Set Constraints and Set-Based Analysis

NEVIN HEINTZE* and JOXAN JAFFAR†

May 1994

1 Introduction

Set expressions over a signature Σ of function symbols are a natural representation of sets of elements constructed from Σ, and set constraints express basic relationships between these sets. In the literature, set constraints have between used mostly in the context of uninterpreted (or Herbrand) function symbols. Although these applications have used set constraints in quite different ways, a common theme is the use of set constraints to obtain an *approximation* of some aspects of a program.

This paper contains two main parts. The first examines the set constraint calculus, discusses its history, and overviews the current state of known algorithms and related issues. Here we will also survey the uses of set constraints, starting from early work in (imperative) program analysis, to more recent work in logic and functional programming systems.

The second part describes *set-based analysis*. The aim here is a declarative interpretation of what it means to approximate the meaning of a program in just one way: ignore dependencies between variables, and instead, reason about each variable as the set of its possible runtime values. The basic approach starts with some description of the operational semantics, and then systematically replaces descriptions of environments (mappings from program variables to values) by set environments (mappings from program variables to *sets* of values) to obtain an approximate semantics called the *set-based program semantics*. The next step is to transform this semantics into a set constraint problem, and finally, the set constraints are solved.

2 Set Constraints

We present here the general calculus, followed by a brief survey of related work.

*School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.
†IBM T.J. Watson Research Center, PO Box 704, Yorktown Heights, NY 10598.

2.1 The Calculus

The set constraint calculus is parameterized by an underlying domain of discourse, and a set of functions Σ. For the purposes of most this paper, we choose Σ to be a given collection of uninterpreted function symbols, and the domain of discourse is then the ground terms constructed from Σ. In addition to Σ, we consider a fixed set of set operators consisting of union, intersection, complementation and projections of Σ functions.

A *set expression* is either a set variable (denoted $\mathcal{V}$, $\mathcal{W}$, $\mathcal{X}$, $\mathcal{Y}$, etc.), or of one of the forms $f(se_1, \ldots, se_n)$ or $op(se_1, \ldots, se_n)$, where $f \in \Sigma$, the se_i are set expressions, and op is a set operator. The set operators include union, intersection, complementation and projection (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 (some works use 1 and 0 instead of $\top$ and $\bot$). A *set constraint* is of the form $se \supseteq se'$ where se and se' are set expressions. We write $se = se'$ as an abbreviation for the two constraints $se \supseteq se'$ and $se' \supseteq se$.

A *solution* to a collection $\mathcal{C}$ 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 terms. Such a mapping can be extended to map from set expressions into sets of values:

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

$\mathcal{I}$ is a *solution* of a collection of constraints $\mathcal{C}$ if $\mathcal{I}(se) \supseteq \mathcal{I}(se')$ for each constraint $se \supseteq se'$ 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 $\mathcal{C}$. 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 $\{f^5(a), f^{11}(a), f^{17}(a), \ldots\}$.

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

In general, a collection of set constraints does not always have a unique smallest solution. For example consider the constraint $\mathcal{X} \cup \mathcal{Y} = a$ which has two minimal solutions: one that maps X to $\{a\}$ and $\mathcal{Y}$ to the empty set, and the other that maps X to the emtpy set and $\mathcal{Y}$ to $\{a\}$. For certain kinds of program analysis, it is natural to consider sub-classes of set constraints for which least models always exist. For example, consider constraints of the form $\mathcal{X} \supseteq se$ where $\mathcal{X}$ is a set variable and se is a set expression that does not use complementation. Such constraints always have a least solution. Somewhat more general are the *definite* set constraints, which have the form $a \supseteq se$ where a is a set expression that is "atomic" in the sense that it is constructed solely from set variables and function symbols, and se is a set expression that does not use complementation. A collection of definite set constraints is such that whenever it has a solution, it will in fact have a least solution. Further, it can be shown that this solution is *regular* in the sense that every variable is a regular set, that is, a set accepted by a nondetermistic tree automaton.

2.2 A Brief History

The use of set constraints for analysis of programs dates back to the early works by Reynolds [29] (who presents an analysis for a first-order functional language), and Jones and Muchnick [22] (who present an analysis for a simple imperative language). In both of these works, the set constraints used are quite simple: the only set operations employed are union and projection (there are no intersections or quantified expressions). We say more about these applications in the next subsection.

The general calculus of set constraints, as defined above, was first formalized and studied in a general setting in [17]. This work also presented a decision procedure for the class of definite set constraints (recall that definite constraints do not contain the complement symbol, and are restricted to the form $a \supseteq se$ where the set expression a contain only variables and function symbols). This procedure further provides an explicit representation of the least model of a (satisfiable) collection of definite set constraints. [17] also posed decidability of the satisfiability problem for general set constraints as an open question.

Later, [1] proved the decidability of a different, and incomparable, class: the *positive* set constraints. These are defined simply to be set constraints not involving projection. This procedure reduces the constraints into a simpler form. When reduction terminates without detecting inconsistency, the resulting constraints are evidently satisfiable. Note that satisfiable positive set constraints do not always have a least model. Subsequently, [9] provided an alternative procedure using tree automata techniques. Starting with Rabin's result [28] that the theory of k-successors is decidable, they generalized the Rabin automaton to accomodate positive set constraints. They further showed that satisfiable positive constraints always have a regular solution (all variables are assigned a regular set), and a minimal and maximal regular solution.

While the class of definite constraints and the class of positive classes are not comparable, the work [5] proved decidability of a class subsuming the two. Briefly, the set constraints considered here are the positive ones, extended to allow projections in a restricted way. The importance of this work probably lies more in the technique used: it is proved that set constraints can be written into equivalent formulas in the *monadic class*, that is, first-order formulas with unrestricted quantification, but no function symbols and only monadic predicate symbols. The transformation is simple and elegant, and gave rise to complexity results on set constraints based on similar results in the monadic class.

The next step was taken by [10], who proved that *negative* set constraints, ie. the extension to positve constraints with negations of subset relationships such as $se_1 \not\subseteq se_2$, remains decidable. Once again, tree automata techniques were used here. An alternative procedure was then given by [4], by reduction to a number-theoretic decision problem. Subsequently, [6] used the abovementioned translation of set constraints to the monadic class to provide a straightforward procedure for deciding negative set constraints. Note that none of these works on negative constraints deal with projections.

In summary, the state of the art for the set constraint decision problem is largely determined by the reduction to the monadic class of formulas. The main question remains how to deal with (unrestricted) projection. At the time of writing, we have verbal communication [26] indicating that the proof in [6] can be extended to solve this problem. Thus the question of whether the general set constraint problem is open, now becomes open!

2.3 Applications

Early works

Two important early works are by Jones and Muchnick [22] and Reynolds [29]. In [22], an analysis is described for an imperative language with LISP-like data structures. The essence here is the construction of set constraints corresponding to a program that capture the flow of values from one variable to another as the program is executed. However, the set constraints here are restricted so that they can be solved by a fairly straightforward algorithm. In particular, the set constraints do not contain a notion of intersection, and their only operation is projection (corresponding to decomposition of data structures). Hence they are not expressive enough to capture a number of important components of programs. For example all information about the conditions in conditional statements is completely omitted. Further, information relating to well definedness of expressions is ignored (for example, after a statement $X = car(Y)$, it must be the case that Y is of the form $cons(\cdots)$ because otherwise the program would have terminated with an error).

In contrast, the earlier paper [29] used set constraints to compute data type definitions for program variables in a first order functional language. The constraints used are similar to those used in [22]. Again the only set operation of the constraints is projection, and so the program approximations obtained can be considerably inaccurate.

In summary, the set constraints used in these early works are simple, but the program approximations that they define are not very accurate. These works viewed set constraints as a tool for obtaining information about the program, and the constraints themselves incorporate a number of *ad hoc* approximations in addition to ignoring inter-variable dependencies. As a result, there is no simple connection between the program and its approximation. This particular shortcoming is one of the motivations for set-based analysis, discussed later in this paper.

Logic Programs

The use of set constraints for the bottom-up analysis of logic programs was first considered in [25]. The set constraints in this relatively early work were rather specialized and used a form of approximation called *tuple-distributive* closure (hereafter just called *closure*). This closure, which was subsequently used in some later works, has the effect of enlarging a set of terms S into S^* as follows:

$$S^\star \stackrel{\text{def}}{=} \{c : c \text{ is a constant in } S\} \;\cup\; \bigcup_{f \in \Sigma} f\left((f^{-1}_{(1)}(S))^\star, \ldots, (f^{-1}_{(arity(f))}(S))^\star\right)$$

where $f(S_1, \ldots, S_n)$ denotes the set $\{f(s_1, \ldots, s_n) : s_i \in S_i\}$ and $f^{-1}_{(i)}(S)$ denotes the set $\{s_i : f(s_1, \ldots, s_n) \in S\}$. Thus for example, closing the set $\{f(a,b), f(c,d)\}$ produces $\{f(a,b), f(a,c), f(b,d), f(c,d)\}$. The set constraints used in [25] are like the general ones defined above, except that the union operation is interpreted to be the closure of the union of sets.

A different approach to approximation starts from the (bottom-up) fixpoint operator T_P of a program P, and the approximate meaning of a program is obtained by imposing closure on each iteration of the operator. For example, [32] defined the operator $Y_P(S) \stackrel{\text{def}}{=} (T_P(S))^*$ and the approximate meaning of the program is the least fixpoint $lfp(Y_P)$ of Y_P (which is always larger than the exact meaning, $lfp(T_P)$). In [16], a more accurate operator τ_P was used. (Roughly, Y_P ignores inter-argument dependencies, while τ_P ignores only inter-variable dependencies.) A more recent work [8] used the closure operators (in conjunction with another approximation technique called widening) to define and compute a program approximation.

The relationship between these closure-based fixpoint operators and set constraints was described in [18]. One result is that the models of the set constraints in [25], essentially correspond to the fixpoints of Y_P. A similar result was that the other fixpoint operator τ_P corresponded to certain formulas obtained from the program. These formulas are similar to but more general than set constraints. The main point here was that the least fixed-point of τ_P provided a more accurate and intuitive notion of approximation, and importantly, the approximation is decidable. It is open as to whether $lfp(Y_P)$ is decidable.

Functional Programs

The general approach of [22, 29] has been extended by [21] to deal with higher-order functions. This approach has been further developed for binding time analysis [24], garbage collection [20] and globalization of function parameters [30]. One presentational difference in these works is the use of various extensions of regular grammars instead of constraints.

Subsequently, a number of set constraint approaches have been developed for the analysis of higher-order functional languages (see, for example, [27, 12, 2, 3, 13]). Perhaps the most developed of these approaches are those by [12, 13] and [2, 3]. The former starts with an operational semantics, and develops a set-based analysis for this semantics. The constraints that arise are briefly sketched in Section 3.3. In the latter, a denotational model of the program inspires the extraction of "type constraints", which are essentially set constraints (involving intersection and complement but not projection) over a domain of downward closed sets of finite elements (essentially the "ideal" model of types). We note that both works include a mechanism for reasoning about non-emptiness of sets (these are called "conditional types" in [3]).

Sorted Unification

Broadly, sorted unification is the problem of unifying two terms in the context of a sort theory, the latter imposing constraints on the values that certain variables can take. The sort theory is typically presented as a sort signature, indicating the hierarchical arrangement of the various sorts, together with a specification on the sorts of the various function symbols. For example,

$$\{even \subseteq int, \quad odd \subseteq int, \quad succ : odd \longrightarrow even, \quad succ : even \longrightarrow odd\}$$

specifies that the sorts *even* and *odd* both belong to *int*, that the function *succ* maps an even integer into an odd one, and vice versa. Such constraints can be naturally specified in set constraints:

$$Int = Odd \cup Even, \quad Odd = 0 \cup succ(Even), \quad Even = 0 \cup succ(Odd)$$

In general, sorted unification is decidable only when the sort theory is restricted in some way. In the literature, a typical restriction is that the sorts are regular sets. In [31], a restricted class of set constraints is used to represent the sort theory, and a new sorted unification algorithm is presented. This work shows that further development in set constraints may be useful for sorted unification.

3 Set-Based Analysis

The basic approach of set-based program analysis starts with some description of the operational semantics. Typically, such a description involves environments, which describe the values that each variable may assume at runtime. The next step is a systematic replacement of environments into *set environments*, which map variables into sets of values, as opposed to a single value. This fundamental step gives rise to the notion of a set-based semantics of a program. Next, the set-based semantics is reduced to a set constraint problem, and finally, the set constraints are solved.

In this paper, we will not go through this process in much formal detail. These details can be found in [12]. Instead, we will show by examples how set constraints indeed model the desired approximation from program fragments.

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. 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 the list $[3,4]$. 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).

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

In summary for this subsection, 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.

3.1 Imperative Programs

The example imperative programs considered above do 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 (A), (B), (C) and (D) indicate points in the program (note that (D) indicates the point at the end of the program). Corresponding to this program, we can construct the following constraints.

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

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:

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

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

3.2 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).
```

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

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).
```

$$
\begin{aligned}
Call_p &\supseteq p(a \cup b \cup c \cup d) \\
Ret_p &\supseteq p(\mathcal{X}^C) \\
\mathcal{X}^A &\supseteq p^{-1}_{(1)}(Call_p) \\
\mathcal{X}^B &\supseteq p^{-1}_{(1)}(Call_p) \cap q^{-1}_{(1)}(Ret_q) \\
\mathcal{X}^C &\supseteq p^{-1}_{(1)}(Call_p) \cap q^{-1}_{(1)}(Ret_q) \ \cap \ r^{-1}_{(1)}(Ret_r) \\
Call_q &\supseteq q(\mathcal{X}^A) \\
Ret_q &\supseteq (q(a) \ \cup \ q(b)) \ \cap \ Call_q \\
Call_r &\supseteq r(\mathcal{X}^B) \\
Ret_r &\supseteq (r(b) \ \cup \ r(c)) \ \cap \ Call_r
\end{aligned}
$$

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

$\{p(b)\}$, $\mathcal{X}^A$ 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).
```

$$\begin{aligned}
Call_p &\supseteq p(a \cup b \cup c \cup d)\\
Ret_p &\supseteq p(\mathcal{X}^C)\\
\mathcal{X}^A &\supseteq p^{-1}_{(1)}(Call_p)\\
\mathcal{X}^B &\supseteq p^{-1}_{(1)}(Call_p)\\
\mathcal{X}^C &\supseteq p^{-1}_{(1)}(Call_p) \cap q^{-1}_{(1)}(Ret_q) \ \cap \ r^{-1}_{(1)}(Ret_r)\\
Call_q &\supseteq q(\mathcal{X}^A)\\
Ret_q &\supseteq (q(a) \ \cup \ q(b)) \ \cap \ Call_q\\
Call_r &\supseteq r(\mathcal{X}^B)\\
Ret_r &\supseteq (r(b) \ \cup \ r(c)) \ \cap \ Call_r
\end{aligned}$$

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).
```

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

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.

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).
```

$$\begin{aligned}
Ret_p &\supseteq p(\mathcal{X}, \mathcal{Y})\\
\mathcal{X} &\supseteq q^{-1}_{(1)}(Ret_q)\\
\mathcal{Y} &\supseteq \top\\
Ret_q &\supseteq q(a)
\end{aligned}$$

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).
```

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

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:

$$
\begin{aligned}
Ret_p &\supseteq p(\mathcal{X}, \mathcal{Y}) \\
\mathcal{X} &\supseteq \Big\{X : \exists Y \ \Big(q(X,Y) \in Ret_q \ \wedge \ r(X,Y) \in Ret_r\Big)\Big\} \\
\mathcal{Y} &\supseteq \Big\{Y : \exists X \ \Big(q(X,Y) \in Ret_q \ \wedge \ r(X,Y) \in Ret_r\Big)\Big\} \\
Ret_q &\supseteq q(a,b) \ \cup \ q(b,a) \\
Ret_r &\supseteq r(a,a)
\end{aligned}
$$

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 [12, 16] for further details).

3.3 Functional Programs

To analyze functional languages such as Standard ML [23], 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
```

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

Again, more complex set operators can be introduced to provide more accurate modeling of certain aspects of the language (particularly case statements). See [12, 13] for further details. The complexity of solving the set constraints is $O(n^3)$ [13]. This basic formulation of constraints has been extended to deal with arrays, continuations and exceptions.

3.4 Comparison with Other Analysis Techniques

A key advantage of set-based analysis (and, more generally, the use of set constraints to perform program analysis), in comparison to standard abstract interpretation techniques [7], is that there is no underlying abstract domain. When using an abstract domain, the requirement of "finite ascending chains" is typically required for termination, and this limits the usable abstract domains. A remedy is to use techniques of "narrowing" and "widening". Even so, termination continues to place a fundamental restriction on the accuracy of the treatment of values. Avoiding the use of abstract domains leads to important advantages in terms of accuracy and uniformity. In particular, set-based analysis does not use "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.

Another benefit of the simplicity and uniformity of the approximation embedded in set-based analysis is that the analysis is extensible and flexible. In the course of implementing a number of prototype set-based analysis systems, we have observed that modifications to incorporate new features are often straightforward. For example, during the development of a system for the analysis of ML programs, the treatment of continuations, side-effects and exceptions required only minor modifications. There appear to be two reasons for this. First, because set-based analysis has a simple and intuitive definition, it is usually straightforward to determine how to treat new features. Second, because the analysis has a uniform definition, the treatment of one component of a language is largely independent of the treatment of other aspects of the language, and so the analysis can be extended in a modular manner.

Of course, the main limitation of set-based analysis is that all inter-variable dependencies are ignored. Such dependencies can be crucial for some kinds of analysis such as mode analysis (see [19] for a discussion of this issue). In contrast, abstract interpretation techniques can 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 [19].

3.5 Efficiency Issues

It is difficult to quantify a comparison between set-based analysis and standard analysis techniques. 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. Moreover, the technology for implementing set constraints is still in its infancy. With this in mind, we now briefly describe results from implementations of set-based analysis for two different languages.

The first deals with analysis of logic programs [11], and computes type, mode and sharing information. This analysis has a worst case exponential complexity. 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 bottom-up semantics is considerably cheaper that for top-down semantics. One of the main lessons of this implementation is 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 [13] focussed on the analysis of ML programs. The core algorithm for this analysis is $0(n^3)$ on the size of the input program. Typical execution times are in the range of 200–400 lines per second for programs up to several thousand lines in length. 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.

The results from the second implementation out-perform current implementations of comparable abstract interpretation based approaches. There appear to be a number of reasons for this. In set-based analysis there is only one pass over the program text. In essence, this performs a "pre-compilation" of the program into a convenient computation form (set constraints). In contrast, many abstract interpretation systems repeatedly pass over (some representation of) program text during the iterative fixed-point computation. In set-based analysis, all approximation is carried out in the translation to set constraints, and so no approximation operations need to be done during the main computational component of the analysis (solving set constraints). Furthermore, set constraints are inherently more incremental than the iterative fixed-point computations of abstract interpretation. In essence, constraints provide a compact implicit representation of information. This representation supports computation over partial information that is particularly well suited to efficient program analysis. We refer to [15] for a deeper discussion of this issue.

3.6 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. We now illustrate this.

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 terms respectively. Finally, we modify the definition of solutions of set constraints to account for these changes. For example, the minimum solution of $\mathcal{X} \supseteq f(\mathit{ground}, \mathit{any})$ 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 $\mathcal{X} \supseteq f(\mathit{ground}, \mathit{any}, a) \cap f(\mathit{any}, \mathit{ground}, \mathit{any})$ 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 modifications for solving these new constraints involve steps such as simplifying $\mathit{ground} \cap \mathit{any}$ into ground, and $f(\mathit{any}) \cap \mathit{ground}$ into $f(\mathit{ground})$. See [12, 15] 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

$$\begin{aligned}
\mathit{Call}_{app} &= \mathit{app}(\mathit{ground}, \mathit{ground}, \mathit{any}) \\
\mathit{Ret}_{app} &= \mathit{app}(\mathit{nil}, \mathit{ground}, \mathit{ground}) \ \cup \ \mathit{app}(\mathit{cons}(\mathit{ground}, \mathcal{X}), \mathit{ground}, \mathit{cons}(\mathit{ground}, \mathcal{Z})) \\
\mathcal{X} &= \mathit{nil} \ \cup \ \mathit{cons}(\mathit{ground}, \mathcal{X}) \\
\mathcal{Z} &= \mathit{ground} \ \cup \ \mathit{cons}(\mathit{ground}, \mathcal{Z})
\end{aligned}$$

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. We refer to [12, 15] for further details.

Interpreted Function Symbols

The set constraints considered so far deal with uninterpreted symbols so as to correspond to the data constructors of the language at hand. For analysis of programs involving operations such as arithmetic, this approach must be generalized. One possibility is to compute descriptions of *how* arithmetic values are obtained. These descriptions are essentially terms built from arithmetic operations and integers. For example, the description of computations for a program variable x might be given by

$$\mathcal{X} = 0 \cup (\mathcal{X} + 1)$$

that is, the set of computations $\{0, 0+1, (0+1)+1, \ldots\}$. Clearly, the actual values of x are included in the set $\{0, 1, 2, \ldots\}$. [14] describes how this approach can be applied to the problem of removing array bounds checks, and this requires that the analysis also reason about arithmetic tests. An example of the kinds of descriptions that arise in this context is:

$$\mathcal{X} = 0 \cup [LE\ 10](X + 2)$$

where $[LE\ 10]$ is a "restriction" operator that essentially picks those elements from a set that are less than 10 (in general a restriction operator is of the form $[op\ se]$ where op is some arithmetic comparison operation and se is some set expression). The least model of the above equation maps $\mathcal{X}$ into $\{0, 2, 4, 6, 8\}$.

4 Conclusion

The calculus of set constraints was presented, and its history of basic results and applications briefly described. The approach of set-based analysis was then presented in an informal style, with a focus on the breadth of applicability of the technique. The relationship between set constraints and set-based analysis is roughly that the approximation of a program by ignoring inter-variable dependencies can be captured by set constraints. It was then argued that set-based analysis can provide accurate and efficient program analysis.

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] A. Aiken, E. Wimmers and T.K. Lakshman, "Soft Typing with Conditional Types" *Proc. 21^{st} ACM Symp. on Principles of Programming Languages*, Portland, OR, pp. 163–173, January 1994.

[4] A. Aiken, D. Kozen and E. Wimmers, "Decidability of Systems of Set Constraints with Negative Constraints", IBM Research Report RJ 9421, 1993.

[5] L. Bachmair, H. Ganzinger and U. Waldmann, "Set Constraints are the Monadic Class", *Proc. 8^{th} IEEE Symp. on Logic in Computer Science*, 75–83, 1993.

[6] W. Charatonik and L. Pacholski, "Negative Set Constraints: an Easy Proof of Decidability", *Proc. 9^{th} IEEE Symp. on Logic in Computer Science*, 1994, to appear.

[7] 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.

[8] J. Gallagher and D.A. de Wall, "Fast and Precise Regular Approximations of Logic Programs", *Proc. International Conf. on Logic Programming*, MIT Press, to appear 1994.

[9] R. Gilleron, S. Tison and M. Tommasi, "Solving Systems of Set Constraints using Tree Automata", *Proc. 10^{th} Annual Symposium on Theoretical Aspects of Computer Science*, pp. 505–514, 1992.

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

[11] 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.

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

[13] N. Heintze, "Set-Based Analysis of ML Programs", to appear, ACM Conference on Lisp and Functional Programming, 1994.

[14] N. Heintze, "Set-Based Analysis of Arithmetic", Carnegie Mellon University technical report CMU-CS-93-221, 20pp., December 1993.

[15] N. Heintze, "Set Constraints in Program Analysis", Workshop on Global Compilation, International Logic Programming Symposium, October 1993.

[16] 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.)

[17] 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.)

[18] N. Heintze and J. Jaffar, "Semantic Types for Logic Programs" in *Types in Logic Programming*, F. Pfenning (Ed.), MIT Press Series in Logic Programming, pp. 141–155, 1992.

[19] 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.

[20] T. Jensen and T. Mogensen, "A Backwards Analysis for Compile-Time Garbage Collection", *Proc. 3^{rd} European Symp. on Programming*, Copenhagen, LNCS 432, pp. 227–239, May 1990.

[21] 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.

[22] 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.

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

[24] T. Mogensen, "Separating Binding Times in Language Specifications", *Proc. Functional Programming and Computer Architecture*, London, ACM, pp. 12–25, September 1989.

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

[26] L. Pacholski, personal communication, March 1994.

[27] J. Palsberg and M. Schwartzbach, "Safety Analysis versus Type Inference for Partial Types" Information Processing Letters, Vol 43, pp. 175–180, North-Holland, September 1992.

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

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

[30] P. Sestoft, "Replacing Function Parameters by Global Variables", *Proc. Functional Programming and Computer Architecture*, London, ACM, pp. 39–53, September 1989.

[31] T.E. Uribe, "Sorted Unification using Set Constraints", *Proc. 11^{th} Intl. Conf. on Automated Deduction*, D. Kapur (Ed), Springer Verlag Lecture Notes in Computer Science, 1992.

[32] E. Yardeni and E.Y. Shapiro, "A Type System for Logic Programs", *Journal of Logic Programming*, Vol. 10, pp. 125 – 153, 1991. (An early version of this paper appears in *Concurrent PROLOG: Collected Papers*, Vol. 2, MIT Press, pp 211 - 244, 1987.)