Satisfiability

If the system $\mathcal{C}$ of set constraints in atomic form is not closed, then there is some constraint of the form

$$f(u_1, \ldots, u_n) \subseteq 0 \ . \tag{27}$$

Property (6) then implies that any satisfying valuation must have $u_i = 0$ for some i, $1 \leq i \leq n$. We can pick some u_i and annihilate it as described above. However, if some $E_g(\bar{u}) = \{u_i\}$, then this last action causes the right hand side of another constraint (20) to vanish, in which case the process must be repeated. If this process ever stabilizes in a system in atomic form in which every $E_f(\bar{u})$ is nonempty, then we have found a closed induced subhypergraph, and by Theorem 6 the system is satisfiable.

The choice of u_i to annihiliate is inherently a nondeterministic process. No algorithm that is significantly more efficient in the worst case is likely to be found, since the general satisfiability problem is nondeterministic exponential-time complete [2,7], and even *NP*-complete when the system is in atomic form. However, if there are no operators of arity two or greater, then there is no nondeterministic choice to be made and the process becomes deterministic. This is the essence of the proof of the result of [2] that the satisfiability problem can be solved in deterministic exponential time in this case.

Even in the presence of operators of arity two or greater, the following greedy heuristic may be useful in improving performance: always annihilate the u_i that removes the largest number of constraints (20) with 0 on the right hand side.

Aiken [1] also suggests the following heuristic: keep track of atoms that are necessary to the solution. For example, if $\bar{u} = u_1, \ldots, u_n$ are all necessary and $E_f(\bar{u}) = \{u'\}$, then u' is necessary. Necessary atoms should never be annihilated. Initially, few, if any, atoms will be necessary. However, as choices are made about which atoms to annihilate, the set of necessary atoms will increase, leading to more deterministic search in later steps.

4.4 Regular Solutions

In this section we give an alternative proof of a result of Gilleron *et al.* [13] that we can restrict our attention to regular solutions of systems of set constraints. This result is essential in the semantics of CLP(SC).

Theorem 7 ([13]) *Every satisfiable system of set constraints has a regular solution.*

PROOF. Let $\mathcal{C}$ be a satisfiable system of set constraints in atomic form. By Theorem 6, the associated hypergraph contains a closed induced subhypergraph; *i.e.*, one can annihilate atoms u to obtain an equisatisfiable system in atomic form in which all $E_f(\bar{u})$ are nonempty. Now perform the following steps in order:

(i) Delete all atoms but one from each $E_f(\bar{u})$.
(ii) Annihilate all atoms except those appearing on the right hand sides of inclusions (20).
(iii) Combine all constraints (20) with the same right hand side u into a single constraint whose left hand side is the disjunction of the left hand sides of all constraints with right hand side u.
(iv) Change all inclusions to equalities.

Each step in the above process strengthens the system (annihilation of u is tantamount to adding the constraint $u = 0$), so any solution of the resulting system is also a solution of the original system $\mathcal{C}$. The resulting system of equations (20) is of the form (13), which has a unique regular solution (see [11]). Moreover, every $f(\bar{u})$ occurs in exactly one equation (20); this implies that (18) and (19) hold as well.

This procedure constructs a closed subhypergraph (not necessarily induced) in which all $E_f(\bar{u})$ are singletons, which can be viewed as a deterministic tree set automaton. □

5 Efficient Unification

In constraint logic programming, unification is just conjunction of constraints. In our case, however, we wish to maintain constraints in atomic form for the sake of efficiency. We show in this section an efficient way to unify two constraint systems $\mathcal{C}$, $\mathcal{D}$ in atomic form into a new constraint system $\mathcal{E}$ in atomic form that is equivalent to the conjunction of $\mathcal{C}$ and $\mathcal{D}$. This is done in two steps: the first, a *common refinement step* in which atoms from $\mathcal{C}$ and $\mathcal{D}$ are paired; and a *minimization step* in which inaccessible atoms are annihilated and equivalent atoms coalesced.

5.1 Common Refinement

Let $\mathcal{C} = (U^{\mathcal{C}},\ X^{\mathcal{C}},\ E^{\mathcal{C}},\ P^{\mathcal{C}})$ and $\mathcal{D} = (U^{\mathcal{D}},\ X^{\mathcal{D}},\ E^{\mathcal{D}},\ P^{\mathcal{D}})$ be two systems of set constraints in atomic form with disjoint sets of atoms. We unify $\mathcal{C}$ and $\mathcal{D}$ by forming their *coarsest common refinement.* The resulting system will be in atomic form and will be equivalent to the conjunction of $\mathcal{C}$ and $\mathcal{D}$.

For $u \in U^{\mathcal{C}}$ and $v \in U^{\mathcal{D}}$, let uv denote a new variable which is formally the ordered pair (u, v) but represents the conjunction $u \cap v$. Define the system $\mathcal{E} = (U^{\mathcal{E}},\ X^{\mathcal{E}},\ E^{\mathcal{E}},\ P^{\mathcal{E}})$ as follows:

$$U^{\mathcal{E}} = \bigcap_{x \in X^{\mathcal{C}} \cap X^{\mathcal{D}}} ((P^{\mathcal{C}}(x) \times P^{\mathcal{D}}(x)) \cup ((U^{\mathcal{C}} - P^{\mathcal{C}}(x)) \times (U^{\mathcal{D}} - P^{\mathcal{D}}(x)))) \quad (28)$$

$$X^{\mathcal{E}} = X^{\mathcal{C}} \cup X^{\mathcal{D}} \quad (29)$$

$$E^{\mathcal{E}}_f(u_1v_1, \ldots, u_nv_n) = (E^{\mathcal{C}}_f(u_1, \ldots, u_n) \times E^{\mathcal{D}}_f(v_1, \ldots, v_n)) \cap U^{\mathcal{E}} \quad (30)$$

$$P^{\mathcal{E}}(x) = \begin{cases} (P^{\mathcal{C}}(x) \times P^{\mathcal{D}}(x)) \cap U^{\mathcal{E}}\ , & x \in X^{\mathcal{C}} \cap X^{\mathcal{D}} \\ (P^{\mathcal{C}}(x) \times U^{\mathcal{D}}) \cap U^{\mathcal{E}}\ , & x \in X^{\mathcal{C}} - X^{\mathcal{D}} \\ (U^{\mathcal{C}} \times P^{\mathcal{D}}(x)) \cap U^{\mathcal{E}}\ , & x \in X^{\mathcal{D}} - X^{\mathcal{C}}\ . \end{cases} \quad (31)$$

This definition can be justified as follows. To obtain (28), we start by taking the atoms of the coarsest common refinement to be conjunctions of pairs of atoms, one from $\mathcal{C}$ and one from $\mathcal{D}$. Some of these atoms will be immediately annihilated, however, due to the constraints (21). If $x \in X^{\mathcal{C}} \cap X^{\mathcal{D}}$, then the two constraints of the form (21) involving x, one from $\mathcal{C}$ and one from $\mathcal{D}$, imply that

$$\bigcup_{u \in P^{\mathcal{C}}(x)} u = \bigcup_{v \in P^{\mathcal{D}}(x)} v\ ,$$

or equivalently that $uv = 0$ for $u \in P^{\mathcal{C}}(x)$ and $v \notin P^{\mathcal{D}}(x)$ or for $u \notin P^{\mathcal{C}}(x)$ and $v \in P^{\mathcal{D}}(x)$. These uv are annihilated, giving the definition of $U^{\mathcal{E}}$ as it appears in (28).

To justify (30), each constraint of the form (20) for $\mathcal{C}$, say

$$f(u_1, \ldots, u_n) \subseteq \bigcup_{u \in E^{\mathcal{C}}_f(\bar{u})} u\ ,$$

and the constraint

$$\bigcup_{v \in U^{\mathcal{D}}} v = 1$$

for $\mathcal{D}$ combine using (2) to give constraints

$$f(u_1 v_1, \ldots, u_n v_n) \subseteq \bigcup_{\substack{u \in E_f^{\mathcal{C}}(\bar{u}) \\ uv \in U^{\mathcal{E}}}} uv \ . \qquad (32)$$

Constraints of the form

$$f(u_1 v_1, \ldots, u_n v_n) \subseteq \bigcup_{\substack{v \in E_f^{\mathcal{D}}(\bar{v}) \\ uv \in U^{\mathcal{E}}}} uv \qquad (33)$$

are obtained in a symmetric fashion by switching $\mathcal{C}$ and $\mathcal{D}$ in the definition. Combining constraints (32) and (33) with like left hand sides, we obtain the constraint

$$f(u_1 v_1, \ldots, u_n v_n) \subseteq \bigcup_{\substack{u \in E_f^{\mathcal{C}}(\bar{u}) \\ v \in E_f^{\mathcal{D}}(\bar{v}) \\ uv \in U^{\mathcal{E}}}} uv \ .$$

The justification for (31) is similar.

5.2 Minimization

As we progress down in the search tree, repeated unifications may result in a proliferation of extraneous atoms. This can be countered by the following process, which attempts to identify redundancy by (i) deleting inaccessible atoms, and (ii) identifying equivalent atoms. The technical notions of *inaccessible* and *equivalent* are defined formally below. This construction is analogous to reducing the number of states in a deterministic or nondeterministic finite state automaton by forming the quotient modulo a suitable equivalence relation.

Definition 8 *Let $\mathcal{C}, \mathcal{D}$ be systems of set constraints in atomic form over primary variables X. We call $\mathcal{C}$ and $\mathcal{D}$* equivalent *if for any solution σ of $\mathcal{C}$ there is a solution τ of $\mathcal{D}$ such that $\sigma(x) = \tau(x)$ for all $x \in X$, and vice versa.*

Definition 9 *Let $\mathcal{C} = (U^{\mathcal{C}}, X, E^{\mathcal{C}}, P^{\mathcal{C}})$ and $\mathcal{D} = (U^{\mathcal{D}}, X, E^{\mathcal{D}}, P^{\mathcal{D}})$ be systems of set constraints in atomic form over primary variables X. A* homomorphism *$h : \mathcal{C} \to \mathcal{D}$ is a map $h : U^{\mathcal{C}} \to U^{\mathcal{D}}$ such that*

$$
\begin{aligned}
P^{\mathcal{C}}(x) &= h^{-1}(P^{\mathcal{D}}(x)) && (34)\\
h(E_f^{\mathcal{C}}(u_1,\ldots,u_n)) &= E_f^{\mathcal{D}}(h(u_1),\ldots,h(u_n)) \ . && (35)
\end{aligned}
$$

Lemma 10 *Let $\mathcal{C} = (U^{\mathcal{C}},\ X,\ E^{\mathcal{C}},\ P^{\mathcal{C}})$ and $\mathcal{D} = (U^{\mathcal{D}},\ X,\ E^{\mathcal{D}},\ P^{\mathcal{D}})$ be systems of set constraints in atomic form over primary variables X, and let $h : \mathcal{C} \to \mathcal{D}$ be a homomorphism. Then $\mathcal{C}$ and $\mathcal{D}$ are equivalent.*

PROOF. Given a run $\theta : T_\Sigma \to U^{\mathcal{C}}$ for $\mathcal{C}$, define

$$
\eta = h \circ \theta : T_\Sigma \to U^{\mathcal{D}} \ . \qquad (36)
$$

A brief argument involving (22) and (35) shows that η is a run for $\mathcal{D}$.

Conversely, given a run $\eta : T_\Sigma \to U^{\mathcal{D}}$ for $\mathcal{D}$, define a run $\theta : T_\Sigma \to U^{\mathcal{C}}$ for $\mathcal{C}$ satisfying (36) inductively: suppose $\eta(t_i) = h(\theta(t_i))$, $1 \le i \le n$. Then

$$
\begin{aligned}
\eta(f(t_1,\ldots,t_n)) &\in E_f^{\mathcal{D}}(\eta(t_1),\ldots,\eta(t_n))\\
&= E_f^{\mathcal{D}}(h(\theta(t_1)),\ldots,h(\theta(t_n)))\\
&= h(E_f^{\mathcal{C}}(\theta(t_1),\ldots,\theta(t_n))) \ ,
\end{aligned}
$$

so there exists $u \in E_f^{\mathcal{C}}(\theta(t_1),\ldots,\theta(t_n))$ such that $h(u) = \eta(f(t_1,\ldots,t_n))$. Setting $\theta(f(t_1,\ldots,t_n)) = u$, we have

$$
h(\theta(f(t_1,\ldots,t_n))) = \eta(f(t_1,\ldots,t_n)) \ .
$$

In either case, by (34) we have

$$
\begin{aligned}
\eta(t) \in P^{\mathcal{D}}(x) &\iff h(\theta(t)) \in P^{\mathcal{D}}(x)\\
&\iff \theta(t) \in P^{\mathcal{C}}(x) \ ,
\end{aligned}
$$

thus

$$
\eta^{-1}(P^{\mathcal{D}}(x)) = \theta^{-1}(P^{\mathcal{C}}(x)) \ .
$$

As argued in Theorem 6, the left and right hand sides of this equation are components (23) of set valuations satisfying $\mathcal{D}$ and $\mathcal{C}$, respectively. □

Definition 11 *Let $\mathcal{C} = (U,\ X,\ E,\ P)$ be a system in atomic form. An equivalence relation $\equiv$ on U is called a* congruence *if the following two conditions hold:*

(i) if $u \equiv v$ and $u \in P(x)$, then $v \in P(x)$;

(ii) if $u_i \equiv v_i$, $1 \leq i \leq n$, then for all $u \in E_f(u_1, \ldots, u_n)$ there exists $v \in E_f(v_1, \ldots, v_n)$ such that $v \equiv u$.

Theorem 12 *Let $\mathcal{C} = (U,\ X,\ E,\ P)$ be a system in atomic form with no inaccessible atoms in the sense of step (viii) of Algorithm 1. The congruences on $\mathcal{C}$ and homomorphic images of $\mathcal{C}$ are in one-to-one correspondence up to isomorphism.*

PROOF. We first show how to construct a quotient system modulo a congruence. This system will be a homomorphic image of $\mathcal{C}$ under the canonical map taking an atom to its congruence class.

Let $\equiv$ be a congruence on U. Associate a new variable $[u]$ with the $\equiv$-congruence class of u. Define

$$\begin{aligned} U' &= \{[u] \mid u \in U\} \\ P'(x) &= \{[u] \mid u \in P(x)\} \\ E'_f([u_1], \ldots, [u_n]) &= \{[u] \mid u \in E_f(u_1, \ldots, u_n)\}\ . \end{aligned}$$

The set $E'_f([u_1], \ldots, [u_n])$ is well-defined by Definition 11(ii). Moreover, $[u] \in P'(x)$ iff $u \in P(x)$; the left-to-right implication depends on Definition 11(i).

Now consider the system $\mathcal{C}/\equiv$ of constraints

$$\bigcup_{[u] \in U'} [u] = 1 \tag{37}$$

$$[u] \cap [v] = 0\ , \quad [u] \neq [v] \tag{38}$$

$$f([u_1], \ldots, [u_n]) \subseteq \bigcup_{[u] \in E'_f([u_1], \ldots, [u_n])} [u] \tag{39}$$

$$x = \bigcup_{[u] \in P'(x)} [u]\ , \quad x \in X \tag{40}$$

This system is in atomic form, and the canonical map $u \mapsto [u]$ is a homomorphism $\mathcal{C} \rightarrow \mathcal{C}/\equiv$.

Conversely, any homomorphism $h : \mathcal{C} \rightarrow \mathcal{D}$ induces a congruence on $\mathcal{C}$ by taking $u \equiv v$ if $h(u) = h(v)$. This operation is inverse to the quotient construction. □

It follows immediately from Lemma 10 that the system $\mathcal{C}$ and its quotient $\mathcal{C}/\equiv$ are equivalent in the sense of Definition 8.

A congruence can be defined on U by setting $u \equiv v$ if for all $f \in \Sigma$, $\bar{u}$, $\bar{v}$, and x,

$$\begin{aligned} u \in P(x) &\Longleftrightarrow v \in P(x) \\ E_f(\bar{u}, u, \bar{v}) &= E_f(\bar{u}, v, \bar{v}) \ . \end{aligned}$$

However, this congruence is by no means optimal. The following construction, analogous to the standard minimization algorithm for finite automata, may give a better solution in some cases.

The algorithm marks unordered pairs of atoms $\{u, v\}$ as inequivalent. All pairs are initially unmarked. If $u \in P(x)$ and $v \notin P(x)$ for some x, mark $\{u, v\}$. Now repeat the following two steps until there are no more marks:

(i) If $\bar{u} = u_1, \ldots, u_n$, $\bar{v} = v_1, \ldots, v_n$, and $E_f(\bar{u})$ contains an element u such that all pairs $\{u, v\}$ for $v \in E_f(\bar{v})$ are marked, then nondeterministically choose some distinct pair $\{u_i, v_i\}$, $1 \leq i \leq n$, and mark it.
(ii) If $\{u, w\}$ is marked but neither $\{u, v\}$ nor $\{v, w\}$ is marked, nondeterministically choose either $\{u, v\}$ or $\{v, w\}$ and mark it.

When done, unmarked pairs are equivalent.

Any nondeterministic execution of this process results in a congruence, and all maximally coarse congruences (resulting in minimal homomorphic images) are achieved by some execution. Moreover, if Σ contains no symbols of arity two or greater, then step (ii) can be dispensed with, since in this case step (i) is deterministic and automatically results in a transitive relation. In this case the entire process is deterministic and gives the unique maximally coarse congruence, resulting in the unique minimal homomorphic image. Very fast algorithms are available for this case [8,25].

6 An Application

In program analysis and compiler optimization, one often wishes to determine information such as whether a given variable can take on a given value at a given point in the program. Of course this is undecidable in general, but it is often possible to describe a superset of the values a variable can take on at a given point, and this approximate information may still be useful in performing optimizations.

Heintze and Jaffar [16] introduced the technique of *monadic approximation* in which variable interdependencies are ignored. See [15] for a thorough introduction to this technique and examples of its application to imperative and logic programs.

In this section we show how CLP(SC) can be used to give a concise characterization of the monadic approximation for a simple imperative programming language consisting of the following constructs:

$x := e$	simple assignment
if $x = y$ **then** p **else** q	conditional
while $x = y$ **do** p	while loop
$p; q$	sequential composition

The test $x = y$ in the conditional and while loop can be replaced by $x \neq y$ or any similar test. Programs in this language are called **while** programs.

This example is included in order to illustrate how a language like CLP(SC) might be applied in program analysis. As a general tool, the language as defined here is somewhat limited by the fact that it does not include certain constructs used in program analysis, such as projections and more general conditional expressions. Extending the language to handle these constructs constitutes a worthwhile topic for further investigation.

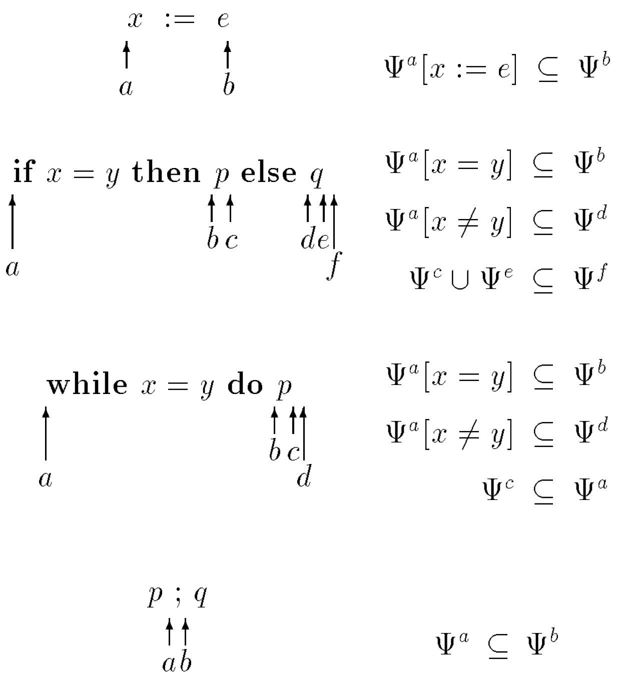
6.1 Collecting Semantics

The *collecting semantics* associates with each point in the program the set of valuations of program variables that can occur at that point during execution. Following Heintze [15], we describe here the collecting semantics for **while** programs.

Let p be a **while** program and let X be the set of program variables occurring in p. We associate with each subprogram q two points, one just before and one just after q. Each such point is labeled with a letter $a, b, c, \ldots$ We denote by Ψ^a the set of valuations $\psi : X \to \{\text{values}\}$ of program variables that ever occur at point a during execution.

Heintze [15] gives a system of set inclusions whose least solution characterizes the sets Ψ^a exactly. These are given in Figure 1. In that figure,

$$\begin{aligned}\Psi[x := e] &= \{\psi[x/\psi(e)] \mid \psi \in \Psi\}\\ \Psi[x = y] &= \{\psi \in \Psi \mid \psi(x) = \psi(y)\}\\ \Psi[x \neq y] &= \{\psi \in \Psi \mid \psi(x) \neq \psi(y)\}\end{aligned}$$


$x := e$
a
b
if $x = y$ **then** p **else** q
a
$b\,c$
$d\,e$
f
while $x = y$ **do** p
a
$b\,c$
d
$p\ ;\ q$
$a\,b$

$$\Psi^a[x := e] \subseteq \Psi^b$$

$$\Psi^a[x = y] \subseteq \Psi^b \qquad \Psi^a[x \neq y] \subseteq \Psi^d \qquad \Psi^c \cup \Psi^e \subseteq \Psi^f$$

$$\Psi^a[x = y] \subseteq \Psi^b \qquad \Psi^a[x \neq y] \subseteq \Psi^d \qquad \Psi^c \subseteq \Psi^a$$

$$\Psi^a \subseteq \Psi^b$$

Fig. 1. The collecting semantics of **while** programs

and $\psi[x/\alpha]$ denotes the valuation that agrees with ψ everywhere except possibly x, and the value of $\psi[x/\alpha]$ at x is α.

If s is the starting point of the program, then we set $\Psi^s = \{\psi_0\}$, where ψ_0 is some initial valuation.

6.2 Monadic Approximation

Heintze [15] shows that the monadic approximation to the collecting semantics can be computed as the least solution to the same set of equations as in Figure 1, except that the meaning of Ψ^a is altered to ignore dependencies among variables. Whereas Ψ^a is a collection of valuations $\psi : X \to \{\text{values}\}$, we define $\widehat{\Psi}^a$ to be a *set valuation*, *i.e.* a mapping

$$\widehat{\Psi}^a : X \to 2^{\{\text{values}\}}$$

that assigns a set of values to each program variable at point a. Under the new interpretation,

$$\widehat{\Psi}[x := e] = \widehat{\Psi}[x/\widehat{\Psi}(e)]$$

$$\widehat{\Psi}[x=y] = \begin{cases} \widehat{\Psi}[x/\widehat{\Psi}(x)\cap\widehat{\Psi}(y),\ y/\widehat{\Psi}(x)\cap\widehat{\Psi}(y)]\ , & \text{if } \widehat{\Psi}(x)\cap\widehat{\Psi}(y) \neq \varnothing \\ \lambda x.\varnothing\ , & \text{otherwise} \end{cases}$$

$$\widehat{\Psi}[x\neq y] = \begin{cases} \lambda x.\varnothing\ , & |\widehat{\Psi}(x)| = |\widehat{\Psi}(y)| = 1,\ \widehat{\Psi}(x) = \widehat{\Psi}(y) \\ \widehat{\Psi}\ , & |\widehat{\Psi}(x)| = |\widehat{\Psi}(y)| = 1,\ \widehat{\Psi}(x) \neq \widehat{\Psi}(y) \\ \widehat{\Psi}[y/\widehat{\Psi}(y)-\widehat{\Psi}(x)]\ , & |\widehat{\Psi}(x)| = 1,\ |\widehat{\Psi}(y)| > 1 \\ \widehat{\Psi}[x/\widehat{\Psi}(x)-\widehat{\Psi}(y)]\ , & |\widehat{\Psi}(x)| > 1,\ |\widehat{\Psi}(y)| = 1 \\ \widehat{\Psi}\ , & |\widehat{\Psi}(x)| > 1,\ |\widehat{\Psi}(y)| > 1\ . \end{cases}$$

Here $|A|$ denotes the cardinality of A; $\widehat{\Psi}[x/A]$ denotes the map that agrees with $\widehat{\Psi}$ everywhere except possibly x, and the value of $\widehat{\Psi}[x/A]$ at x is A; and $\widehat{\Psi}(e)$ is the set of values denoted by the expression e under the set-theoretic interpretation of the operators, where the variables occurring in e are interpreted by $\widehat{\Psi}$. The inclusions $\subseteq$ of Figure 1 are interpreted pointwise.

The definitions of $\widehat{\Psi}[x = y]$ and $\widehat{\Psi}[x \neq y]$ may seem rather complicated. Intuitively, $\widehat{\Psi}[x = y]$ is the minimal set valuation approximating the collection of valuations

$$\{\psi \mid \psi(x) = \psi(y) \text{ and } \forall z \in X\ \psi(z) \in \widehat{\Psi}(z)\}\ .$$

The set $\widehat{\Psi}[x = y]$ can be constructed as follows.

(i) Form the maximal set of valuations Ψ of which $\widehat{\Psi}$ is an approximation. This is just the direct product

$$\Psi = \prod_{z\in X} \widehat{\Psi}(z) \quad = \quad \{\psi \mid \forall z \in X\ \psi(z) \in \widehat{\Psi}(z)\}\ .$$

(ii) Intersect Ψ with the diagonal set

$$\{\psi \mid \psi(x) = \psi(y)\}$$

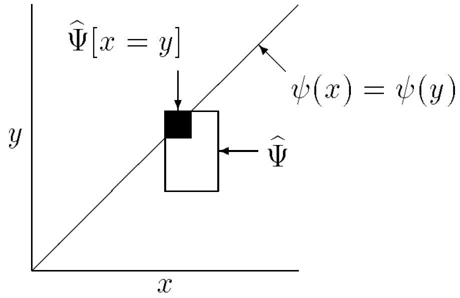
to obtain the set $\Psi[x = y]$ as defined above. (Any other reasonable test can be used here.)

(iii) Take

$$\widehat{\Psi}[x = y] = \lambda x \in X.\{\psi(x) \mid \psi \in \Psi[x = y]\}\ ,$$

the so-called *cartesian closure* of $\Psi[x = y]$ [16]. This is the smallest set valuation approximating $\Psi[x = y]$.

This construction is illustrated in the following diagram.



The construction of $\widehat{\Psi}[x \neq y]$ is similar, except that the set $\{\psi \mid \psi(x) \neq \psi(y)\}$ is used in step (ii).

One can show that $\widehat{\Psi}^a(x)$ is a superset of the set $\{\psi(x) \mid \psi \in \Psi^a\}$ of the values assigned to x under the old interpretation; *i.e.*, the monadic interpretation is a safe approximation to the collecting semantics. See Heintze [15] for further details.

Below we give a CLP(SC) program to compute the monadic approximation to the collecting semantics. In this program, the formula

$$ma(\bar{x}, \ulcorner p \urcorner, \bar{y})$$

asserts that if the set variables $\bar{x} = x_1, \ldots, x_n$ are instantiated with sets of values for the program variables (also denoted $\bar{x} = x_1, \ldots, x_n$), then after executing program p, the final sets of values assigned to the program variables under the monadic approximation are given by the values of the set variables $\bar{y} = y_1, \ldots, y_n$. The expression $\ulcorner p \urcorner$ denotes the representation of program p in some suitable encoding.

```
$ma(\bar{x}, \ulcorner x_i := e(\bar{x}) \urcorner, x_1, \ldots, x_{i-1}, e(\bar{x}), x_{i+1}, \ldots, x_n).$
$ma(\bar{x}, \ulcorner \textbf{if } b \textbf{ then } p \textbf{ else } q \urcorner, \bar{y} \cup \bar{z})$ :-
    $test(\bar{x}, \ulcorner b \urcorner, \bar{u}),\ ma(\bar{u}, \ulcorner p \urcorner, \bar{y}),$
    $test(\bar{x}, \ulcorner \neg b \urcorner, \bar{v}),\ ma(\bar{v}, \ulcorner q \urcorner, \bar{z}).$
$ma(\bar{x}, \ulcorner \textbf{while } b \textbf{ do } p \urcorner, \bar{z})$ :-
    $\bar{u} = \bar{x} \cup \bar{y},$
    $test(\bar{u}, \ulcorner b \urcorner, \bar{v}),\ ma(\bar{v}, \ulcorner p \urcorner, \bar{y}),$
    $test(\bar{u}, \ulcorner \neg b \urcorner, \bar{z}).$
$ma(\bar{x}, \ulcorner p; q \urcorner, \bar{z})$ :- $ma(\bar{x}, \ulcorner p \urcorner, \bar{y}),\ ma(\bar{y}, \ulcorner q \urcorner, \bar{z}).$

$test(\bar{x}, \ulcorner x = y \urcorner, \bar{0})$ :- $empty(x \cap y).$
$test(\bar{x}, \ulcorner x = y \urcorner, \ldots, x \cap y, \ldots, x \cap y, \ldots)$ :- $nonempty(x \cap y).$
$test(\bar{x}, \ulcorner x \neq y \urcorner, \bar{0})$ :- $x = y,\ sng(x),\ sng(y).$
```

$$\begin{array}{l}
\mathit{test}(\bar{x}, \ulcorner x \neq y \urcorner, \bar{x}) \text{ :- } \mathit{unequal}(x,y),\ \mathit{sng}(x),\ \mathit{sng}(y).\\
\mathit{test}(\bar{x}, \ulcorner x \neq y \urcorner, \ldots, x, \ldots, y - x, \ldots) \text{ :- } \mathit{sng}(x),\ \mathit{atleast2}(y).\\
\mathit{test}(\bar{x}, \ulcorner x \neq y \urcorner, \ldots, x - y, \ldots, y, \ldots) \text{ :- } \mathit{atleast2}(x),\ \mathit{sng}(y).\\
\mathit{test}(\bar{x}, \ulcorner x \neq y \urcorner, \bar{x}) \text{ :- } \mathit{atleast2}(x),\ \mathit{atleast2}(y).
\end{array}$$

If p is a program, the query

$$?\!-\ \mathit{ma}(\psi_0(x_1), \ldots, \psi_0(x_n), \ulcorner p \urcorner, \bar{y}).$$

will instantiate the variables $\bar{y}$ with the sets of possible final values of the program variables under the monadic approximation to the collecting semantics, assuming that the initial values are given by the valuation ψ_0.

Acknowledgement

I am indebted to Alex Aiken, Jeff Foster, Nevin Heintze, Joxan Jaffar, Jean-Pierre Jouannaud, Brent Knight, Tobias Nipkow, Wolfgang Thomas, Moshe Vardi, Ed Wimmers, and the anonymous referees for valuable ideas and suggestions. The support of the National Science Foundation under grant CCR-9317320 and the U.S. Army Research Office through the ACSyAM branch of the Mathematical Sciences Institute of Cornell University under contract DAAL03-91-C-0027 is gratefully acknowledged.

An abstract of this paper appeared in [21].

References

[1] A. Aiken. Personal communication, 1994.

[2] A. Aiken, D. Kozen, M. Vardi, and E. Wimmers. The complexity of set constraints. In E. Börger, Y. Gurevich, and K. Meinke, editors, *Proc. 1993 Conf. Computer Science Logic (CSL'93)*, volume 832 of *Lect. Notes in Comput. Sci.*, pages 1–17. Eur. Assoc. Comput. Sci. Logic, Springer, September 1993.

[3] A. Aiken, D. Kozen, and E. Wimmers. Decidability of systems of set constraints with negative constraints. *Infor. and Comput.*, 1995. To appear. Also Cornell University Tech. Report 93-1362, June, 1993.

[4] A. Aiken and B. Murphy. Implementing regular tree expressions. In *Proc. 1991 Conf. Functional Programming Languages and Computer Architecture*, pages 427–447, August 1991.

[5] A. Aiken and B. Murphy. Static type inference in a dynamically typed language. In *Proc. 18th Symp. Principles of Programming Languages*, pages 279–290. ACM, January 1991.

[6] A. Aiken and E. Wimmers. Solving systems of set constraints. In *Proc. 7th Symp. Logic in Computer Science*, pages 329–340. IEEE, June 1992.

[7] L. Bachmair, H. Ganzinger, and U. Waldmann. Set constraints are the monadic class. In *Proc. 8th Symp. Logic in Computer Science*, pages 75–83. IEEE, June 1993.

[8] A. Cardon and M. Crochemore. Partitioning a graph in $O(|A| \log_2 |V|)$. *Theor. Comput. Sci*, 19:85–98, 1982.

[9] W. Charatonik and L. Pacholski. Negative set constraints with equality. In *Proc. 9th Symp. Logic in Computer Science*, pages 128–136. IEEE, July 1994.

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

[11] J. Englefriet. Tree automata and tree grammars. Technical Report DAIMI FN-10, Aarhus University, April 1975.

[12] T. Frühwirth, E. Shapiro, M. Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proc. 6th Symp. Logic in Computer Science*, pages 300–309. IEEE, July 1991.

[13] R. Gilleron, S. Tison, and M. Tommasi. Solving systems of set constraints using tree automata. In *Proc. Symp. Theor. Aspects of Comput. Sci.*, volume 665, pages 505–514. Springer-Verlag Lect. Notes in Comput. Sci., February 1993.

[14] R. Gilleron, S. Tison, and M. Tommasi. Solving systems of set constraints with negated subset relationships. In *Proc. 34th Symp. Foundations of Comput. Sci.*, pages 372–380. IEEE, November 1993.

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

[16] N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Proc. 17th Symp. Principles of Programming Languages*, pages 197–209. ACM, January 1990.

[17] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. Symp. Principles of Programming Languages (POPL) 1987*, pages 111–119. ACM, January 1987.

[18] B. Jayaraman and D. A. Plaisted. Programming with equations, subsets, and relations. In E. L. Lusk and R. A. Overbeek, editors, *Proc. North Amer. Conf. Logic Programming 1989*, volume 2, pages 1051–1068. MIT Press, 1989.

[19] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Proc. 6th Symp. Principles of Programming Languages*, pages 244–256. ACM, January 1979.

[20] D. Kozen. Logical aspects of set constraints. In E. Börger, Y. Gurevich, and K. Meinke, editors, *Proc. 1993 Conf. Computer Science Logic (CSL'93)*, volume 832 of *Lect. Notes in Comput. Sci.*, pages 175–188. Eur. Assoc. Comput. Sci. Logic, Springer, September 1993.

[21] D. Kozen. Set constraints and logic programming (abstract). In J.-P. Jouannaud, editor, *Proc. First Conf. Constraints in Computational Logics (CCL'94)*, volume 845 of *Lect. Notes in Comput. Sci.*, pages 302–303. ESPRIT, Springer, September 1994.

[22] G. M. Kuper. Logic programming with sets. In *Proc. Symp. Principles of Database Systems (PODS) 1987*, pages 11–20. ACM, 1987.

[23] P. Mishra. Towards a theory of types in PROLOG. In *Proc. 1st Symp. Logic Programming*, pages 289–298. IEEE, 1984.

[24] P. Mishra and U. Reddy. Declaration-free type checking. In *Proc. 12th Symp. Principles of Programming Languages*, pages 7–21. ACM, 1985.

[25] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.

[26] J. C. Reynolds. Automatic computation of data set definitions. In *Information Processing 68*, pages 456–461. North-Holland, 1969.

[27] K. Stefánsson. Systems of set constraints with negative constraints are NEXPTIME-complete. In *Proc. 9th Symp. Logic in Computer Science*, pages 137–141. IEEE, June 1994.

[28] F. Stolzenburg. An algorithm for general set unification and its complexity. In E. Omodeo and G. Rossi, editors, *Proc. Workshop Logic Programming with Sets, in conjunction with 10th Int. Conf. Logic Programming*, pages 17–22, June 1993.

[29] F. Stolzenburg. Logic programming with sets by membership-constraints. In N. E. Fuchs and G. Gottlob, editors, *Proceedings of the 10th Logic Programming Workshop*, Universität Zürich, 1994. Institut für Informatik. Technical Report ifi 94.10.