# Set Constraints in Program Analysis

NEVIN HEINTZE

*School of Computer Science*
*Carnegie Mellon University*[*]

Set constraints provide a mechanism for analyzing programs that is based on the intuition of treating program variables as sets of "values". This paper presents an overview of how set constraints may be used in program analysis. We present the basic ideas behind the use of set constraints to approximate program behaviour, and outline algorithms for "solving" these constraints. This is illustrated by examples from logic, imperative and functional programming. We also describe advantages and disadvantages of the set constraint approach.

## 1 Introduction

Reasoning about a program by treating program variables as denoting sets of "values" leads to a simple, accurate and intuitively appealing notion of program approximation. This "set based" approach to program analysis consists of two phases. First, *set constraints* are constructed to approximate the behaviour of the program. These constraints are *solved* to find their minimum solution (this solving process is the main computational part of the analysis). In the following examples, we shall use 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). Consider the statement X := cons(Y, X). To model this statement, set variables are introduced to collect the values of the variables X and Y just before and just after the statement (we suppose that these are the only program variables of interest). Let $\mathcal{X}_1$ and $\mathcal{Y}_1$ be the set variables to collect the values of X and Y just before execution of the statement, and let $\mathcal{X}_2$ and $\mathcal{Y}_2$ be the set variables for just after statement execution. Now, the values for X just after execution of the statement include all values $cons(v_Y, v_X)$ such that $v_X \in \mathcal{X}_1$ and $v_Y \in \mathcal{Y}_1$, and so we write $\mathcal{X}_2 \supseteq \{cons(v_Y, v_X) : v_X \in \mathcal{X}_1, v_Y \in \mathcal{Y}_1\}$, which is abbreviated by $\mathcal{X}_2 \supseteq cons(\mathcal{Y}_1, \mathcal{X}_1)$. In contrast, the values for Y just after execution of the statement are exactly those before execution, and so we write the constraint $\mathcal{Y}_2 \supseteq \mathcal{Y}_1$. Hence, from the above program statement, we construct two set constraints: $\mathcal{X}_2 \supseteq cons(\mathcal{Y}_1, \mathcal{X}_1)$ and $\mathcal{Y}_2 \supseteq \mathcal{Y}_1$. Note that for this example, we could have replaced $\supseteq$ by $=$ and written the equations $\mathcal{X}_2 = cons(\mathcal{Y}_1, \mathcal{X}_1)$ and $\mathcal{Y}_2 = \mathcal{Y}_1$. However, for a number of reasons, it is somewhat more convenient to use inequalities rather than equalities[1].

Similarly, for the statement X := cdr(X) we construct the two constraints $\mathcal{X}_2 \supseteq cdr(\mathcal{X}_1)$ and $\mathcal{Y}_2 \supseteq \mathcal{Y}_1$, where $cdr(\mathcal{X}_1)$ abbreviates $\{v_2 : cons(v_1, v_2) \in \mathcal{X}_1\}$, and $\mathcal{X}_1, \mathcal{Y}_1, \mathcal{X}_2, \mathcal{Y}_2$ are as before.

[*]5000 Forbes Ave., Pittsburgh, PA 15213. *Email:* nch@cs.cmu.edu.

[1]In particular, the construction of set constraints is simpler in the presence of statements that change the flow of control.

In general, the use of sets to reason about a program leads to an approximation of the program's actual behaviour. This is because the use of sets ignores dependencies between variable values. For example, consider the following program

```
X := car(W);
Y := cdr(W);
W := cons(X, Y);
```

Let $\mathcal{W}_i, \mathcal{X}_i$ and $\mathcal{Y}_i$, $i = 1..4$, be the set variables introduced to collect the values of W, X and Y just before the first statement, just before the second statement, just before the third statement, and just after the third statement respectively. Constructing constraints as before yields:

$$\begin{array}{lll} \mathcal{W}_2 \supseteq \mathcal{W}_1 & \mathcal{W}_3 \supseteq \mathcal{W}_2 & \mathcal{W}_4 \supseteq cons(\mathcal{X}_3, \mathcal{Y}_3) \\ \mathcal{X}_2 \supseteq car(\mathcal{W}_1) & \mathcal{X}_3 \supseteq \mathcal{X}_2 & \mathcal{X}_4 \supseteq \mathcal{X}_3 \\ \mathcal{Y}_2 \supseteq \mathcal{Y}_1 & \mathcal{Y}_3 \supseteq cdr(\mathcal{W}_2) & \mathcal{Y}_4 \supseteq \mathcal{Y}_3 \end{array}$$

Now, suppose that at the start of the program, the variable W is either the list $[1,2]$ or $[3,4]$. Then the set for $\mathcal{X}_2$ (and $\mathcal{X}_3$) is $\{1,3\}$, and the set for $\mathcal{Y}_3$ is $\{[2],[4]\}$. Hence, the set for $\mathcal{W}_4$ is $\{[1,2],[3,4],[1,4],[3,2]\}$. In contrast, the only possible values for W after execution of the third statement are $[1,2]$ and $[3,4]$.

The key property of the constraints constructed from a program is that any solution of the constraints conservatively approximates the operational semantics of the program. This means that to obtain a safe approximation of the program, it is sufficient to construct a solution to the constraints. The constraint solving process will typically compute the minimum solution to the constraints since this is the most accurate approximation (described by the constraints).

To summarize, set constraints can be constructed to approximate the execution of a program by first introducing set variables to capture the values of the program variables at each program point, and then writing constraints between these set variables to approximate the relationships between these variables that are inherent in the program. In effect, the construction of constraints reduces the problem of analyzing the program to the problem of reasoning about set constraints.

One of the major differences between the set constraint approach to program analysis, and more traditional abstract interpretation approaches [4], is that the set constraint approach does not employ an abstract domain to approximate the underlying domain of computation. Instead, approximation is introduced in the treatment of variables – in essence, all inter-variable dependencies are ignored through the treatment of program variables as sets of values.

The focus of this paper is on the use of set constraints in program analysis for a variety of programming languages. In particular, we shall ignore issues such as correctness (which are dealt with at length elsewhere [7]). Similarly, we shall compromise accuracy for constraint simplicity. This is in contrast to some other presentations which have sought a natural, canonical definition of set based program approximation, and then have used set constraints as a tool to compute this approximation [9, 7, 8].

We begin in Section 2 by describing the basic form of the set constraints we shall use. Next, in Section 3 we show how set constraints can be used to approximate the run-time behaviour of a variety of programming languages. For each language considered, we shall introduce new set operations to model the semantic operations of the language. In Section 4 we sketch the core part of the set constraint algorithm. Section 5 describes some extensions to the basic approach to provide

mode and structure sharing information. Section 6 gives a critique of the set constraint approach, and section 7 describes some future directions.

## 2 Set Constraints

A *set expression* is either a set variable (denoted $\mathcal{V}$, $\mathcal{W}$, $\mathcal{X}$, $\mathcal{Y}$, etc.), or of one of the forms $f(se_1, \ldots, se_n)$, $se_1 \cup se_2$ or $op(se_1, \ldots, se_n)$ where $f$ is a function symbol (data-constructor), the $se_i$ are set expressions, and $op$ is a set operator. Example set operators include intersection (denoted $\cap$) and projections (denoted $f^{-1}_{(i)}$ where $f$ is an $n$-ary function symbol and $1 \leq i \leq n$). As an example of projection, the operator $cons^{-1}_{(1)}$ denotes the first projection with respect to the constructor *cons*, and is the "set" counterpart of *car*. It is also convenient to include $\top$ and $\bot$ in the definition of set expressions to respectively denote the set of all terms and the empty set. A *set constraint* is of the form $\mathcal{X} \supseteq se$ where $\mathcal{X}$ is a set variable and $se$ is a set expression.

In essence, set constraints are a calculus for expressing relationships between sets of values. This calculus is parameterized by the underlying domain of values and the set operators employed in the calculus. For the purposes of this section, the underlying values shall be ground terms constructed from the data constructors at hand (although we shall discuss other possibilities in section 5), and the set operators shall consist of intersection and projections.

A *solution* to a collection of set constraints is an assignment of sets to set variables that satisfies each constraint. Specifically, let $\mathcal{I}$ be a mapping from set variables into sets of values (ground terms). Such a mapping can be extended to map from set expressions into sets of values:

- $\mathcal{I}(f(se_1, \ldots, se_n)) = \{f(v_1, \ldots, v_n) : v_i \in \mathcal{I}(se_i)\}$;
- $\mathcal{I}(se_1 \cup se_2) = \mathcal{I}(se_1) \cup \mathcal{I}(se_2)$;
- $\mathcal{I}(se_1 \cap se_2) = \mathcal{I}(se_1) \cap \mathcal{I}(se_2)$;
- $\mathcal{I}(f^{-1}_{(i)}(se)) = \{v_i : f(v_1, \ldots, v_n) \in \mathcal{I}(se)\}$;
- $\mathcal{I}(\top) =$ all values, and $\mathcal{I}(\bot) = \{\}$

$\mathcal{I}$ is a *solution* of a collection of constraints $\mathcal{C}$ if $\mathcal{I}(\mathcal{X}) \supseteq \mathcal{I}(se)$ for each constraint $\mathcal{X} \supseteq se$ in $\mathcal{C}$.

For example, let $\mathcal{C}$ denote the single constraint $\mathcal{X} \supseteq c \cup f(f(\mathcal{X}))$, where $c$ is a constant and $f$ is a unary symbol. $\mathcal{C}$ has many models, including the mapping that maps all set variables into the set $\{c, f(c), f(f(c)), \ldots\}$. Another solution of $\mathcal{C}$ is the mapping $\mathcal{I}$ defined by

$$\mathcal{I}(\mathcal{Y}) = \begin{cases} \{c, f^2(c), f^4(c), \ldots\} & \text{if } \mathcal{Y} \text{ is } \mathcal{X} \\ \{\} & \text{if } \mathcal{Y} \text{ is different from } \mathcal{X} \end{cases}$$

where $f^n$ abbreviates $n$ applications of $f$. This solution is smaller than the first, and is in fact the smallest solution of $\mathcal{C}$. In general, a collection of set constraints always has a unique smallest solution[2], and it is this smallest solution that shall be computed by the constraint solv-

[2]Strictly speaking, a smallest solution exists whenever the set operators are monotonic, and this shall be the case for all of the set operators considered in this paper. We also note that for more general definitions of set constraints (for example involving constraints of the form $se_1 \supseteq se_2$), unique minimum solutions may not exist even if the operators are monotonic (see [10]).

ing algorithms. As another example, the smallest solution of the following constraint collection maps $\mathcal{X}$ into $\{a, f^3(a), f^6(a), \ldots\}$, maps $\mathcal{Y}$ into $\{a, f^2(a), f^4(a), \ldots\}$ and maps $\mathcal{Z}$ into $\{f^5(a), f^{11}(a), f^{17}(a), \ldots\}$.

$$\begin{array}{lcl} \mathcal{X} & \supseteq & a \cup f^3(\mathcal{X}) \\ \mathcal{Y} & \supseteq & a \cup f^2(\mathcal{Y}) \\ \mathcal{Z} & \supseteq & f^{-1}_{(1)}(\mathcal{X} \cap \mathcal{Y}) \end{array}$$

# 3 Construction of Set Constraints

## Imperative Programs

The example imperative programs considered in the introduction did not illustrate how conditional statements and recursion are handled, and these are probably the most interesting aspects of the analysis. In particular, recursion introduces the possibility of infinite sets of values. Consider the following program

```
        X := cons(a, cons(b, cons(c, cons(d, nil))));
        Y := nil;
(A)
        while(car(X) ≠ c) do
(B)
            Y := cons(car(X), Y);
            X := cdr(X);
(C)
(D)
```

where a, b, c and d are constants. After execution of this program, X is $cons(c, cons(d, nil))$ and Y is $cons(b, cons(a, nil))$; in other words the program reverses the initial segment of X up until the first occurrence of $c$. The markers Ⓐ, Ⓑ, Ⓒ and Ⓓ indicate points in the program (note that Ⓓ indicates the point at the end of the program). Corresponding to this program, we can construct the following constraints.

$$\begin{array}{lcl} \mathcal{X}^A & \supseteq & cons(a, cons(b, cons(c, cons(d, nil)))) \\ \mathcal{Y}^A & \supseteq & nil \\ \mathcal{X}^B & \supseteq & \mathcal{X}^A \cap \overline{cons(c, \top)} \\ \mathcal{Y}^B & \supseteq & \mathcal{Y}^A \\ \mathcal{X}^B & \supseteq & \mathcal{X}^C \cap \overline{cons(c, \top)} \\ \mathcal{Y}^B & \supseteq & \mathcal{Y}^C \end{array} \qquad \begin{array}{lcl} \mathcal{X}^C & \supseteq & cdr(\mathcal{X}^B) \\ \mathcal{Y}^C & \supseteq & cons(car(\mathcal{X}^B), \mathcal{Y}^B) \\ \mathcal{X}^D & \supseteq & \mathcal{X}^A \cap cons(c, \top) \\ \mathcal{Y}^D & \supseteq & \mathcal{Y}^A \\ \mathcal{X}^D & \supseteq & \mathcal{X}^C \cap cons(c, \top) \\ \mathcal{Y}^D & \supseteq & \mathcal{Y}^C \end{array}$$

The set expression $\overline{cons(c, \top)}$ (the complement of the set denoted by $cons(c, \top)$) is the set of all values $v$ such that car$(v)$ differs from $c$. In general, it is useful to introduce a restricted form of complementation in the constraints used to analyze imperative programs. However, these uses are always sufficiently limited that the constraints obtained are still "monotonic". The minimum solution of the above constraints is given by the following mapping:

$$
\begin{array}{lcl}
\mathcal{X}^A & \mapsto & \{cons(a, cons(b, cons(c, cons(d, nil))))\} \\
\mathcal{Y}^A & \mapsto & \{nil\} \\
\mathcal{X}^B & \mapsto & \{cons(a, cons(b, cons(c, cons(d, nil)))),\ \ cons(b, cons(c, cons(d, nil)))\} \\
\mathcal{Y}^B & \mapsto & list_{a,b} \\
\mathcal{X}^C & \mapsto & \{cons(b, cons(c, cons(d, nil))),\ \ cons(c, cons(d, nil))\} \\
\mathcal{Y}^C & \mapsto & \textit{non-nil-list}_{a,b} \\
\mathcal{X}^D & \mapsto & \{cons(c, cons(d, nil))\} \\
\mathcal{Y}^D & \mapsto & list_{a,b}
\end{array}
$$

where $list_{a,b}$ denotes the set of all lists constructed from $a$ and $b$, and *non-nil-list*$_{a,b}$ denotes the set of all non-empty lists constructed from $a$ and $b$.

## Logic Programs

The construction of set constraints for logic programs is similar to that for imperative programs. However, for logic programs, there is a choice for the underlying operational semantics used in the analysis. We begin by illustrating the construction of constraints corresponding to a bottom-up execution. Again we introduce a set variable for each program variable. We also introduce set variables $Ret_p$, for each predicate $p$, to collect the set of "return" values for that predicate. Consider the following logic program and constraints constructed to model the bottom-up semantics of the program.

```
p(X) :- q(X), r(X).
q(a).
q(b).
r(b).
r(c).
```

$$
\begin{array}{l}
Ret_p \supseteq p(\mathcal{X}) \\
\quad \mathcal{X} \supseteq q^{-1}_{(1)}(Ret_q) \cap r^{-1}_{(1)}(Ret_r) \\
Ret_q \supseteq q(a) \cup q(b) \\
Ret_r \supseteq r(b) \cup r(c)
\end{array}
$$

The minimum solution of these constraints maps $Ret_p$ into $\{p(b)\}$, maps $\mathcal{X}$ into $\{b\}$, maps $Ret_q$ into $\{q(a), q(b)\}$, and maps $Ret_r$ into $\{r(b), r(c)\}$. Now, consider constructing constraints corresponding to a top-down left-to-right execution of the program starting from the goal `?- p(`$t$`)` where $t$ is either $a$, $b$, $c$ or $d$. The main change here is the introduction of set variables $Call_p$, for each predicate $p$, to collect the set of "calls" to that predicate. The program points Ⓐ, Ⓑ and Ⓒ respectively denote the points just before execution of $q(X)$, just before execution of $r(X)$ and just after execution of $r(X)$.

```
p(X) :- Ⓐ, q(X), Ⓑ, r(X), Ⓒ.
q(a).
q(b).
r(b).
r(c).
```

$$
\begin{array}{l}
\textit{Call}_p \supseteq p(a \cup b \cup c \cup d) \\
\textit{Ret}_p \supseteq p(\mathcal{X}^C) \\
\mathcal{X}^A \supseteq p^{-1}_{(1)}(\textit{Call}_p) \\
\mathcal{X}^B \supseteq p^{-1}_{(1)}(\textit{Call}_p) \cap q^{-1}_{(1)}(\textit{Ret}_q) \\
\mathcal{X}^C \supseteq p^{-1}_{(1)}(\textit{Call}_p) \cap q^{-1}_{(1)}(\textit{Ret}_q) \cap r^{-1}_{(1)}(\textit{Ret}_r) \\
\textit{Call}_q \supseteq q(\mathcal{X}^A) \\
\textit{Ret}_q \supseteq (q(a) \cup q(b)) \cap \textit{Call}_q \\
\textit{Call}_r \supseteq r(\mathcal{X}^B) \\
\textit{Ret}_r \supseteq (r(b) \cup r(c)) \cap \textit{Call}_r
\end{array}
$$

The minimum solution of these constraints maps $Call_p$ into $\{p(a), p(b), p(c), p(d)\}$, $Ret_p$ into

$\{p(b)\}$, $\mathcal{X}^A$ into $\{a,b,c,d\}$, $\mathcal{X}^B$ into $\{a,b\}$, $\mathcal{X}^C$ into $\{b\}$, $Call_q$ into $\{q(a),q(b),q(c),q(d)\}$, $Ret_q$ into $\{q(a),q(b)\}$, $Call_r$ into $\{r(a),r(b)\}$, and $Ret_r$ into $\{r(b)\}$. As a third alternative, consider constructing constraints corresponding to a top-down parallel execution of the program starting from the same goals. The program points Ⓐ, Ⓑ and Ⓒ respectively denote the points just before execution of $q(X)$, just before execution of $r(X)$ and just after execution of the entire body of the first rule.

```
p(X) :- Ⓐ, q(X), Ⓑ, r(X), Ⓒ.
q(a).
q(b).
r(b).
r(c).
```

$$
\begin{aligned}
\mathit{Call}_p &\supseteq p(a \cup b \cup c \cup d)\\
\mathit{Ret}_p &\supseteq p(\mathcal{X}^C)\\
\mathcal{X}^A &\supseteq p^{-1}_{(1)}(\mathit{Call}_p)\\
\mathcal{X}^B &\supseteq p^{-1}_{(1)}(\mathit{Call}_p)\\
\mathcal{X}^C &\supseteq p^{-1}_{(1)}(\mathit{Call}_p) \cap q^{-1}_{(1)}(\mathit{Ret}_q) \cap r^{-1}_{(1)}(\mathit{Ret}_r)\\
\mathit{Call}_q &\supseteq q(\mathcal{X}^A)\\
\mathit{Ret}_q &\supseteq (q(a) \cup q(b)) \cap \mathit{Call}_q\\
\mathit{Call}_r &\supseteq r(\mathcal{X}^B)\\
\mathit{Ret}_r &\supseteq (r(b) \cup r(c)) \cap \mathit{Call}_r
\end{aligned}
$$

The minimum solution of these constraints maps $Call_p$ into $\{p(a),p(b),p(c),p(d)\}$, $Ret_p$ into $\{p(b)\}$, $\mathcal{X}^A$ and $\mathcal{X}^B$ into $\{a,b,c,d\}$, $\mathcal{X}^C$ into $\{b\}$, $Call_q$ into $\{q(a),q(b),q(c),q(d)\}$, $Ret_q$ into $\{q(a),q(b)\}$, $Call_r$ into $\{r(a),r(b),r(c),r(d)\}$, and $Ret_r$ into $\{r(b),r(c)\}$.

Observe that in all three examples, the use of set constraints has lead to an exact analysis, and that the sets obtained were finite. Neither observation holds in general, as is illustrated by the following bottom-up analysis example:

```
p(f(X), f(Y)) :- p(X, Y).
p(a,b).
```

$$
\begin{aligned}
\mathit{Ret}_p &\supseteq p(f(\mathcal{X}), f(\mathcal{Y})) \cup p(a,b)\\
\mathcal{X} &\supseteq p^{-1}_{(1)}(\mathit{Ret}_p)\\
\mathcal{Y} &\supseteq p^{-1}_{(2)}(\mathit{Ret}_p)
\end{aligned}
$$

In the least model of the constraints, $Ret_p$ is mapped into the set $\{p(a,b)\} \cup \{p(f^i(a), f^j(b)) : i \geq 1, j \geq 1\}$, and this set contains elements such as $p(f(a), f(f(b)))$ which are not part of the program's (exact) meaning. Appendix III contains a more substantial logic program example that further illustrates these issues and also gives some insight into the kind of information obtained using set constraints.

So far, we have made no mention of variables that appear in the head of a rule and not in the body of a rule. Such variables can take on any value. Hence they are modeled using the $\top$ constant, as illustrated in the following example.

```
p(X, Y) :- q(X).
q(a).
```

$$
\begin{aligned}
\mathit{Ret}_p &\supseteq p(\mathcal{X}, \mathcal{Y})\\
\mathcal{X} &\supseteq q^{-1}_{(1)}(\mathit{Ret}_q)\\
\mathcal{Y} &\supseteq \top\\
\mathit{Ret}_q &\supseteq q(a)
\end{aligned}
$$

We conclude this discussion of the analysis of logic programs by noting that the accuracy of the information obtained using set constraints can be improved by using more complex set operators. For example, consider the following program and its (bottom-up) set constraints:

```
p(X, Y) :- q(X, Y), r(X, Y).
q(a, b).
q(b, a).
r(a, a).
```

$$
\begin{aligned}
\mathit{Ret}_p &\supseteq p(\mathcal{X},\mathcal{Y}) \\
\mathcal{X} &\supseteq q^{-1}_{(1)}(\mathit{Ret}_q) \cap r^{-1}_{(1)}(\mathit{Ret}_r) \\
\mathcal{Y} &\supseteq q^{-1}_{(2)}(\mathit{Ret}_q) \cap r^{-1}_{(2)}(\mathit{Ret}_r) \\
\mathit{Ret}_q &\supseteq q(a,b) \cup q(b,a) \\
\mathit{Ret}_r &\supseteq r(a,a)
\end{aligned}
$$

The minimum solution of these constraints maps $Ret_p$ into $p(a,a)$, $\mathcal{X}$ and $\mathcal{Y}$ into $\{a\}$, $Ret_q$ into $\{q(a,b), q(b,a)\}$, and $Ret_r$ into $\{r(a,a)\}$. Another way of constructing constraints is to introduce quantified set expressions, which have the form $\{X : \exists X_1 \ldots \exists X_m (t_1 \in se_1 \wedge \cdots \wedge t_n \in se_n)\}$ where $X, X_1, \ldots, X_m$ are program variables, $t_1, \ldots, t_n$ are atoms or terms whose variables are from $X, X_1, \ldots X_m$, and $se_1, \ldots, se_n$ are set expressions. The constraints using quantified set expressions that are constructed for the (bottom-up) analysis of the above program are:

$$
\begin{aligned}
\mathit{Ret}_p &\supseteq p(\mathcal{X},\mathcal{Y}) \\
\mathcal{X} &\supseteq \left\{X : \exists Y \left(q(X,Y) \in \mathit{Ret}_q \wedge r(X,Y) \in \mathit{Ret}_r\right)\right\} \\
\mathcal{Y} &\supseteq \left\{Y : \exists X \left(q(X,Y) \in \mathit{Ret}_q \wedge r(X,Y) \in \mathit{Ret}_r\right)\right\} \\
\mathit{Ret}_q &\supseteq q(a,b) \cup q(b,a) \\
\mathit{Ret}_r &\supseteq r(a,a)
\end{aligned}
$$

and the minimum solution of these constraints maps $Ret_p$, $\mathcal{X}$ and $\mathcal{Y}$ into the empty set, $Ret_q$ into $\{q(a,b), q(b,a)\}$, and $Ret_r$ into $\{r(a,a)\}$. The more complex constraints using quantified expressions not only provide more accurate program approximation, but they are also more faithful to the notion of set based analysis. In particular, they have closer and much simpler relationship to the underlying operational semantics (see [7, 9] for further details).

## Functional Programs

To analyze functional languages such as Standard ML [15], set constraints must be extended with a mechanism to deal with higher-order functions. In essence, this is achieved by the addition of three new components. First, the set of underlying values is enriched to include a new collection of constants to denote functions. In the following examples, we shall use function identifiers for this purpose; in more formal presentations, it is convenient to use abstractions in an appropriate lambda calculus. Second, for each function constant $f$, we introduce two set variables $Call_f$ and $Ret_f$ to capture the values on which $f$ is called, and the values that calls to $f$ return, respectively. Third, a new set operator $apply$ is introduced to model function application. The meaning of a set expression $apply(se_1, se_2)$ under a mapping $\mathcal{I}$ is defined as follows:

$$\mathcal{I}(apply(se_1, se_2)) \stackrel{\text{def}}{=} \bigcup_{f \in \mathcal{I}(se_1)} Ret_f, \quad \text{provided } \mathcal{I}(Call_f) \supseteq \mathcal{I}(se_2) \text{ for all } f \in \mathcal{I}(se_1)$$

If the side condition is not met then $\mathcal{I}(apply(se_1, se_2))$ is not defined. The notion of solution of a collection of set constraints is appropriately modified so that $\mathcal{I}$ is a solution of the constraints if it is defined on each set expression and satisfies each constraint. Note that the meaning of this expanded class of set expressions involving $apply$ is somewhat unusual, because now set expressions themselves may impose restrictions on solutions, independent of the constraints in

which they appear. Importantly, unique minimum solutions are still guaranteed to exist.

To illustrate the construction of set constraints to analyze function programs, consider the following program and its constraints. The set variable $\mathcal{E}$ is introduced to capture the set of values resulting from program evaluation. The minimum solution of the constraints maps $\mathcal{X}$, $Call_{id}$, $Ret_{id}$, and $\mathcal{E}$ into $\{c\}$.

```
let fun id X = X
in
   id c
end
```

$$\begin{aligned} \mathcal{X} &\supseteq Call_{id} \\ Ret_{id} &\supseteq \mathcal{X} \\ \mathcal{E} &\supseteq apply(id, c) \end{aligned}$$

A less trivial example is presented in Appendix IV. Again, more complex set operators can be introduced to provide more accurate modeling of certain aspects of the language (particularly case statements). See [7, 8] for further details. The complexity of solving the set constraints is $O(n^3)$ [8]. This basic formulation of constraints can also be extended to deal with arrays, continuations and exceptions [8].

# 4 Solving Set Constraints

Having outlined how set constraints may be obtained to analyze programs in a variety of languages, we now show how these constraints may be solved. The key correctness property of the constraints constructed for a program is that any solution of the constraints describes a superset of the run-time values of each program variable. Clearly the minimum solution of the constraints gives the most accurate approximation of the program's run-time behaviour, and it is this minimum solution that is computed by the constraint solving algorithms. Note that since this minimum solution will in general map set variables into infinite sets of values, the output of the algorithm will take the form of a finite representation of an infinite set. What we require of this representation is that it is explicit in the sense that simple questions such as membership, emptiness and containment can be directly determined.

To illustrate the underlying ideas of the set constraint solving process, consider the constraints $\mathcal{C}$:

$$\begin{aligned} &\mathcal{X} \supseteq a \\ &\mathcal{X} \supseteq f(f(\mathcal{X})) \\ &\mathcal{X} \supseteq f^{-1}_{(1)}(\mathcal{X}) \end{aligned}$$

The minimum solution of $\mathcal{C}$ maps $\mathcal{X}$ into $\{a, f(a), f(f(a)), \ldots\}$. Now, the fact that $\mathcal{X}$ contains $a, f(f(a)), \ldots,$ is essentially "explicit" because, by inspection of the constraints, it is clear that $\mathcal{X}$ contains all values of the form $f^{2n}(a)$. However, the fact that $\mathcal{X}$ contains $f(a)$ is "implicit" because it depends on two facts: (i) $\mathcal{X}$ contains $f^{-1}_{(1)}(v)$ for all $v$ in $\mathcal{X}$, and (ii) $\mathcal{X}$ contains $f(f(a))$.

Intuitively, a collection of set constraints is solved by a process of transformation that incrementally makes information about the minimum solution of the constraints explicit. To illustrate this, consider again the constraints $\mathcal{C}$ given above. These constraints imply that (i) $\mathcal{X}$ contains $a$, (ii) $\mathcal{X}$ contains all values of the form $f(f(v))$ such that $v$ is in $\mathcal{X}$, and (iii) $\mathcal{X}$ contains all values $v$ such that $f(v)$ is in $\mathcal{X}$. Now, (iii) implies that $\mathcal{X}$ contains all values of the form $f(v)$ such that

**input** a collection $\mathcal{C}$ of set constraints;
**repeat**
    **apply** a simplification step if the step generates new constraints
**until** no simplification step changes $\mathcal{C}$;
**output** $explicit(\mathcal{C})$;

Figure 1: **Set Constraint Simplification Algorithm**

$f(f(v))$ is in $\mathcal{X}$. Combining this with (ii) shows that $\mathcal{X}$ contains all values of the form $f(v)$ such that $v$ is in $\mathcal{X}$. Hence, we could add the constraint $\mathcal{X} \supseteq f(X)$ to $\mathcal{C}$. This step can be rephrased as follows: substitute $f(f(\mathcal{X}))$ for $\mathcal{X}$ in the right hand side of the constraint $\mathcal{X} \supseteq f^{-1}_{(1)}(\mathcal{X})$ to obtain $\mathcal{X} \supseteq f^{-1}_{(1)}(f(f(\mathcal{X})))$, cancel the $f^{-1}$ and $f$ to obtain the constraint $\mathcal{X} \supseteq f(\mathcal{X})$, and add this new constraint to $\mathcal{C}$ to obtain a collection of constraints $\mathcal{C}'$. Clearly $\mathcal{C}'$ and $\mathcal{C}$ are equivalent in the sense that they have the same solutions. Moreover, the constraints $\mathcal{C}'$ are such that the minimum solution of $\mathcal{C}'$ is now explicit in the sense that, by inspection, $\mathcal{X}$ contains all values of the form $f^n(a)$, $n \geq 0$.

This process of substitution and cancellation/simplification is the key idea behind the solving of set constraints. Before describing this in more detail, we first spell out what is meant by explicit. In essence, an expression is explicit if it just consists of set variables and function symbols. Specifically, a set expression is *atomic* if it is either a set variable, or of the form $f(ae_1, \ldots, ae_n)$ where each $ae_i$ is atomic. A constraint is in *explicit form* if it has the form $\mathcal{X} \supseteq ae$ where $ae$ is an atomic set expression that is not a set variable ($ae$ may of course *contain* set variables). A collection of constraints is in explicit form if each constraint therein is in explicit form. If $\mathcal{C}$ is a collection of constraints, then $explicit(\mathcal{C})$ denotes the explicit form constraints of $\mathcal{C}$. We note that explicit form constraints can be regarded as regular term grammars by treating set variables as non-terminals and regarding a constraint $\mathcal{X} \supseteq ae$ as a production $\mathcal{X} \Rightarrow ae$ (see Appendix I for a summary of regular term grammars and regular term languages). However, observe that unlike some works that are based on abstract interpretation, we do not start by using a fixed set of regular term languages. The fact that regular term grammars are sufficient for representing the minimum solution of a collection of set constraints is a (non-trivial) property of the set constraints[3].

To simplify the algorithm, it is convenient to preprocess constraints to put them into a simpler form. This preprocessing removes union symbols and certain kinds of nesting of set expressions. Specifically, the constraints produced by the preprocessing are of the form $\mathcal{X} \supseteq se$ where $\mathcal{X}$ is a set variable and $se$ is a set expression that is either an atomic set expression, or of the form $op(\mathcal{X}_1, \ldots, \mathcal{X}_n)$ where $\mathcal{X}_1, \ldots, \mathcal{X}_n$ are set variables. The preprocessing consists of steps such as replacing $\mathcal{X} \supseteq se_1 \cup se_2$ by $\mathcal{X} \supseteq se_1$ and $\mathcal{X} \supseteq se_2$, and replacing $\mathcal{X} \supseteq f^{-1}_{(i)}(se)$ by $\mathcal{X} \supseteq f^{-1}_{(i)}(\mathcal{Y})$ and $\mathcal{Y} \supseteq se$ provided $se$ is not a set variable and $\mathcal{Y}$ is a new set variable.

The basic structure of set constraint solving algorithms is presented in Figure 1. In essence, the algorithm consists of a family of simplification steps that are exhaustively applied to the input collection of constraints. The algorithm outputs the explicit form constraints in the resulting collection. Correctness of the algorithm has three aspects: (i) each transformation steps must be *sound* in the sense that it does not change the minimum solution of the constraints; (ii) the

[3] In particular, there are solutions of set constraints that are not representable by regular term grammars.

collection of transformations must be *complete* in the sense that when no transformations steps can be applied, all information about the minimum solution of the constraints appears in the explicit form constraints of the collection (and so on termination of the main loop of the algorithm, we can just delete the constraints that are not in explicit form), and (iii) the collection of transformations are such that their exhaustive application terminates. Appendix II describes simplification steps for solving constraints involving projection and intersection. See [7, 9] for simplification steps involving more complex set operators.

# 5 Extensions

So far we have focussed on the use of set constraints to obtain an approximation of the possible run-time values of variables in a program. However, the basic process of constructing set constraints from a program and then solving these set constraints preserves numerous structural properties of a program. It is therefore possible, with only minor modifications to the set constraint algorithm, to compute approximations to a variety of other program properties. The first two extensions considered show how this property can be exploited for mode analysis and structure sharing analysis. The last extension deals with aspects of poly-variance (that is, separately analyzing different calls to procedures/functions).

### Mode Analysis (for Logic Programs)

To adapt set constraints to compute mode information for logic programs, we first change the underlying set of values from the set of all "ground" terms to the set of all terms. Then we replace the constant $\top$ by two new constants *ground* and *any*, which shall denote the set of all ground terms and the set of all (not necessary ground) terms respectively. Finally, we modify the definition of solutions of set constraints to account for these changes. For example, the minimum solution of $\mathcal{X} \supseteq f(ground, any)$ maps $\mathcal{X}$ into the set of all terms of form $f(t_1, t_2)$ such that $t_1$ is ground. The minimum solution of $\mathcal{X} \supseteq f(ground, any, a) \cap f(any, ground, any)$ maps $\mathcal{X}$ into the set of terms $f(t_1, t_2, a)$ such that $t_1$ and $t_2$ are both ground. The constraints generated for mode analysis are essentially unchanged, excepting that *any* and *ground* may be used to describe the initial goals.

The changes required to the algorithm to accommodate the constraints for mode analysis are minor and relate to the simplification of *ground* and *any*. The following table contains some sample simplification steps.

| if $\mathcal{C}$ contains the constraints: | then add the constraint(s): |
|---|---|
| $\mathcal{X} \supseteq \mathcal{Y} \cap \mathcal{Z},\ \ \mathcal{Y} \supseteq ground,\ \ \mathcal{Z} \supseteq any$ | $\mathcal{X} \supseteq ground$ |
| $\mathcal{X} \supseteq \mathcal{Y} \cap \mathcal{Z},\ \ \mathcal{Y} \supseteq f(\mathcal{Y}_1, \mathcal{Y}_2),\ \ \mathcal{Z} \supseteq ground$ | $\mathcal{X} \supseteq f(\mathcal{Y}_1', \mathcal{Y}_2'),$<br>$\mathcal{Y}_1' \supseteq \mathcal{Y}_1 \cap ground,\ \ \mathcal{Y}_2' \supseteq \mathcal{Y}_2 \cap ground$ |
| $\mathcal{X} \supseteq f_{(2)}^{-1}(\mathcal{Y}),\ \ \mathcal{Y} \supseteq ground$ | $\mathcal{X} \supseteq ground$ |

See [7] for further details. Note that the constants *ground* and *any* behave in essentially the same way as $\top$, and may appear in the output of the algorithm (that is, they may appear in the explicit representations that are computed by the algorithm). For example, when the program

```
app(nil, Y, Y).
app(cons(X', X), Y, cons(X', Z)) :- app(X, Y, Z).
```

is analyzed in the context of the goal `?- app(ground, ground, any)`, the output of the algorithm relevant to $Call_{app}$ and $Ret_{app}$ is

$$\begin{aligned} Call_{app} &= app(ground, ground, any) \\ Ret_{app} &= app(nil, ground, ground) \cup app(cons(ground, \mathcal{X}), ground, cons(ground, \mathcal{Z})) \\ \mathcal{X} &= nil \cup cons(ground, \mathcal{X}) \\ \mathcal{Z} &= ground \cup cons(ground, \mathcal{Z}) \end{aligned}$$

## Structure Sharing Analysis

Structure sharing analysis seeks information of the following form: given two variables, determine whether the bindings of these variables can "share" sub-structures (in the sense that the sub-structures have the same heap location). Such information can be used to determine when data structures can be updated in place or when they can be garbage collected. This kind of analysis may be performed by first giving each occurrence of a function symbol a unique label. Then set constraints are constructed as before, with care to preserve the labels on function symbols – call the resulting constraints *labeled* set constraints. The meaning of these constraints is defined by mapping set expressions into sets of *labeled* terms. The algorithm for solving labeled set constraints is a slight modification of the algorithm given previously. Again we give some example simplification steps:

| if $\mathcal{C}$ contains the constraints: | then add the constraint(s): |
|---|---|
| $\mathcal{X} \supseteq \mathcal{Y} \cap \mathcal{Z}, \;\; \mathcal{Y} \supseteq f^{\lambda_1}(\mathcal{Y}_1, \mathcal{Y}_2), \;\; \mathcal{Z} \supseteq f^{\lambda_2}(\mathcal{Z}_1, \mathcal{Z}_2),$ | $\mathcal{X} \supseteq f^{\lambda_1}(\mathcal{X}_1, \mathcal{X}_2)$<br>$\mathcal{X}_1 \supseteq \mathcal{Y}_1 \cap \mathcal{Z}_1, \;\; \mathcal{X}_2 \supseteq \mathcal{Y}_2 \cap \mathcal{Z}_2$ |
| $\mathcal{X} \supseteq f^{-1}_{(2)}(\mathcal{Y}), \;\; \mathcal{Y} \supseteq f^{\lambda_1}(\mathcal{Y}_1, \mathcal{Y}_2)$ | $\mathcal{X} \supseteq \mathcal{Y}_2$ |

Note that in the first step, the expression $f^{\lambda_1}(\mathcal{X}_1, \mathcal{X}_2)$ in the new constraint inherits the label of the left most component of the intersection. This is because $\cap$ is the means by which we model unification, and given that a variable is bound to something of the form $f^{\lambda_1}(\cdots)$, unification of this variable with another structure will not change this label. This means that $\cap$ is no longer a commutative operation, and this reflects the extra complexity of reasoning about this level of implementation behaviour.

In effect, this analysis treats each occurrence of a function symbol as an allocation operation that always returns the same memory cell. Put another way, all objects allocated at a given program point are indistinguishable. Such a treatment of allocation is essentially a consequence of the set constraint approach to analysis, since the underlying idea behind the set constraint approach is the association of a single set expression with each program expression, and it follows that all cells allocated by an occurrence of an allocation operation will be collected together. This represents a considerable approximation of the allocation behaviour of a program – we address this issue below in the broader context of poly-variance. We remark that the idea of analyzing allocation behaviour using the textual allocation points of a program has been widely used before. However, to our knowledge, the idea has not been used in the analysis of logic programs, or in conjunction with the use of set constraints.

The above sketch of sharing analysis is far from complete. For example, we have not described the interpretation of $\top$. To illustrate this issue, consider the following program, and a (somewhat simplified) selection of its set constraints:

```
?- Ⓐ, q(X, Y), Ⓑ, r(Y), Ⓒ.
q(W, W).
r(a).
```

$$
\begin{aligned}
Ret_q &\supseteq q(\top, \top) \\
Ret_r &\supseteq r(a) \\
\mathcal{X}^C &\supseteq q^{-1}_{(1)}(Ret_q) \\
\mathcal{Y}^C &\supseteq q^{-1}_{(2)}(Ret_q) \cap r^{-1}_{(1)}(Ret_r)
\end{aligned}
$$

Now, because of the aliasing of $X$ to $Y$, there is sharing of structure between $X$ and $Y$ after the execution of the goal :- p(X, Y), r(Y). However this is not directly reflected in the set constraints for the program. This issue is closely tied to the interpretation and manipulation of $\top$. In essence, special labels must be introduced for $\top$ constants. We omit the details.

## Poly-variance

At the heart of the set constraint approach to program analysis is the association of a set expression to each program term. The purpose of the set expression is to capture the runtime values of the corresponding term. In particular, the construction of constraints for procedures and functions is monomorphic (or *mono-variant* in the sense that all calls to a procedure or function are combined). Although this reduces the complexity of the analysis, it can result in a significant and unacceptable loss of information. Particularly problematic is analysis of functional programs involving polymorphic and higher order functions.

One way of addressing this shortcoming is to "split" different calls to functions and procedures and analyze them separately. In essence, this can be achieve by duplicating the constraints constructed for a function or procedure. We have been investigating this approach in the context of an implementation of set constraints for the analysis of functional programs. To allow for fast duplication of constraints, function are first translated into a *constraint template* form such that constraints corresponding to the analysis of the function can be rapidly constructed from the template.

However, the most important issue for an efficient poly-variant analysis is how to control of the duplication process. The approach we have taken involves splitting the analysis into two phases. The first phase is a (fast) mono-variant analysis of the program. The second phase then exploits the information obtained from the first phase to control a poly-variant analysis. Initial results suggest that for many functions, poly-variance leads to little benefit. In effect, the two phase approach provides a means by which effort may be concentrated on the aspects of the program that benefit most from poly-variance (see [8] for further details). More generally, the use of a simple underlying analysis and a judicial use of duplication (poly-variance) appears to be a useful paradigm for building realistic systems for accurate and understandable program analysis.

## 6 Advantages and Disadvantages of Set Constraints

We briefly evaluate the set constraint approach to program analysis in terms of accuracy, flexibility and computational cost. The use of set constraints is loosely based on the idea of treating program variables as sets of values, and in the process, all dependencies between variable values are ignored. In terms of accuracy, the use of set constraints has the advantage that there is no need to employ an underlying abstract domain. This means that the restrictions on accuracy that are inherent in the use of abstract domains[4] are avoided. Moreover, the use of set constraints provides a uniform and accurate treatment of data structures. There are no "depth-limits" or other *a priori* restrictions on the sets of values that can be manipulated. We contend that this reduces the potential for chaotic and unintuitive behaviour. (Compare this to abstract interpretation approaches to analysis in which the finitary nature of the underlying abstract domain typically leads to very subtle interactions between the abstract domain and the semantic operations of the language).

One clear disadvantage of the basic set constraint approach is that all inter-variable dependencies are ignored. Such dependencies can be crucial for some kinds of analysis such as mode analysis (see [12] for a discussion of this issue). Contrast this to abstract interpretation, where it is possible to retain a limited amount of information about dependencies (although there is, of course, additional computational cost associated with maintaining information about dependencies). Motivated by this observation, hybrid approaches that combine aspects of set constraints with abstract interpretation have been developed [12].

Another advantage of set constraints is their flexibility. In the course of implementing a number of prototype program analysis systems based on set constraints, it has been particularly easy to modify the systems to incorporate new features. For example, in a recently developed system for the analysis of ML programs, additions such as continuations, side-effects and exceptions involved very straight forward modifications. One reason for this appears to be the simple underlying approximation embodied in the set constraint approach.

One of the more difficult areas of evaluation is computational cost. A major problem here is that while worst case complexity costs can be obtained, it is not clear what conclusions we can draw from these results about the practicality of the various approaches, particularly on medium to large programs. Moreover, the technology for implementing set constraints is still in its infancy. With this in mind, we now briefly describe results from two implementations of set constraints for program analysis. The first is an implementation of set constraints for the analysis of logic programs [6] (this implementation included extensions for mode analysis). While substantial progress was made during the development of this implementation, the results indicate that we are still some distance from practical analysis of medium to large programs. Currently, top-down analysis of programs of the order of 50 rules can be achieved in a few seconds. As expected, analysis based on top-down semantics is considerably more expensive that for bottom-up semantics. One of the main lessons of this implementation was the expense of solving set constraints involving intersection. Much of the work of the implementation was directed at reducing this cost.

The second implementation effort provides a contrasting experience. This implementation [8] focussed on the analysis of ML programs. Results so far indicate that programs of over 1000

[4] Abstract domains employed in abstract interpretation are typically required to satisfy the finite ascending chains condition (or some analogous condition) and this represents a fundamental restriction on the accuracy of domains that can be employed.

lines can be routinely analyzed in a few seconds. The main reason for the substantial difference between the results from the two implementations seems to hinge on the fact that intersection is not used in the constraints generated from ML programs. Based on this observation, we are currently investigating ways of constructing constraints for logic programs that provide similar levels of accuracy, but either eliminate or substantially reduce the use of intersection.

We also observe that the results from the second implementation appear to out-perform current implementations of comparable abstract interpretation based analyses. We conjecture three possible reasons for this. First, the set constraint approach passes over the program text only once and essentially "pre-compiles" it into a convenient form for computation. In contrast, many abstract interpretation approaches work from the original program text, and so when recomputation of a program fragment is required in a fixed-point computation, the analysis makes a repeated passes over the program fragment. Clearly the technique of pre-compiling a program into a convenient equational/constraint form is equally applicable to abstract interpretation approaches. It appears likely that explicit manipulation of equations/constraints will be more widely used in all areas of program analysis.

The second reason is simply that all of aspects of approximation have been performed in the translation of the program into set constraints (and typically, this is a simple, linear-time process). In particular, the set constraint simplification algorithm does not have to perform any approximation operations. In contrast, the iterative fixed-point algorithm of abstract interpretation has to repeatedly perform approximation (abstraction) operations on the fly.

The third reason for better performance is based on the observation that set constraint approaches are inherently more incremental than abstract interpretation approaches. We illustrate two aspects of this with examples. First view both the interative fixed-point computation of abstract interpretation and the set constraint solving algorithm as involving the exhaustive propagation of information. Now, suppose the current "information" for $X$ indicates that $X$ is any even length list, and consider updating $X$ with new information that says $X$ may be any odd length list. In the set constraint algorithm, we only propagate the new fragment of information (that $X$ may now be an odd length list) to see if it changes anything depending on $X$. In contrast[5], an abstract interpretation approach updates the information for $X$ to "$X$ is now an arbitrary list", and then proceeds to fully recompute everything that depends on $X$. In other words, with set constraints, only new information is propagated. Whereas, in abstract interpretation, it is typically the case that the new information is "lubbed" with the old information, and the result (which is a combination of new information and old information) is then propagated.

As another example of the incrementality of the set constraint algorithm, suppose that a collection of constraints contains the constraint $\mathcal{Y} \supseteq cons(1, \mathcal{X})$ amongst others. Now, in an abstract interpretation approach, $\mathcal{Y}$ will be recomputed whenever $\mathcal{X}$ changes. In contrast, in the set constraint algorithm, this constraint will not cause $\mathcal{Y}$ to be recomputed whenever $\mathcal{X}$ changes. To continue the example, suppose that in addition to $\mathcal{Y} \supseteq cons(1, \mathcal{X})$, the constraints also contain $\mathcal{Z} \supseteq cons^{-1}_{(2)}(\mathcal{Y})$. Now, in abstract interpretation, any change to $\mathcal{X}$ will cause $\mathcal{Y}$ to be recomputed and this may then result in a recomputation of $\mathcal{Z}$. Using set constraints, a new constraint $\mathcal{Z} \supseteq \mathcal{X}$ will constructed by the set constraint simplification algorithm, and so changes to $\mathcal{X}$ will have a

[5]Strictly speaking, it is possible to construct an abstract interpretation that behaves like the set constraint approach in this regard. What is required is an abstract domain such that each abstract value can be considered to be the finite union of some collection of abstract values whose concretizations are disjoint. For a variety of reasons, abstract domains are rarely constructed to satisfy this property.

direct influence on $\mathcal{Z}$ (in effect, $\mathcal{Y}$ is removed from the chain of recomputations that are triggered when $\mathcal{X}$ is changed). To summarize, the set constraint algorithm computes with implicit or partial descriptions of information.

## 7 Related Work

The use of set constraints for analysis of programs dates back to the early works by Reynolds [17] (who presents an analysis for a first-order functional language), and Jones and Muchnick [14] (who present an analysis for a simple imperative language). Both works employ constraints whose only set operation is projection, and for example, ignore information about program tests. The use of constraints for the bottom-up analysis of logic programs was first considered by Mishra [16]. Although he used intersection, he used an alternative and less accurate "tuple-distributive" form of union. Later work by Jaffar and the present author developed constraints for the bottom-up analysis of logic programs using normal union and a generalized version of intersection [9]. Set constraints were first used for top-down analysis of logic programs by Jaffar and the present author [11]. [11] also presents more accurate constraints for the analysis of imperative programs, using a generalized notion of intersection to reason about program tests. The application of set constraints to the analysis of higher-order functional programs has been independently developed by Aiken and Wimmers [2] and the present author [7, 8]. We note that these two works have taken substantially different approaches.

The calculus of set constraints was first defined and studied in a general setting by Jaffar and the present author [10]. [10] also contained a decision procedure for a class of set constraints involving projection and intersection. Later works have provided algorithms for different classes of set constraints (Aiken and Murphy [1] have dealt with complementation and projection; Jaffar and the present author have dealt with set constraint operators that are designed for analyzing logic programs and imperative programs [7, 9], and combinations of set constraint techniques and abstract interpretation techniques [12]), as well as generalizing previous results (Bachmair, Ganzinger and Waldmann [3] establish a connection between certain kinds of set constraints and a fragment of logic shown decidable by Löwenheim, and in the process generalize the earlier results in [1] and [10]).

We also note that there are close connections between the use of set constraints and the use of grammars in program analysis, such as work by Jones [13].

## 8 Future Directions

One important issue that has received little attention is applications; that is, how can the information obtained using set constraints be put to use. In this regard, work is currently underway to use the implementation described in [8] in two ways: (i) for binding time analysis as part of a partial evaluator for ML, and (ii) to improve code generation for ML (applications such as data-structure representation and in-lining, as well as removal of redundant tests such as those involving array bounds checking, are currently being investigated).

Another important issue is implementation. Much work remains both in the areas of exper-

imenting with current implementations, as well as developing implementations of new forms of constraints, such as those outlined in [12]. Other extensions are possible to the basic set constraint framework. We believe that the approach outlined in [12] may provide a useful basis for analyzing constraint logic programs. Additionally, it seems possible to modify the construction of set constraints for logic programs so that the use of intersection is either eliminated or significantly reduced. Finally, extensions to deal with arithmetic operations in the context of functional programs have been considered in [8], but more work is required.

## References


[1] A. Aiken and E. Wimmers, "Solving Systems of Set Constraints", *Proc.* $7^{th}$ *IEEE Symp. on Logic in Computer Science*, Santa Cruz, pp. 329–340, June 1992.

[2] A. Aiken and E. Wimmers, "Type Inclusion Constraints and Type Inference", *Proc. 1993 Conf. on Functional Programming and Computer Architecture*, Copenhagen, pp. 31-41, June 1993.

[3] L. Bachmair, H. Ganzinger and U. Waldmann, "Set Constraints are the Monadic Class", Technical Report MPI–I–92–240, Max-Planck-Institute for Computer Science, December 1992.

[4] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints", *Proc.* $4^{th}$ *ACM Symp. on Principles of Programming Languages*, Los Angeles, pp. 238–252, January 1977.

[5] F. Gècseg and M. Steinby, *Tree Automata*, Akadèmiai Kiadò, Budapest, 1984.

[6] N. Heintze, "Practical Aspects of Set Based Analysis", *Proc. Joint International Conf. and Symp. on Logic Programming*, Washington D.C., MIT Press, pp. 765–779, November 1992.

[7] N. Heintze, "Set Based Program Analysis", Ph.D. thesis, School of Computer Science, Carnegie Mellon University, October 1992.

[8] N. Heintze, "Set Based Analysis of ML Programs", Carnegie Mellon University Technical Report CMU–CS–93–193, 20pp., July 1993.

[9] N. Heintze and J. Jaffar, "A Finite Presentation Theorem for Approximating Logic Programs", *Proc.* $17^{th}$ *ACM Symp. on Principles of Programming Languages*, San Francisco, pp. 197–209, January 1990. (A full version of this paper appears as IBM Technical Report RC 16089 (# 71415), 66 pp., August 1990.)

[10] N. Heintze and J. Jaffar, "A Decision Procedure for a Class of Herbrand Set Constraints", *Proc.* $5^{th}$ *IEEE Symp. on Logic in Computer Science*, Philadelphia, pp. 42–51, June 1990. (A full version of this paper appears as Carnegie Mellon University Technical Report CMU–CS–91–110, 42 pp., February 1991.)

[11] N. Heintze and J. Jaffar, "Set Based Program Analysis", draft manuscript, January 1991.

[12] N. Heintze and J. Jaffar, "An Engine for Logic Program Analysis", *Proc.* $7^{th}$ *IEEE Symp. on Logic in Computer Science*, Santa Cruz, pp. 318–328, June 1992.

[13] N. Jones, "Flow Analysis of Lazy Higher-Order Functional Programs", in *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin (Eds.), Ellis Horwood, 1987.

[14] N. Jones and S. Muchnick, "Flow Analysis and Optimization of LISP-like Structures", *Proc.* $6^{th}$ *ACM Symp. on Principles of Programming Languages*, San Antonio, pp. 244–256, January 1979.

[15] R. Milner, M. Tofte and R. Harper, "The Definition of Standard ML", MIT Press, 1990.

[16] P. Mishra, "Toward a Theory of Types in PROLOG", *Proc.* $1^{st}$ *IEEE Symp. on Logic Programming*, Atlantic City, pp. 289–298, 1984.

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

## Appendix I: Regular Term Grammars

A *regular term grammar* [5] consists of a set of non-terminals, a set of function symbols (each with a unique arity) and a finite set of productions. A *production* is of the form $nt \Rightarrow t$ such that $nt$ is a non-terminal and $t$ is a term. A *term* is either a non-terminal or of the form $f(t_1, \ldots, t_n)$ where $f$ is an $n$-ary function symbol and each $t_i$ is a term. The productions of a grammar define a derivability relation in the obvious way: $t_1 \Rightarrow t_2$ if there is a production $nt \Rightarrow t$ and $t_2$ is obtained from $t_1$ by replacing an occurrence of $nt$ in $t_1$ by $t$. Let $\Rightarrow^\star$ denote the transitive reflexive closure of $\Rightarrow$. The language corresponding to a non-terminal $nt$ is defined by the set:

$\{t : \ nt \Rightarrow^\star t$ and $t$ does not contain non-terminal symbols$\}$

For example, consider the grammar with non-terminals *list* and *int*, function symbols $cons$, $nil$, $succ$ and $zero$, and productions

$$\begin{array}{lcl} int & \Rightarrow & succ(int) \\ int & \Rightarrow & zero \\ list & \Rightarrow & cons(int, list) \\ list & \Rightarrow & nil. \end{array}$$

The language of *int* is the set of integers in successor-zero notation, and the language of *list* is the set of lists of such integers.

A set of terms is *regular* if there is a term grammar such that $S$ is the language of some non-terminal in the grammar. Such a grammar provides a representation of $S$ that is explicit in the sense that there are straightforward polynomial-time algorithms to determine membership and emptiness. Moreover, given such representations for two sets, we can find representations of their intersection, union, complementation or determine containment relationships by using standard algorithms for intersection, union, complementation and containment of regular term grammars. In short, a regular term grammar representation of a set of terms exposes much of the internal structure of the set.

## Appendix II: Sample Simplification Steps

We describe the simplification steps for solving constraints involving projection and intersection. The first step is essentially a bookkeeping transformation that simplifies the other transformations. In essence, it ensures that if $\mathcal{C}$ contains a chain of constraints $\mathcal{X}_1 \supseteq \mathcal{X}_2 \supseteq \cdots \supseteq \mathcal{X}_n \supseteq a$ where $a$ is a non-variable atomic expression, then $\mathcal{C}$ also contains $\mathcal{X}_1 \supseteq a$.

**Transformation 1 (Variable)** *If $\mathcal{X} \supseteq \mathcal{Y}$ and $\mathcal{Y} \supseteq a$ appear in $\mathcal{C}$ and $a$ is an atomic expression that is not a set variable, then add $\mathcal{X} \supseteq a$ to $\mathcal{C}$.* □

The next transformation step deals with projection. It is essentially the general case of the "cancellation" step outlined in Section 4. The "nonempty" condition in this step is needed to deal with constraints such as $\mathcal{X} \supseteq f^{-1}_{(1)}(\mathcal{Y}),\ \mathcal{Y} \supseteq f(a, \mathcal{Z})),\ \mathcal{Z} \supseteq a \cap b$. The minimum solution of these constraints maps both $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{Z}$ into the empty set. Now, if the projection transformation is applied without the nonempty check, then the constraint $\mathcal{X} \supseteq a$ would be added, and this would change the minimum solution so that $\mathcal{X}$ maps into $\{a\}$. To determine whether or not an atomic expression is empty at some stage in the algorithm, we proceed as follows. We associate a boolean value $nonempty(\mathcal{X})$ with each variable $\mathcal{X}$ that appears in the constraints. Initially each value $nonempty(\mathcal{X})$ is set to *false*. These values are then repeatedly updated using the following step: if $nonempty(\mathcal{X})$ is *false* and $\mathcal{X} \supseteq ae$ appears in $\mathcal{C}$ where $ae$ is an atomic expression such that $nonempty(\mathcal{Y})$ is *true* for all variables $\mathcal{Y}$ appearing in $ae$, then update $nonempty(\mathcal{X})$ to *true*. Finally, $nonempty(ae)$ is true if $nonempty(\mathcal{Y})$ for all variables $\mathcal{Y}$ that appear in the atomic expression $ae$.

**Transformation 2 (Projection)** *If $\mathcal{X} \supseteq f^{-1}_{(i)}(\mathcal{Y})$ and $\mathcal{Y} \supseteq f(ae_1, \ldots, ae_n)$ appear in $\mathcal{C}$ and nonempty$(ae_i)$, $i = 1..n$, then add $\mathcal{X} \supseteq ae_i$ to $\mathcal{C}$.* □

The last transformation simplifies intersection, and its main complexity is that it introduces new set variables. To ensure termination, these new variables must be carefully controlled using a special naming scheme. Specifically, each introduced variable is of the form $\mathcal{V}_{\{ae_1,\ldots,ae_n\}}$, $n \geq 1$, where the $ae_i$ are atomic expressions appearing in the constraints at the start of the algorithm. Intuitively such a variable, call it a *composite variable*, is equivalent to the expression $ae_1 \cap \cdots \cap ae_n$. Thus a variable such as $\mathcal{V}_{\{f(\mathcal{X}),\mathcal{Y},f(\mathcal{Z})\}}$ not only represents $f(\mathcal{X}) \cap \mathcal{Y} \cap f(\mathcal{Z})$, but also expressions such as $\mathcal{V}_{\{f(\mathcal{X}),\mathcal{Y}\}} \cap f(\mathcal{Z})$ and $\mathcal{V}_{\{f(\mathcal{X}),f(\mathcal{Z})\}} \cap \mathcal{V}_{\{f(\mathcal{X}),\mathcal{Y}\}}$. To facilitate the manipulation of composite variables, introduce a mapping $\mathcal{N}$ such that $\mathcal{N}(V_T)$ is $T$, and $\mathcal{N}(ae)$ is $\{ae\}$ if $ae$ is not of the form $V_T$. For example, $\mathcal{N}(\mathcal{V}_{\{f(\mathcal{X}),f(\mathcal{Z})\}}) = \{f(\mathcal{X}), f(\mathcal{Z})\}$, $\mathcal{N}(\mathcal{X}) = \{\mathcal{X}\}$, and $\mathcal{N}(f(\mathcal{X})) = \{f(\mathcal{X})\}$. The intersection transformation can now be stated as follows:

**Transformation 3 (Intersection)** *If $\mathcal{X} \supseteq \mathcal{Y}_1 \cap \cdots \cap \mathcal{Y}_m$ and $\mathcal{Y}_i \supseteq f(ae_{i,1}, \ldots, ae_{i,n})$, $i = 1..m$, then let $S_j$ be $\mathcal{N}(ae_{1,j}) \cup \cdots \cup \mathcal{N}(ae_{m,j})$, $j = 1..n$, and add the constraint $\mathcal{X} \supseteq f(\mathcal{V}_{S_1}, \ldots, \mathcal{V}_{S_n})$ to $\mathcal{C}$. For each $j$, if $\mathcal{C}$ does not contain a constraint of the form $\mathcal{V}_{S_j} \supseteq se$, then add $\mathcal{V}_{S_j} \supseteq ae_{1,j} \cap \cdots \cap ae_{m,j}$ to $\mathcal{C}$.* □

## Appendix III: Example Logic Program and Its Analysis

The following program is a complete listing of an interpreter for a small imperative language.

```
prog_var(x).
prog_var(y).
prog_var(z).

diff_var(x, y).
diff_var(x, z).
diff_var(y, x).
diff_var(y, z).
diff_var(z, x).
diff_var(z, y).

prog_const(a).
prog_const(b).
prog_const(c).

diff_const(a, b).
diff_const(a, c).
diff_const(b, a).
diff_const(b, c).
diff_const(c, a).
diff_const(c, b).

lookup_var(Var, [[Var, E] | _Env], E).
lookup_var(Var, [[Var2, _E2] | Env], E) :-
                diff_var(Var, Var2),
                lookup_var(Var, Env, E).

change_env(Var, E, [[Var, _E2] | Env], [[Var, E] | Env]).
change_env(Var, E, [], [[Var, E]]).
change_env(Var, E, [[Var1, E1] | Env1], [[Var1, E1] | Env2]) :-
                diff_var(Var, Var1),
                change_env(Var, E, Env1, Env2).

eval_exp(C, _Env, C) :-
                prog_const(C).
eval_exp([], _Env, []).
eval_exp([E | F], Env, [E1 | F1]) :-
                eval_exp(E, Env, E1),
                eval_exp(F, Env, F1).
eval_exp(car(E), Env, F1) :-
                eval_exp(E, Env, [F1 | _F2]).
eval_exp(cdr(E), Env, F2) :-
                eval_exp(E, Env, [_F1 | F2]).
eval_exp(Var, Env, E) :-
                prog_var(Var),
                lookup_var(Var, Env, E).
```

```
not_equal_val(C1, C2) :-
                diff_const(C1, C2).
not_equal_val(C, []) :-
                prog_const(C).
not_equal_val([], C) :-
                prog_const(C).
not_equal_val(C, [_E | _F]) :-
                prog_const(C).
not_equal_val([_E | _F], C) :-
                prog_const(C).
not_equal_val([], [_E | _F]).
not_equal_val([_E | _F], []).

eval_cond(match_cons(E), Env) :-
                eval_exp(E, Env, [_E1 | _E2]).
eval_cond(eq(E1, E2), Env) :-
                eval_exp(E1, Env, F),
                eval_exp(E2, Env, F).
eval_cond(and(C1, C2), Env) :-
                eval_cond(C1, Env),
                eval_cond(C2, Env).
eval_cond(or(C1, _C2), Env) :-
                eval_cond(C1, Env).
eval_cond(or(_C1, C2), Env) :-
                eval_cond(C2, Env).
eval_cond(not(C), Env) :-
                not_eval_cond(C, Env).


not_eval_cond(match_cons(E), Env) :-
                eval_exp(E, Env, F),
                not_equal_val(F, [_F1, _F2]).
not_eval_cond(eq(E1,E2), Env) :-
                eval_exp(E1, Env, F1),
                eval_exp(E2, Env, F2),
                not_equal_val(F1, F2).
not_eval_cond(and(C1, _C2), Env) :-
                not_eval_cond(C1, Env).
not_eval_cond(and(_C1, C2), Env) :-
                not_eval_cond(C2, Env).
not_eval_cond(or(C1, C2), Env) :-
                not_eval_cond(C1, Env),
                not_eval_cond(C2, Env).
not_eval_cond(not(C), Env) :-
                eval_cond(C, Env).

eval_stat(assign(Var, E), Env1, Env2) :-
                prog_var(Var),
                eval_exp(E, Env1, F),
                change_env(Var, F, Env1, Env2).
eval_stat(if(C, S1, _S2), Env1, Env2) :-
```

```
                eval_cond(C, Env1),
                eval_stat(S1, Env1, Env2).
eval_stat(if(C, _S1, S2), Env1, Env2) :-
                not_eval_cond(C, Env1),
                eval_stat(S2, Env1, Env2).
eval_stat(while(C, S), Env1, Env2) :-
                eval_cond(C, Env1),
                eval_stat(S, Env1, Env1a),
                eval_stat(while(C, S), Env1a, Env2).
eval_stat(while(C, _S), Env, Env) :-
                not_eval_cond(C, Env).
eval_stat(sq(S1, S2), Env1, Env2) :-
                eval_stat(S1, Env1, Env1a),
                eval_stat(S2, Env1a, Env2).
eval_stat(nop, Env, Env).
```

The top-down analysis of this program starting with the goal `?- eval_stat(`*any*`, [],` *any*`)` is performed in 5.3s (on a Sun Sparc 1+ with 24MB, running Mach), and yields the following information for the third argument of the goal:

$$
\begin{aligned}
Env &= nil \cup cons(cons(Var, cons(Val, nil)), Env) \\
Val &= c \cup b \cup a \cup nil \cup cons(Val, Val) \\
Var &= z \cup y \cup x
\end{aligned}
$$

## Appendix IV: Example ML Program and Its Constraints

An example program (involving the map function) and its set constraints are given below. In the minimum solution of these constraints, $\mathcal{F}$ is mapped into $\{id\}$, $\mathcal{X}$ is mapped into $\{1,2,3\}$, $\mathcal{L}$ is mapped into the set of lists $\{[2,3],[3],[\,]\}$, and $\mathcal{E}$ is mapped into the set of all lists constructed from 1, 2 and 3.

```
let fun map(F, cons(X, L)) = cons(F X, map(F, L))
      | map _ = nil
  and id Y = Y
in
   map(id, cons(1, cons(2, cons(3, nil))))
end
```

$$
\begin{aligned}
Ret_{map} &\supseteq cons(apply(\mathcal{F},\ \mathcal{X}),\ apply(map,\ (\mathcal{F},\ \mathcal{L}))) \\
Ret_{map} &\supseteq nil \\
\mathcal{F} &\supseteq cons^{-1}_{(1)}(Call_{map}) \\
\mathcal{X} &\supseteq cons^{-1}_{(1)}(cons^{-1}_{(2)}(Call_{map})) \\
\mathcal{L} &\supseteq cons^{-1}_{(2)}(cons^{-1}_{(2)}(Call_{map})) \\
Ret_{id} &\supseteq \mathcal{Y} \\
\mathcal{Y} &\supseteq Call_{id} \\
\mathcal{E} &\supseteq apply\,(map,\ (id,\ cons((1,\ cons(2,\ cons(3,\ nil))))))
\end{aligned}
$$