A Decision Procedure for a Class of Set Constraints

(Extended Abstract)

Nevin Heintze
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
(nch@cs.cmu.edu)

Joxan Jaffar
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
(joxan@ibm.com)

Summary

A set constraint is of the form $exp_1 \supseteq exp_2$ where exp_1 and exp_2 are set expressions constructed using variables, function symbols, projection symbols, and the set union, intersection and complement symbols. While the satisfiability problem for such constraints is open, restricted classes have been useful in program analysis. The main result herein is a decision procedure for *definite* set constraints which are of the restricted form $a \supseteq exp$ where a contains only constants, variables and function symbols, and exp is a positive set expression (that is, it does not contain the complement symbol). A conjunction of such constraints, whenever satisfiable, has a least model and the algorithm will output an explicit representation of this model.

An additional feature of the algorithm is that it deals with another important class of set constraints. These are the *solved form* set constraints which have the form $X_1 = exp_1, \cdots, X_n = exp_n$ where the X_i are distinct variables and the exp_i are positive set expressions. A solved form constraint is always satisfiable and possesses a least and a greatest model. Our algorithm can output explicit representations of both.

1 Introduction

We consider a formalism for elementary set algebra which is useful for describing properties of programs whose underlying domain of computation is a Herbrand universe. The domain of discourse for this formalism is the powerset of this universe. Its functions include set union, intersection and complement, in addition to a family Σ of functions intended to be the set-based counterparts of the functions used in the program. More specifically, all functions f in Σ satisfy a distributive property such as $f(X \cup Y) = f(X) \cup f(Y)$. Finally, for each n-ary symbol f in Σ, we have a collection $f_{(1)}^{-1}, \cdots, f_{(n)}^{-1}$ representing the projections of f. A *set constraint* is of the form $sexp_1 \supseteq sexp_2$ where $sexp_1$ and $sexp_2$ are set expressions constructed using the above function symbols and variables. The satisfiability problem for a conjunction of set constraints is open; the rest of this section discusses important decidable sub-classes.

Set based reasoning can be expressed in the predicate calculus by regarding a unary predicate as the set of values on which it is true. There are decidability results which deal with cases where symbols are unary; see [1] for example. (These results were in fact shown for the second-order calculus.) Rabin [8], using the second-order theory of k successors and a standard interpretation, showed that the sets definable are recursive, and furthermore, he showed that there are algorithms to decide questions such as emptiness and containment. These works, however, have limited application to reasoning about programs since they are restricted to unary symbols.

Decision procedures have been obtained for set constraints involving function symbols of arbitrary arity, but at the cost of restricting the use of quantifiers, logical connectives, complement and intersection. Such procedures have been useful in an area of program analysis concerned with approximating the set of values that a variable may take at run-time. These approximations can be described using set constraints by first associating a set variable with each program variable. Set constraints can then be used to impose relationships between the set variables so that any model of these constraints defines a superset of the actual run-time values. In summary, set constraints provide a methodology of program approximation which ignores dependency between variables.

Reynolds [9] and Jones and Muchnick [5], for example, considered functional programs and used a class of set constraints whose satisfiability is guaranteed. The focus of these works was to provide an explicit representation of the model defined by the constraints. Mishra [7], in his framework for type inference in logic programs, used set constraints to model a superset of the success set of a program. Though a full decision procedure was not presented, algorithms for subcases were investigated. Heintze and Jaffar [4] used set constraints to obtain a recursive and accurate approximation of the meaning of a logic program.

In this paper, a decision procedure is presented for the class of *definite* set constraints which have the form $a \supseteq exp$ where a contains only constants, variables and function symbols, and exp is a positive set expression. Restriction to definite constraints is analogous to the restriction to definite clauses from general clauses in the sense that a definite set constraint has a least model whenever it has a model. This class of constraints is expressive enough to be useful for modeling program properties via constraints (for example, see the references cited above).

The decision procedure presented is constructive because it outputs an explicit representation of the least model of a given conjunction of definite set constraints. An additional feature of the algorithm is that it deals with another important class of set constraints. These are the *solved form* set constraints which have the form $X_1 = exp_1$, $\cdots$, $X_n = exp_n$ where $=$ denotes the obvious abbreviation of two $\supseteq$ constraints, the X_i are distinct variables and the exp_i are set expressions. A solved form constraint is always satisfiable and possesses a least and a greatest model. The algorithm can output explicit representations of both.

2 Preliminaries

Let $\mathcal{V}$ and Σ respectively denote denumerable sets of variables and functors; we assume that Σ contains at least one constant. Let $\tau(\Sigma)$ and $\tau(\Sigma \cup \mathcal{V})$ and denote the corresponding Herbrand universe (HU) and Herbrand terms. Let $var(t)$, where t is any expression, denote the set of variables appearing in t. Let $A, A_1, \cdots, A_n \subseteq \tau(\Sigma)$, $n \geq 1$. Then $f(A_1, \cdots, A_n) \stackrel{\text{def}}{=} \{f(a_1, \cdots, a_n) : a_i \in A_i\}$, and $f_{(i)}^{-1}(A) \stackrel{\text{def}}{=} \{a_i : f(a_1, \cdots, a_n) \in A\}$.

A *set expression* is defined to be either a variable or constant, the special constant $\perp$ or $\top$, or of one of the forms $f(sexp_1, \cdots, sexp_n)$, $f_{(i)}^{-1}(sexp)$, $sexp \cup sexp'$ and $sexp \cap sexp'$ where $1 \leq i \leq n$, $f \in \Sigma$ is n-ary, and $sexp$, $sexp'$, $sexp_1$, $\cdots$, $sexp_n$ are set expressions. A set expression is *atomic* if it does not contain any union, intersection, projection or complement symbols. A *set constraint* is of the form $sexp_1 \supseteq sexp_2$; if $sexp_1$ is atomic and $sexp_2$ does not contain the complement symbol, then we say that the set constraint is *definite*.

An *interpretation* is a map from $\mathcal{V} \longrightarrow 2^{HU}$. An interpretation $\mathcal{I}$, induces, in an obvious way, a mapping from set expressions $sexp$ into 2^{HU} as follows. $\mathcal{I}(\perp) = \emptyset$; $\mathcal{I}(\top) = HU$; $\mathcal{I}(c) = \{c\}$ where $c \in \Sigma$ is a constant; $\mathcal{I}(f_{(i)}^{-1}(t)) = f_{(i)}^{-1}(\mathcal{I}(t))$ and $\mathcal{I}(f(t_1, \cdots, t_n)) = f(\mathcal{I}(t_1), \cdots, \mathcal{I}(t_n)$ where $f \in \Sigma$ is n-ary; and finally, $\mathcal{I}(t \cup t') = \mathcal{I}(t) \cup \mathcal{I}(t')$ and $\mathcal{I}(t \cap t') = \mathcal{I}(t) \cap \mathcal{I}(t')$ and $\mathcal{I}(\overline{t}) = HU - \mathcal{I}(t)$. This mapping, in turn, induces a mapping from collections $\mathcal{C}$ of set constraints into *true* or *false* in the obvious way. If $\mathcal{I}$ maps $\mathcal{C}$ into *true*, we say that $\mathcal{I}$ is a *model* of $\mathcal{C}$. We write $\mathcal{I} \supseteq \mathcal{I}'$ if $\mathcal{I}(X) \supseteq \mathcal{I}'(X)$ for all $X \in \mathcal{V}$[1]. The following are some basic properties about interpretations on *ground* set expressions t and t': (a) $\mathcal{I}(t)$ denotes a unique value regardless of the choice of $\mathcal{I}$ – denote this by $[\![t]\!]$; (b) if t does not contain any union symbols, there exists a unique ground term t'' such that $\mathcal{I}(t) = \mathcal{I}(t'')$ for all $\mathcal{I}$; (c) if t and t' are different and are both terms, then $\mathcal{I}(t) \neq \mathcal{I}(t')$.

Consider next the problem of explicitly representing a given interpretation. A criteria is that the representation can be used to check whether or not an arbitrary set constraint is *true* in the interpretation. Toward this aim, define that a *term grammar* is a grammar $\mathcal{G}$ whose nonterminals are variables in $\mathcal{V}$, and each production $X \Longrightarrow t$ satisfies: t is either a constant, a special constant or of the form $f(X_1, \cdots, X_n)$, where $f \in \Sigma$ is n-ary, and the variables $X_1, \cdots, X_n$ appear in $lhs(\mathcal{G})$, the left-hand-sides of $\mathcal{G}$. Derivability $=_{\mathcal{G}}\Rightarrow$ is defined in an obvious way: $sexp =_{\mathcal{G}}\Rightarrow sexp'$ if $sexp'$ is the result of replacing *one* occurrence of a variable X in $sexp$ by t, and $X \Longrightarrow t$ is a production in $\mathcal{G}$. A term grammar $\mathcal{G}$ then defines an interpretation $\mathcal{I}_{\mathcal{G}}$ since it associates to each variable a set of ground terms as follows:

$$\mathcal{I}_{\mathcal{G}}(X) = \bigcup_{X =_{\mathcal{G}}\Rightarrow\star\, t} [\![t]\!]$$

where t is ground and $=_{\mathcal{G}}\Rightarrow\star$ is the reflexive and transitive closure of $=_{\mathcal{G}}\Rightarrow$[2].

We note that term grammar is essentially a parenthesis grammars [6]. Also a term grammar can be viewed as a description of a finite tree automaton. Hence there are algorithms for combining term grammars over the boolean and projection operations, and for testing emptyness and containment (see [10], [2]; also see [3] for a survey of relevant results). This means that, given a representation of an interpretation $\mathcal{I}$ in the form of a term grammar, standard algorithms from the literature can be used to check whether a given set constraint, $sexp_1 \supseteq sexp_2$, is *true* in $\mathcal{I}$.

The main result of this paper can now be stated as follows: Given a conjunction of definite set constraints, the algorithm presented below (a) determines if the conjunction is satisfiable, and (b) if so, outputs a term grammar which represents its least model.

3 Set Expression Grammars

We now define a class of grammars, generalizing term grammars, intended to provide a representation of an interpretation which is more explicit than the set constraint at hand, but not necessarily as explicit as a term grammar. That is, we cannot, in general, directly determine whether or not a given set constraint is *true* in the interpretation defined by a set expression grammar. (This is in constrast to a term grammar representation of an interpretation.) A key result of this paper, presented in the next section, is that a set expression grammar can be transformed into an equivalent term grammar.

A *set expression grammar* $\mathcal{G}$ is defined to be like a term grammar except that each production $X \Longrightarrow sexp$ in $\mathcal{G}$ is such that $sexp$ may contain the intersection or projection symbols[3]. As with term grammars, $\mathcal{G}$ defines an interpretation, call it $\mathcal{I}_{\mathcal{G}}$, defined as follows:

$$\mathcal{I}_{\mathcal{G}}(X) \stackrel{\text{def}}{=} \bigcup_{X \stackrel{\star}{\Longrightarrow} t} [\![t]\!]$$

where $X \in VAR$ and t is a ground set expression. Consider for example the set expression grammar $\mathcal{G}$ consisting of the two productions $X \Longrightarrow f(X) \cap f(a)$ and $X \Longrightarrow a$. Here the only ground set expressions derivable from X are $f(a) \cap f(a)$ and a, and so $\mathcal{I}_{\mathcal{G}}$ maps X to the set $\{f(a), a\}$ and all other variables to $\emptyset$.

The following two propositions pertain to a set expression $\mathcal{G}$, and are easy to verify.

[1] $\mathcal{I} \supseteq \mathcal{I}'$ implies, by monotonicity, that $\mathcal{I}(sexp) \supseteq \mathcal{I}'(sexp)$ for any $sexp$.

[2] In what follows, we may abbreviate $=_{\mathcal{G}}\Rightarrow\star$ by $\stackrel{\star}{\Longrightarrow}$ whenever there is only one grammar $\mathcal{G}$ in context.

[3] The union and complement symbols are still omitted.

Proposition 1 *Let ψ be a functor, or a projection, intersection or union symbol. Let the arity of ψ be $n \geq 1$ and let $t_1, \cdots, t_n$, $t'_1, \cdots, t'_n$ be set expressions. Then*

$\psi(t_1, \cdots, t_n) \stackrel{\star}{\Longrightarrow} \psi(t'_1, \cdots, t'_n)$ *iff* $t_1 \stackrel{\star}{\Longrightarrow} t'_1$, $\cdots$, $t_n \stackrel{\star}{\Longrightarrow} t'_n$. $\square$

Proposition 2 *Let t be a set expression. Then* $\mathcal{I}_{\mathcal{G}}(t) = \bigcup_{t \stackrel{\star}{\Longrightarrow} t'} [\![t']\!]$
where t' ranges over ground set expressions. $\square$

4 Solved Form Set Constraints

A conjunction $\mathcal{C}$ of set constraints is in *solved form* if it is of the form $X_1 = sexp_1, \cdots, X_n = sexp_n$ where the X_i are distinct variables and the $sexp_i$ are set expressions. The problem at hand is to find an explicit representation of the least and greatest models of $\mathcal{C}$, hereafter denoted $lm(\mathcal{C})$ and $gm(\mathcal{C})$ respectively.

In case $\mathcal{C}$ does not contain any intersection or projection symbols, the explicit representation of $lm(\mathcal{C})$ is easy. In case $\mathcal{C}$ does not contain any intersection symbols (but may contain projection symbols), early work by Reynolds [9], and later by Jones and Muchnick [5], gave algorithms for the explicit representation of $lm(\mathcal{C})$. The representations used in these papers are essentially the same as our term grammars. If, on the other hand, $\mathcal{C}$ does not contain any projection symbols (but may contain intersection), then a simplification algorithm contained in [4] solves the problem for $lm(\mathcal{C})$. Once again, the explicit representation used there is essentially the same as a term grammar.

The main contribution of this section is twofold: first, it deals with the case where $\mathcal{C}$ may contain both intersection and projection symbols; second, it deals not only with $lm(\mathcal{C})$, but with $gm(\mathcal{C})$ as well.

4.1 Least Model

We now present an algorithm to find an explicit representation of $lm(\mathcal{C})$. We first construct a set expression grammar which defines $lm(\mathcal{C})$. The bulk of this subsection then deals with the problem of converting this set expression grammar into an equivalent term grammar.

We begin by transforming $\mathcal{C}$ into *grammar form*; this form is defined as a conjunction, each of whose constraints is of the form $X \supseteq sexp$, where $sexp$ does not contain the union or complement symbol. This is achieved by the following simple algorithm.

> **replace** each $X = sexp$ in $\mathcal{C}$ by $X \supseteq sexp$;
> **while** $\mathcal{C}$ contains a union symbol **do**
>   **choose** an ocurrence of an expression
>   of the form $s \cup t$ in $\mathcal{C}$;
>   **replace** this occurrence by a new variable, Z;
>   **add** the constraints $Z \supseteq s$ and $Z \supseteq t$;
> **endwhile**

We can show that the conjunction $\mathcal{C}$ obtained by this algorithm has essentially the same least model as the original conjunction. The essential benefit of a conjunction being in grammar form is that we can construct a grammar representing its least model as follows.

Proposition 3 *Let $\mathcal{C}$ be a grammar form conjunction of set constraints and let $\mathcal{G}$ be the set expression grammar such that $X \Longrightarrow t \in \mathcal{G}$ just in case $X \supseteq t \in \mathcal{C}$. Then*

- $sexp \stackrel{\star}{\Longrightarrow} sexp'$ *implies* $\mathcal{C} \models sexp \supseteq sexp'$, *and*
- $\mathcal{I}_{\mathcal{G}} = lm(\mathcal{C})$. $\square$

From the correctness of the algorithm above and proposition 3, we have

Lemma 1 *Let $\mathcal{C}$ be a solved form conjunction of set constraints. Then we can construct a set expression grammar $\mathcal{G}$ such that*
$\mathcal{I}_{\mathcal{G}}(X) = (lm(\mathcal{C}))(X)$ *for all* $X \in var(\mathcal{C})$. $\square$

We now deal with the problem of converting a set expression grammar into an equivalent term grammar. We do this in three main parts. The first part reduces an arbitrary set expression grammar $\mathcal{G}$ into an equivalent one in a restricted form. This reduction will employ the following three transformations:

Transformation 1 (Unfold) *If $\mathcal{G}$ contains an occurrence of a non-variable expression t nested within the scope of a functor or a projection symbol, then replace the occurrence by a new variable Z, and add the production $Z \Longrightarrow t$.* $\square$

Transformation 2 (Substitute) *If $\mathcal{G}$ contains a production $X \Longrightarrow sexp$ such that $sexp$ contains an occurrence of a variable Y different from X and this occurrence is not nested inside a functor or a projection symbol, then*

- *let $Y \Longrightarrow t_1 \cdots, Y \Longrightarrow t_n$ be all of the productions for Y and*
- *let $sexp_1, \cdots, sexp_n$ denote the result of replacing the occurrence of Y in $sexp$ by $t_1, \cdots, t_n$ respectively*

and replace the production $X \Longrightarrow sexp$ by the productions $X \Longrightarrow sexp_1, \cdots, X \Longrightarrow sexp_n$. We say that the transformation using *Y was applied to $\mathcal{G}$.* $\square$

Transformation 3 (Cycle-Removal) *If $\mathcal{G}$ contains a production $X \Longrightarrow sexp$ such that $sexp$ has an occurrence of the variable X which is not nested inside a functor or a projection symbol, then delete this production.* $\square$

We now present an algorithm called STANDARDIZE, and we define that its operation on an input grammar $\mathcal{G}$ results in a grammar which is in *standardized* form. In what follows, the term "exhaustively-apply" means repeated application of a transformation until no change is observed.

exhaustively-apply transf^n 1;
for each variable X in $lhs(\mathcal{G})$ **do**
  exhaustively-apply transf^n 3;
  exhaustively-apply transf^n 2 using X;
endfor
delete productions containing $\perp$;
delete productions containing a variable $\notin lhs(\mathcal{G})$;

We remark that the relative order of applying these transformations, as indicated in the algorithm above, is important.

Lemma 2 *Let $\mathcal{G}$ be a set expression grammar. Then*

- *the STANDARDIZE terminates on $\mathcal{G}$, yielding a set expression grammar, say $\mathcal{G}'$;*
- *all productions in $\mathcal{G}'$ are of the form $X \Longrightarrow t_1 \cap \cdots \cap t_n$, $n \geq 1$, such that each t_i is of the form $\eta(X_1, \cdots, X_k)$ where the X_j are variables from $lhs(\mathcal{G}')$ and η is a functor or a projection symbol;*
- *for all $X \in var(\mathcal{G})$, $\mathcal{I}_{\mathcal{G}}(X) = \mathcal{I}_{\mathcal{G}'}(X)$.* $\square$

We next consider elimination of projection symbols from a set expression grammar $\mathcal{G}$. Toward this aim, we define another transformation. In condition (c) below, we assume that there exists a recursive function ORACLE$(\mathcal{G}, X)$ which returns *true* if and only if $X \stackrel{\star}{\Longrightarrow} t$ for some ground set expression t which (a) has no projection symbols, and (b) $[\![t]\!] \neq \emptyset$. We provide an implementation of ORACLE later.

Transformation 4 (Projection-Elimination) *If $\mathcal{G}$ contains a production R which has a subexpression $f_{(i)}^{-1}(Z)$ and if*

(a) $\mathcal{G}$ contains a production
  $Z \Longrightarrow f(Y_{1,1}, \cdots, Y_{1,n}) \cap \cdots \cap f(Y_{k,1}, \cdots, Y_{k,n})$ $\cap t_1 \cap \cdots \cap t_m$ where $k \geq 0$, $m \geq 0$ and the t_i are $\top$;

(b) ORACLE$(\mathcal{G}, Y_{j,i'})$, $1 \leq j \leq k$, $1 \leq i' \leq n$; and

(c) $\mathcal{G}$ contains the productions
$$Y_{j,i} \implies \alpha_j \ (1 \leq j \leq k),$$

then add to $\mathcal{G}$ the production resulting from replacing that subexpression in R by $\alpha_1 \cap \cdots \cap \alpha_k$ (or $\top$ in case $k = 0$) and any removing duplicate α_j's. ∎

Lemma 3 *Let $\mathcal{G}$ be a standardized set expression grammar. Then*

- *the exhaustive application of transformation 4 terminates, yielding $\mathcal{G}'$, and*
- *$\mathcal{I}_{\mathcal{G}} = \mathcal{I}_{\mathcal{G}'} = \mathcal{I}_{\mathcal{G}''}$ where $\mathcal{G}''$ contains just those productions in $\mathcal{G}'$ which do not contain the projection symbol.* ∎

Finally, we deal with eliminating intersection symbols. Given two term grammars, we can construct a third whose language is the intersection of the first two [10] [2]. We cannot, however, use this result to eliminate occurrences of the intersection symbol and reduce a set expression grammar into a term grammar. This is because a grammar which has intersection symbols *within* its productions cannot, in general, be broken into intersection-free parts whose interpretations combine to equal the original interpretation.

As before, we proceed by defining two new transformations on a set expression grammar $\mathcal{G}$.

Transformation 5 ($\cap$-distribution)
If (a) $\mathcal{G}$ contains a subexpression of the form $f(X_1, \cdots, X_n) \cap f(Y_1, \cdots, Y_n)$, and if (b) there are variables Z_i such that $\mathcal{I}_{\mathcal{G}}(Z_i) = \mathcal{I}_{\mathcal{G}}(X_i \cap Y_i)$ for all $1 \leq i \leq n$, then replace this subexpression by $f(Z_1, \cdots, Z_n)$. ∎

Transformation 6 ($\cap$-clash) *If $\mathcal{G}$ contains a production of the form $X \implies f(\cdots) \cap g(\cdots) \cap sexp$ where $f \neq g$, then delete this production.* ∎

It is easy to show that these productions are correct in the sense that the least model is preserved. That these productions are complete, in the sense that one of them is applicable whenever $\mathcal{G}$ still contains an intersection symbol, is in fact not true. This is because condition (b) of the $\cap$-distribution transformation may not be satisfied. To remedy this situation, new variables must be introduced. A special naming scheme is used for these new variables; each introduced variable is of the form $V_{\{X_1, \cdots, X_n\}}$ where $X_1, \cdots, X_n$ appear in $\mathcal{G}$ at the start of the algorithm. The intention here is that such a variable represents $X_1 \cap \cdots \cap X_n$. Thus a variable such as $V_{\{X,Y,Z\}}$ not only represents $X \cap Y \cap Z$, but also expressions such as $V_{\{X,Y\}} \cap Z$ and $V_{\{Y,Z\}} \cap V_{\{X,Z\}}$. The following algorithm details how this is achieved.

> **rename** each variable X in $\mathcal{G}$ into $V_{\{X\}}$;
> **repeat**
>   **exhaustively-apply** transf$^{\text{n}}$ 6 to $\mathcal{G}$;
>   **if** transf$^{\text{n}}$ 5 is applicable to $\mathcal{G}$
>   that is, $\mathcal{G}$ contains a subexpression of form
> $f(V_{S_1}, \cdots, V_{S_n}) \cap f(V_{T_1}, \cdots, V_{T_n})$
>   **then**
>   replace it by $f(V_{S_1 \cup T_1}, \cdots, V_{S_n \cup T_n})$
>   **for** each $i : 1 \leq i \leq n$ **do**
>   **if** $V_{S_i \cup T_i}$ is a new variable in $\mathcal{G}$ **then**
>   add to $\mathcal{G}$ the collection
> $\{V_{S_i \cup T_i} \implies u \cap w :$
> $V_{S_i} \implies u \in \mathcal{G}$ and $V_{T_i} \implies w \in \mathcal{G}\}$
>   **endif**
>   **endfor**
>   **endif**
> **until** neither transf$^{\text{n}}$ 5 nor 6 is applicable to $\mathcal{G}$;
> **rename** each variable $V_{\{X\}}$, $X \in \mathcal{V}$, back into X;

The following lemma states the correctness and completeness of this algorithm. The proof is mainly in establishing certain properties of the introduced variables.

Lemma 4 *Let $\mathcal{G}$ be a standardized set expression grammar containing no projection symbols. Then*

- *the algorithm above terminates when input with $\mathcal{G}$, yielding a term grammar $\mathcal{G}'$, and*
- *$\mathcal{I}_{\mathcal{G}}(X) = \mathcal{I}_{\mathcal{G}'}(X)$ for all $X \in var(\mathcal{G})$.* ∎

We now return to the problem of showing that the function ORACLE used in transformation 4 is recursive. We recall that ORACLE($\mathcal{G}, X$) is true iff

$X \stackrel{\star}{\Longrightarrow} t$ for some set expression t which (a) has no projection symbols, and (b) $[\![t]\!] \neq \emptyset$. Since t does not contain projection symbols, its derivation from X in $\mathcal{G}$ cannot involve a production containing a projection symbol. Hence ORACLE($\mathcal{G}, X$) iff ORACLE($\mathcal{G}', X$) where $\mathcal{G}'$ is the subset of productions from $\mathcal{G}$ which do not contain the projection symbol. By lemma 4 (which does not use transformation 4), we can reduce $\mathcal{G}'$ to an equivalent term grammar $\mathcal{G}''$. Since we can determine whether or not $\mathcal{I}_{\mathcal{G}''}(X) = \emptyset$, we are done.

Theorem 1 *Let $\mathcal{C}$ be a solved form conjunction of set constraints. The algorithm described in this subsection, when input with $\mathcal{C}$:*

- *terminates, yielding a term grammar $\mathcal{G}$, and*
- *$\mathcal{I}_{\mathcal{G}}(X) = (lm(\mathcal{C}))(X)$ for all $X \in var(\mathcal{C})$.* ∎

4.2 Greatest Models

We will transform the problem of explicitly representing the greatest model of a solved form conjunction $\mathcal{C}$ into the problem of explicitly representing the least model of a conjunction closely related to $\mathcal{C}$. We begin by defining the dual of a set expresson $sexp$, denoted $dual(sexp)$, as follows: if $sexp$ is:

- a variable, then $dual(sexp)$ is $sexp$;
- a constant, say c, then $dual(sexp)$ is the expression Ω_c;
- $\perp$ or $\top$, then $dual(sexp)$ is $\top$ or $\perp$ respectively;
- $f(t_1, \cdots, t_n)$, then $dual(sexp)$ is
  $\Omega_f \cup f(dual(t_1), \top, \cdots, \top) \cup$
  $f(\top, dual(t_2), \cdots, \top) \cup \cdots \cup$
  $f(\top, \top, \cdots, dual(t_n))$;
- $f_{(i)}^{-1}(t)$, then $dual(sexp)$ is $f_{(i)}^{-1}(dual(t))$;
- $s \cup t$, then $dual(sexp)$ is $dual(s) \cap dual(t)$;
- $s \cap t$, then $dual(sexp)$ is $dual(s) \cup dual(t)$

where Ω_f (f is a functor or constant) denotes the expression $\bigcup g(\top, \cdots, \top)$, where g ranges over all functors and constants different from f[4]. Now let $\mathcal{I}$ be an interpretation, $sexp$ be a set expression and $\mathcal{C}$ be a solved form conjunction of set constraints. Define $dual(\mathcal{C}) = \{X = dual(sexp) : X = sexp$ is an equation in $\mathcal{C}\}$ and $\overline{\mathcal{I}}(X) = HU - \mathcal{I}(X)$. Then

Lemma 5 *(a) $\overline{\mathcal{I}}(dual(sexp)) = HU - \mathcal{I}(sexp)$, and (b) $\mathcal{I} \models \mathcal{C}$ iff $\overline{\mathcal{I}} \models dual(\mathcal{C})$.* ∎

Combining lemma 5 and the following algorithm, we obtain Theorem 2 below.

construct $dual(\mathcal{C}) =$
  $\{X = dual(sexp) : X = sexp \in \mathcal{C}\}$;
run the algorithm in theorem 1 on $dual(\mathcal{C})$
  to obtain the term grammar $\mathcal{G}_1$;
construct $\mathcal{C}_1 = \{X = \bigcup s : X \Longrightarrow s \in \mathcal{G}_1\}$;
construct $dual(\mathcal{C}_1) =$
  $\{X = dual(sexp) : X = sexp \in \mathcal{C}_1\}$;
run the algorithm in theorem 1 on $dual(\mathcal{C}_1)$
  to obtain $\mathcal{G}$;

Theorem 2 *Let $\mathcal{C}$ be a solved form conjunction of set constraints. Then*

- *the algorithm, when input with $\mathcal{C}$, terminates, yielding a term grammar $\mathcal{G}$;*
- *$\mathcal{I}_{\mathcal{G}}(X) = (gm(\mathcal{C}))(X)$ for all $X \in var(\mathcal{C})$.* ∎

5 Definite Set Constraints

We now apply the algorithms developed in section 4 to the the satisfiability problem for set constraints. First we state

Proposition 4 *A conjunction $\mathcal{C}$ of definite set constraints has a least model whenever it has a model.* ∎

The following transformation serves to simplify the left-hand-sides of expressions in a conjunction of definite set constraints.

[4] In this section we assume for convenience that Σ is finite. This restriction can be easily lifted.

Transformation 7 (Flattening)
Let $\mathcal{C}$ be a conjunction of definite constraints, and let $f(t_1, \cdots, t_n) \supseteq sexp$, $n \geq 1$, be a constraint in $\mathcal{C}$. Then replace this constraint with the constraints $t_1 \supseteq f^{-1}_{(1)}(sexp)$, $\cdots$, $t_n \supseteq f^{-1}_{(n)}(sexp)$. $\square$

This transformation does not preserve equivalence because the transformed conjunction admits, in general, more models than the original conjunction possesses. However, we do have

Lemma 6 *Let $\mathcal{C}$ be a conjunction of definite set constraints, and let $\mathcal{C}'$ be the result of one or more applications of the flattening transformation on $\mathcal{C}$. Then*

- *if $\mathcal{C}'$ is unsatisfiable, then $\mathcal{C}$ is unsatisfiable;*
- *if $lm(\mathcal{C}') \not\models \mathcal{C}$, then $\mathcal{C}$ is unsatisfiable;*
- *if $lm(\mathcal{C}') \models \mathcal{C}$, then $lm(\mathcal{C}') = lm(\mathcal{C})$.* $\square$

An exhaustive application of the flattening transformation on a conjunction $\mathcal{C}$ can be easily shown to terminate. Constraints in the resulting conjunction $\mathcal{C}'$ are of the form (a) $X \supseteq sexp$ or (b) $c \supseteq sexp$ where X is variable, c is a constant and $sexp$ is a set expression. Let $active(\mathcal{C}')$ denote the subset of $\mathcal{C}'$ corresponding to (a). The following is the key property we require of constraints like $\mathcal{C}'$:

Lemma 7 *Let $\mathcal{C}'$ be a conjunction of definite set constraints whose left-hand-sides are either variables or constants. Then*

- *$lm(active(\mathcal{C}'))$ exists;*
- *if $lm(active(\mathcal{C}')) \not\models \mathcal{C}'$, then $\mathcal{C}'$ is unsatisfiable;*
- *if $lm(active(\mathcal{C}')) \models \mathcal{C}'$, then $lm(active(\mathcal{C}')) = lm(\mathcal{C}')$.* $\square$

Recall that given a term grammar representation of an interpretation $\mathcal{I}$, we can determine whether or not an arbitrary set constraint is *true* in $\mathcal{I}$. Combining this with Theorem 1 and the above lemmas, we have that the following algorithm decides the satisfiability of an input conjunction $\mathcal{C}$ of definite set constraints.

> **exhaustively-apply** the flattening transf$^{\text{n}}$ to $\mathcal{C}$
>   to obtain $\mathcal{C}'$;
> **run** the algorithm in theorem 1 on $active(\mathcal{C}')$
>   to obtain grammar $\mathcal{G}$;
> **if** $\mathcal{I}_{\mathcal{G}} \models \mathcal{C}$ **then** output $\mathcal{G}$;
> **else** output "unsatisfiable";

Theorem 3 *Let $\mathcal{C}$ be a conjunction of definite set constraints. Then*

- *the above algorithm terminates;*
- *if $\mathcal{C}$ is satisfiable, the algorithm outputs a term grammar $\mathcal{G}$ such that $\mathcal{I}_{\mathcal{G}}(X) = (lm(\mathcal{C}))(X)$ for all $X \in var(\mathcal{C})$;*
- *if $\mathcal{C}$ is unsatisfiable, the algorithm outputs "unsatisfiable".* $\square$

References

[1] W. Ackermann, "Solvable Cases of the Decision Problem", Studies in Logic and Foundations of Mathematics, North-Holland, 1962.

[2] J. Doner, "Tree Acceptors and Some of their Applications", *Journal of Computer and System Sciences* 4, pp 406 - 451, 1970.

[3] F. Gècseg and M. Steinby, "Tree Automata", Akadèmiai Kiadò, Budapest, 1984.

[4] N. Heintze and J. Jaffar, "A Finite Presentation Theorem for Approximating Logic Programs", draft manuscript (abstract appears in *Proc. ACM-POPL-90*), November 1989, 56 pp.

[5] N.D. Jones and S.S. Muchnick, "Flow Analysis and Optimization of LISP-like Structures", *Proceedings 6th ACM Symposium on Principles of Programming Languages*, pp 244 - 256, 1979.

[6] R. McNaughton, "Parenthesis Grammars", *Journal of the ACM* 14(3), pp 490 - 500, 1967.

[7] P. Mishra, "Toward a Theory of Types in PROLOG", *Proceedings 1st IEEE Symposium on Logic Programming*, Atlantic City, pp 289 - 298, 1984.

[8] M.O. Rabin, "Decidability of Second-order Theories and Automata on Infinite Trees", *Transactions of the American Math. Society* 141, pp 1 - 35, 1969.

[9] J.C. Reynolds, "Automatic Computation of Data Set Definitions", *Information Processing 68*, North-Holland, pp 456 - 461, 1969.

[10] J.W. Thatcher and J.B. Wright, "Generalized Finite Automata with an Application to a Decision Problem of Second-order Logic", *Mathematical Systems Theory* 2, pp 57 - 82, 1968.

Appendix: Some Program Approximation Examples

One application of set constraints is in the approximate modeling of programs. The idea here is to extract from a program a collection of constraints which capture an approximation of the desired program property. In this section we will use this approach to obtain a conservative approximation of the values that program variables may assume during execution. We give only an informal outline of how the appropriate set constraints may be obtained from a program; a more detailed formulation of the extraction process is currently being investigated.

Example 1

Consider a simple imperative programming language with basic facilities for data structure creation (e.g. *cons* and *nil* for list creation) and data destructuring/projection (e.g. *car* and *cdr* for list destructuring). We also assume a simple template matching facility. This takes the form of a condition, $x == template$, which is *true* if the value of x is an instance of *template*. Now, let P be the program

0. $x := cons(1, cons(2, cons(3, nil)));$
1. **while** $(x == cons(z_1, cons(z_2, z_3)))$ {
2. $\quad y := car(x);$
3. $\quad x := cdr(x);$
4. }

The focus of this example is the while loop at lines 1-4 which, for concreteness, we have preceeded by a very simple initialization to the variable x. The general idea, however, is that the loop will be in the context of some larger program. Then, the possible values for x on entry to the loop will be determined, not by a simple initialization, but by set constraints corresponding to the rest of the program[5].

To model the above program P, we first introducing set variables, corresponding to the program variables, whose intent is to capture the possible values that program variables may take. Constraints corresponding to P may then be written:

$$\begin{array}{llcl}
(A) & X_{\mathrm{entry}} & = & cons(1, cons(2, cons(3, nil))) \\
(B) & X_{\mathrm{while}} & \supseteq & X_{\mathrm{entry}} \\
(C) & X_{\mathrm{body}} & = & X_{\mathrm{while}} \cap cons(\top, cons(\top, \top)) \\
(D) & Y & = & cons^{-1}_{(1)}(X_{\mathrm{body}}) \\
(E) & X_{\mathrm{while}} & \supseteq & cons^{-1}_{(2)}(X_{\mathrm{body}}) \\
(F) & X_{\mathrm{exit}} & = & X_{\mathrm{while}} \cap \overline{cons(\top, cons(\top, \top))}
\end{array}$$

where $cons^{-1}_{(1)}$ and $cons^{-1}_{(2)}$ respectively model *car* and *cdr*. Constraint (A) corresponds to line 0; constraints (B) and (C) correspond to line 1; constraints (D) and (E) correspond to lines 2 and 3 respectively; and constraint (F) corresponds to lines 1-4. The four variables X_{entry}, X_{while}, X_{body} and X_{exit} are intended to capture, respectively, the values of the variable X (a) before the execution of the while statement, (b) at the head of the while statement (c) at the start of the body of the while statement and (d) after execution of the while statement. (A single set variable could be used to capture all of these uses of X, resulting in a simpler, but less accurate, analysis.) The set of values of y is modeled by the set variable Y. The least model of the above constraints describes a conservative approximations to program variable values and is given by

[5] In particular, the set of possible values for x may be infinite.

$$
\begin{aligned}
X_{\text{entry}} &= \{cons(1, cons(2, cons(3, nil)))\} \\
X_{\text{while}} &= \{cons(1, cons(2, cons(3, nil))), \\
&\quad\; cons(2, cons(3, nil)), \\
&\quad\; cons(3, nil)\} \\
X_{\text{body}} &= \{cons(1, cons(2, cons(3, nil))), \\
&\quad\; cons(2, cons(3, nil))\} \\
Y &= \{1, 2\} \\
X_{\text{exit}} &= \{cons(3, nil)\}.
\end{aligned}
$$

Example 2

An important aspect of the set constraint approach to program approximation which is not illustrated example 1, is the ability to uniformly deal with infinite sets of values. We conclude with a simple example of an analysis to approximate the success set of a logic program. Let m and n be some given positive integers. Let $P_{m,n}$ be the logic program

$$
\begin{aligned}
&p(a). \\
&p(s^m(X)) \leftarrow p(X). \\
&p(Y) \leftarrow p(s^n(Y)).
\end{aligned}
$$

Then, on letting Υ denote an approximation to the set of terms t such that $p(t)$ is true, we obtain an approximation to the meaning of p by noting that either (a) t is a; (b) t is $s^m(t')$ such that t' is in Υ; or (c) t is such that $s^n(t)$ is in Υ. Writing this as a set constraint, we obtain:

$$\Upsilon = a \cup s^m(\Upsilon) \cup s^{-n}(\Upsilon).$$

The least model of this constraint gives $\Upsilon = \{s^{\alpha k}(a)\}$ where k is the greatest common multiple of m and n, and α ranges over the natural numbers. Hence, for this example, we have in fact obtained an exact description of the success set of $P_{m,n}$.