Practical Aspects of Set Based Analysis

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

March, 1992

Summary

The set based approach to compile time analysis of logic programs has numerous advantages over other approaches, including a simple, intuitive definition, and a very accurate and uniform treatment of term structure. However the algorithms for set based analysis that have been presented to date have focussed on establishing decidability results, and do not provide a practical basis for program analysis. This paper demonstrates that very substantial progress can be made towards a practical system by redesigning the algorithms according to implementation criteria, designing appropriate representations techniques, and exploiting the redundancy which is typically present. An implementation is described and experimental evidence is presented which suggests that set based analysis can be made practical for comprehensive and accurate analysis of logic programs.

1 Introduction

The set based approach to program analysis [3, 4, 5] has its origins in the use of constraints to perform type analysis of programs [12, 8, 10]. In essence set based analysis involves first writing set constraints (a calculus for expressing relationships between sets of program values) to describe the runtime behaviour of a program, and then solving these constraints to find their least model. This approach uses a single notion of approximation: all dependencies between the values of program variables are ignored. A major advantage of set based analysis is that it provides a very accurate treatment of structures. In particular there is no a priori limit on the depth or shape of sets of terms that can be represented[1], and so the analysis can compute very precise typing information in a uniform manner. One feature of set based analysis is that it can be defined as an approximate program semantics, and this definition is simple, intuitive and independent of algorithmic notions. Furthermore, because set based analysis uses a very uniform notion of approximation, it yields more predictable information, which has important implications for scaleability.

[1] This is because set based analysis does not use an abstract domain, and this represents a key difference between set based analysis and abstract interpretation.

What has not been addressed is practicality. That is, given the set constraints corresponding to a program, can we efficiently solve them to find their least model? The main components of set constraints are operations of set projection, intersection and union, and it is clear that solving such constraints can be expensive in the worst case. (This is mainly due to the exponential behaviour of the intersection operation.) Exponential upper and lower bounds for one case of set constraint analysis are established in [13]. However it is not clear what these observations have to say about the practicality of set based analysis. Typical programs rarely exhibit the extremes of behaviour which are exploited to obtain worst case behaviour.

This paper is a progress report on an ongoing effort to incorporate set based analysis in an experimental compiler. While set based analysis has substantial advantages in terms of accuracy, it also has substantial disadvantages in terms of computational cost. To address the question of whether the cost is prohibitive, this paper develops and evaluates an implementation of set constraints. Although straightforward implementation of the algorithms described in [3, 4] is unworkable, substantial progress has been made towards a practical system by developing an algorithm specifically for implementation purposes, designing appropriate representations techniques and exploiting the redundancy which is typically present.

After reviewing how set constraints can be used for program analysis, we begin the main part of the paper (in Section 3) by reformulating the set constraint algorithm in a way which is more appropriate for implementation purposes. In section 4, we identify some of the central operations required by the algorithm. This motivates a number of key design decisions, which lead to a basic implementation approach. Section 5 describes a crucial modification for more effectively dealing with the intersection operation. Section 6 contains some experimental results, with particular focus on justifying the design decisions outlined in the paper.

2 Background: Set Constraints and Program Analysis

We briefly outline how set constraints may be used to obtain safe approximations of program execution (see [3, 4, 9] for more details). Specifically, for each point in the program, we wish to obtain a safe description of the possible substitutions that are encountered at that point during program execution. To this end, we first introduce set variables to capture the sets of values of each program variable at each point in the program. Then we write constraints between these sets to safely approximate the local consistency conditions implicit in the program. The following example illustrates a program and the constraints constructed to approximate its success set.

$$\begin{array}{ll} & Ret_p = p(\mathcal{X}) \\ p(X) \leftarrow q(X), r(X). & Ret_q = q(a) \cup q(f(\mathcal{Y})) \\ q(a). & Ret_r = r(f(\mathcal{Z})) \\ q(f(Y)) \leftarrow q(Y). & \mathcal{X} = q^{-1}(Ret_q) \cap r^{-1}(Ret_r) \\ r(f(Z)). & \mathcal{Y} = q^{-1}(Ret_q) \\ & \mathcal{Z} = \top \end{array}$$

The variables Ret_p, Ret_q and Ret_r capture the sets of ground atoms in the success set corresponding to the predicates p, q and r respectively. The variables $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{Z}$ capture the sets of values for the program variables X, Y and Z respectively. The symbols q^{-1} and r^{-1} denote projection with respect to q and r, so that $q^{-1}(Ret_q)$ denotes the set $\{t : q(t) \in Ret_q\}$. Hence, the constraint $\mathcal{X} = q^{-1}(Ret_q) \cap r^{-1}(Ret_r)$ indicates that the set of values for the program variable X equals those values t such that $q(t)$ is in Ret_q and $r(t)$ is in Ret_r. The symbol $\top$ denotes the set of all ground terms. A model of these constraints is a mapping from the variables $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ into sets of ground terms such that each constraint is satisfied. For example the mapping

$$
\begin{array}{lll}
\mathcal{X} & \mapsto & \{f(a), f(f(a)), f(f(f(a))), \cdots, \} \\
\mathcal{Y} & \mapsto & \{a, f(a), f(f(a)), f(f(f(a))), \cdots, \} \\
\mathcal{Z} & \mapsto & \{\text{all terms}\}
\end{array}
\qquad
\begin{array}{lll}
Ret_p & \mapsto & \{p(f(a)), p(f(f(a))), \cdots, \} \\
Ret_q & \mapsto & \{q(a), q(f(a)), q(f(f(a))), \cdots, \} \\
Ret_r & \mapsto & \{r(f(t)) : \text{for all terms } t\}
\end{array}
$$

is a model of the constraints, and is in fact the least model. Hence the constraints constructed for the program are exact in the sense that their least model describes the success set of the program. This is not true in general. By their nature, set constraints collect the values for each program variable into sets and hence they ignore dependencies between program variables. The following example shows a program, its constraints, and the least model of these constraints.

$$
\begin{array}{l}
p(X,Y) \leftarrow q(X,Y). \\
q(a,b). \\
q(c,d).
\end{array}
\qquad
\begin{array}{l}
Ret_p = p(\mathcal{X}, \mathcal{Y}) \\
Ret_q = q(a,b) \cup q(c,d) \\
\mathcal{X} = q^{-1}_{(1)}(Ret_q) \\
\mathcal{Y} = q^{-1}_{(2)}(Ret_q)
\end{array}
\qquad
\begin{array}{rll}
Ret_p & \mapsto & \{p(a,b), p(a,d), p(c,b), p(c,d)\} \\
Ret_q & \mapsto & \{p(a,b), p(c,d)\} \\
\mathcal{X} & \mapsto & \{a, c\} \\
\mathcal{Y} & \mapsto & \{b, d\}
\end{array}
$$

Here $q^{-1}_{(1)}(Ret_q)$ and $q^{-1}_{(2)}(Ret_q)$ denote projection on q at the first and second arguments respectively, so that $q^{-1}_{(1)}(Ret_q)$ is just $\{t_1 : q(t_1, t_2) \in Ret_q\}$.

The examples provided so far have shown the use of set constraints to approximate the success set of a program. However, the set constraint approach is by no means restricted to approximating the success set, but rather can be adapted in a straightforward manner to deal with other strategies such as top-down left-to-right execution or parallel execution. The following example illustrates a program and the constraints constructed to approximate a top-down left-to-right execution of the program.

$$
\begin{array}{l}
① : p(X) \leftarrow {}^{②}q(X),\ {}^{③}r(X). \\
④ : q(a). \\
⑤ : r(Y).
\end{array}
\qquad
\begin{array}{l}
Ret_p = p(\mathcal{X}^{①}) \\
Ret_q = q(a) \\
Ret_r = r(\mathcal{Y}^{⑤}) \\
Call_q = q(\mathcal{X}^{②}) \\
Call_r = r(\mathcal{X}^{③}) \\
\mathcal{X}^{②} = p^{-1}(Call_p) \\
\mathcal{X}^{③} = p^{-1}(Call_p) \cap q^{-1}(Ret_q) \\
\mathcal{X}^{①} = p^{-1}(Call_p) \cap q^{-1}(Ret_q) \cap r^{-1}(Ret_r) \\
\mathcal{Y}^{⑤} = r^{-1}(Call_r)
\end{array}
$$

The first stage of constructing these constraints is to introduce program points to indicate the various stages of execution of the body of a rule. For example, the point ② indicates program execution just before $q(X)$ is called in the body of the first rule. The point ③ similarly indicates execution just before $r(X)$. The point ① similarly indicates the point after execution of the first rule. Now, for each variable and relevant program point, we introduce a set to describe the values of the variable at that program point. For example $\mathcal{X}^{①}$, $\mathcal{X}^{②}$, and $\mathcal{X}^{③}$ respectively denote the values of $\mathcal{X}$ at the points ①, ② and ③. The main difference between these constraints and the previous constraints is that variables $Call_p$, $Call_q$ and $Call_r$ have been introduced to capture the possible calls to p, q and r respectively. For example the constraint $\mathcal{X}^{③} = p^{-1}(Call_p) \cap q^{-1}(Ret_q)$ indicates that the values of X at point ③ consists of all t such that $p(t)$ is in $Call_p$ and $q(t)$ is in Ret_q.

The construction of set constraints corresponding to a program has only been informally outlined here. Further details and correctness proofs can be found in [3, 6, 5, 9]. The key point is that, given a notion of program execution (such as topdown left-to-right, topdown parallel) we can give a schema for generating appropriate constraints from a program. We note that the constraints used in this paper are simpler and slightly less accurate than those used in [3]. However the differences are only minor, and it is easy to adapt the correctness proofs in [3] to the constraints considered here. Most importantly, the constraints used here provide a similar uniform treatment of structures, and in some sense represent the core part of the more complex constraints. The advantage of using the simpler constraints is that they simplify some of the details of the algorithm and its presentation, without changing its essence.

We conclude this section with a formal definition of the constraints considered and a statement of what is involved in constructing the least model of such constraints. A *set expression* (denoted $sexp$, possibly subscripted) is either a set variable (denoted $\mathcal{X}, \mathcal{Y}, \cdots$), the empty set symbol $\perp$, the universal set symbol $\top$, a constant, the union of two set expresions, the intersection of two set expression, or of one of the forms $f(sexp_1, \cdots, sexp_n)$ or $f_{(i)}^{-1}(sexp)$ where f is some n-ary function symbol, $1 \leq i \leq n$ and $sexp, sexp_1, \cdots, sexp_n$ are set expressions. The set constraints considered in this paper have the form $\mathcal{X}_1 = sexp_1, \cdots, \mathcal{X}_m = sexp_m$, where $\mathcal{X}_1, \cdots, \mathcal{X}_m$ are distinct set variables, and the $sexp_i$ are set expressions. An *interpretation* $\mathcal{I}$ of these constraints is a mapping from the set variables in the constraints into sets of terms. Such a mapping can be extended to map set expressions into sets of terms in an obvious way, for example $\mathcal{I}(f(sexp_1, \cdots, sexp_n) \stackrel{\text{def}}{=} \{f(x_1, \cdots, x_n) : x_i \in \mathcal{I}(sexp_i)\}$, $\mathcal{I}(f_{(i)}^{-1})(sexp) \stackrel{\text{def}}{=} \{x_i : f(x_1, \cdots, x_n) \in \mathcal{I}(sexp)\}$, and $\mathcal{I}(sexp \cap sexp') \stackrel{\text{def}}{=} \mathcal{I}(sexp) \cap \mathcal{I}(sexp')$. An interpretation $\mathcal{I}$ of the constraints $\mathcal{X}_1 = sexp_1, \cdots, \mathcal{X}_m = sexp_m$ is a *model* if $\mathcal{I}(\mathcal{X}_i) = \mathcal{I}(sexp_i)$ for each $1 \leq i \leq m$. Models are ordered as follows: $\mathcal{I} \leq \mathcal{I}'$ iff $\mathcal{I}(\mathcal{X}) \subseteq \mathcal{I}(\mathcal{X})$ for all relevant set variables $\mathcal{X}$.

For the purposes of set based analysis, we shall only be interested in the least models of set constraints, and these always exist for the constraints considered here. Moreover, [4] shows that least models are always describable by regular term grammars because the set defined for each set variable is definable using a regular term grammar or equivalently a frontier-to-root tree automaton (see [2]). Computing the least model of a collection of set constraints involves constructing an explicit representation of the least model using regular tree grammars. This representation is explicit in the sense that its structure is evident and questions about membership and emptiness can be readily answered. More precisely, when input a collection of set constraints, the algorithm implemented in this paper outputs a collection of *explicit form* set constraints, whose least model is equivalent to that of the input constraints. To define explicit form constraints, first define that a set expression is *atomic* if it is $\top$, a set variable, or of the form $f(s_1, \cdots, s_n)$ where each s_i is an atomic set expression. Then, an explicit form collection of set constrains is of the form $\mathcal{X}_1 = sexp_1, \cdots, \mathcal{X}_m = sexp_m$ where each $\mathcal{X}_i$ is a distinct set variable, and each $sexp_i$ consists of a (possibly empty) union of non-variable atomic set expressions. Explicit form set constraints are essentially a specification of a regular term grammar for each variable appearing in the constraints.

3 The Basic Set Constraint Algorithm

We begin by presenting an algorithm for computing the least model of a collection of constraints $\mathcal{X}_1 = sexp_1, \cdots, \mathcal{X}_m = sexp_m$. This algorithm is closely related to the algorithm in [4] (see also [3]); we shall discuss the differnces at the end of this section. For convenence of presentation, we shall ignore the symbol $\top$. The details of the transformations to deal with $\top$ are straightforward and can be found in [4]. The first step of the algorithm is a preprocessing stage which puts the constraints in a standard form, and this essentially involves removing nested occurrences of union, intersection and projection symbols. Specifically, an *intersector* is of the form $s_1 \cap \cdots \cap s_n$ where the s_i are atomic set expressions. A *projector* is of the form $f_{(i)}^{-1}(s)$ where s is an atomic set expression. A constraint is in *standard form* if it is of the form $\mathcal{X} = sexp_1 \cup \cdots \cup sexp_n$ where at most one of the $sexp_i$ is either a projector or an intersector, and the remaining $sexp_i$ are atomic expressions. A collection of constraints $\mathcal{X}_1 = sexp_1, \cdots, \mathcal{X}_m = sexp_m$ is in *standard form* if each $\mathcal{X}_i$ is a distinct set variable, and each constraint is in standard form.

To put a collection of equations $\mathcal{X}_1 = sexp_1, \cdots, \mathcal{X}_m = sexp_m$ into standard form, it suffices to repeatedly identify an occurrence of a set expression which does not satisfy the standard form definition, replace it by a new variable and then add a new equation between the new variable and the replaced set expression. For example, consider the equation $\mathcal{X} = (\mathcal{Y} \cup \mathcal{Z}) \cap p_{(1)}^{-1}(p_{(2)}^{-1}(\mathcal{Y}))$. This is not in standard form for three reasons: $\mathcal{Y} \cup \mathcal{Z}$ and $p_{(1)}^{-1}(p_{(2)}^{-1}(\mathcal{Y}))$ are both nested inside a $\cap$ symbol and $p_{(2)}^{-1}(\mathcal{Y})$ is nested inside a projection symbol. Three replacement steps lead to the equations

$$\begin{array}{rcl} \mathcal{X} & = & \mathcal{W}_1 \cap \mathcal{W}_2, \\ \mathcal{W}_1 & = & \mathcal{Y} \cup \mathcal{Z}, \\ \mathcal{W}_2 & = & p_{(2)}^{-1}(\mathcal{W}_3), \\ \mathcal{W}_3 & = & p_{(2)}^{-1}(\mathcal{Y}). \end{array}$$

The details of this are straightforward and can be adapted from [4]. Importantly the resulting equations are equivalent to the original equations in the sense that the least model of the former, when restricted to the variables in the original equations, is the least model of the latter.

The core part of the algorithm consists of the exhaustive application of three transformations to constraints $\mathcal{X}_1 = rhs_1, \cdots, \mathcal{X}_m = rhs_m$ in standard form. Notationally we shall write $rhs(\mathcal{X})$ to denote that the right hand side of the equation for $\mathcal{X}$, and we shall treat rhs as an updatable array. For convenience we shall extend the definition of rhs and write $rhs(s)$ to denote $\{s\}$ in the case where s is not a variable (however we shall never assign to $rhs(s)$ unless s is a variable). If $rhs(\mathcal{X})$ is of the form $conj_1 \cup \cdots \cup conj_n$, we shall write $sexp \in rhs(\mathcal{X})$ to indicate that $sexp$ is one of the expressions $conj_1, \cdots, conj_n$. We now describe the first two of the three transformation steps.

Transformation 1 (Variable) *If $\mathcal{Y} \in rhs(\mathcal{X})$ and there is an atomic expression $s \in rhs(\mathcal{Y})$ such that $s \notin rhs(\mathcal{X})$, then update $rhs(\mathcal{X})$ to be the set expression $rhs(\mathcal{X}) \cup s$.* $\square$

Transformation 2 (Projector) *If $f_{(i)}^{-1}(s) \in rhs(\mathcal{X})$, $t \in rhs(s)$, t is nonempty and of the form $f(t_1, \cdots, t_n)$, then if $t_i \notin rhs(\mathcal{X})$ then update $rhs(\mathcal{X})$ to be the set expression $rhs(\mathcal{X}) \cup t_i$.* $\square$

The nonempty condition in the projector transformation is needed to deal with constraints such as $\mathcal{X} = f_{(1)}^{-1}(f(a, \mathcal{U}))$, $\mathcal{U} = a \cap b$. The least model of this constraint maps both $\mathcal{X}$ and $\mathcal{U}$ into the empty set. Now, if the projector transformation is applied to this constraint, it would lead to the constraint $\mathcal{X} = a \cup f_{(1)}^{-1}(f(a, \mathcal{U}))$ whose least model now maps $\mathcal{X}$ into $\{a\}$. To determine whether an atomic expression is empty or not at some stage in the algorithm, we proceed as follows. We associate a boolean value $nonempty(\mathcal{X})$ with each variable $\mathcal{X}$ which 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 t is an atomic expression appearing in $rhs(\mathcal{X})$

such that *nonempty*($\mathcal{Y}$) is *true* for all variables $\mathcal{Y}$ appearing in t, then update *nonempty*($\mathcal{X}$) to *true*. On termination of these updating steps, an atomic expression t is determined to be empty iff all variables that appear in the term are nonempty.

Before giving the last transformation, which deals with simplifying intersection, we need some preliminaries. The main difficulty with intersection is that we need to introduce new variables, and this introduction must be carefully controlled. We introduce a special naming scheme: each introduced variable is of the form $\mathcal{V}_{\{t_1,\cdots,t_n\}}$, $n \geq 1$, where $t_1, \cdots, t_n$ are atomic expressions appearing in equations at the start of the algorithm. The intention here is that such a variable, called a *composite variable*, is equivalent to the expression $t_1 \cap \cdots \cap t_n$. 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, let $\mathcal{N}$ denote the mapping from an atomic expression to the set of atomic expressions it represents:

$$\mathcal{N}(t) = \begin{cases} T & \text{if } t \text{ is } \mathcal{V}_T; \\ \{t\} & \text{otherwise.} \end{cases}$$

For example, $\mathcal{N}(\mathcal{V}_{\{f(\mathcal{X}),f(\mathcal{Z})\}} \cap \mathcal{V}_{\{f(\mathcal{X}),\mathcal{Y}\}}) = \{f(\mathcal{X}), \mathcal{Y}, f(\mathcal{Z})\}$. Similarly $\mathcal{N}(\mathcal{V}_{\{f(\mathcal{X}),\mathcal{Y}\}} \cap f(\mathcal{Z}))$ and $\mathcal{N}(f(\mathcal{X}) \cap \mathcal{Y} \cap f(\mathcal{Z}))$ are also equal to $\{f(\mathcal{X}), \mathcal{Y}, f(\mathcal{Z})\}$. Finally, we can state

Transformation 3 (Intersector) *If $s_1 \cap \cdots \cap s_m \in rhs(\mathcal{X})$ and $f(t_{i,1}, \cdots, t_{i,n}) \in rhs(s_i)$*[2] *for $1 \leq i \leq m$, then let s be $f(\mathcal{V}_{S_1}, \cdots, \mathcal{V}_{S_n})$, where S_j is $\mathcal{N}(t_{1,j}) \cup \cdots \cup \mathcal{N}(t_{m,j})$, $1 \leq j \leq n$. If $s \notin rhs(\mathcal{X})$ then update $rhs(\mathcal{X})$ to be the expression $rhs(\mathcal{X}) \cup s$. For each j, $1 \leq j \leq n$, if $rhs(\mathcal{V}_{S_j})$ is not defined, then update $rhs(\mathcal{V}_{S_j})$ to be $t_{1,j} \cap \cdots \cap t_{m,j}$.* □

The algorithm can now be stated as: exhaustively apply transformations 1 – 3, and on termination output the explicit form constraints resulting from the deletion of all intersectors, constructors and variables from *rhs*.

This formulation of the set constraint algorithm is more appropriate for implementation purposes in that it attempts to minimize the number of transformations that have to be performed. The main difference is that the projector and intersector transformations also contain part of the previous substitution transformation. In essence, the variable transformation is all that remains of the original substitution transformation. A minor difference is that [4] used an extended version of regular grammars, whereas we deal directly with set equations. We now briefly outline the proof of correctness of this formulation of the algorithm, with particular emphasis on the parts of the proof that differ from that in [4].

Theorem 1 *If constraints $\mathcal{C}_{in}$ are input to the algorithm then the algorithm terminates and output constraints $\mathcal{C}_{out}$ in explicit form. Moreover $\mathcal{I}_{in}(\mathcal{X}) = \mathcal{I}_{out}(\mathcal{X})$ for all $\mathcal{X}$ which appears in $\mathcal{C}_{in}$, where $\mathcal{I}_{in}$ and $\mathcal{I}_{out}$ are the least models of $\mathcal{C}_{in}$ and $\mathcal{C}_{out}$ respectively.*

Proof Outline: First, it is clear that there are only a finite number of variables $\mathcal{V}_S$ added by the algorithm since, for each such variable, S must be a subset of the atomic expressions contained in the input collection $\mathcal{C}_{in}$. Now, each atomic expression considered by the algorithm must either be an atomic expression from $\mathcal{C}_{in}$, or an expression of the form $f(\mathcal{V}_{S_1}, \cdots, \mathcal{V}_{S_n})$, hence there are only a finite number of such expressions. Termination of the algorithm then follows from the fact that each productive transformation adds an atomic expression to the *rhs* of some variable, and such additions are never repeated.

[2] Recall that if for some i, s_i is not a variable, then $f(t_{i,1}, \cdots, t_{i,n}) \in rhs(s_i)$ means $f(t_{i,1}, \cdots, t_{i,n}) = s_i$.

We now address correctness. Let $\mathcal{C}'$ be the constraints obtained after the exhaustive application of the transformations and before the deletion step. It is clear that the transformations preserve the least model of the input constraints (although these transformation may introduce new variables), and so it follows that the least model of $\mathcal{C}'$ is equivalent to $\mathcal{I}_{in}$ when restricted to the variables in $\mathcal{C}_{in}$. It remains to show that the least model of $\mathcal{C}'$ is equal to the least model of $\mathcal{C}_{out}$. This follows from that fact that $\mathcal{I}_{out}$ is a model of $\mathcal{C}'$. To prove this fact, consider an equation in $\mathcal{C}' - \mathcal{C}_{out}$. Such an equation must be of the form $\mathcal{X} = s_1 \cup \cdots s_n \cup op$ where $X = s_1 \cup \cdots \cup s_n$ appears in $\mathcal{C}_{out}$ and op is either a projector or intersector. We prove the former case; the latter is similar. By definition $\mathcal{I}_{out}(\mathcal{X}) = \mathcal{I}_{out}(s_1 \cup \cdots \cup s_n)$, and so it suffices to show that $\mathcal{I}_{out}(f^{-1}_{(i)}(t)) \subseteq \mathcal{I}_{out}(s_1 \cup \cdots \cup s_n)$, where op is $f^{-1}_{(i)}(t)$. Let $x_i \in \mathcal{I}_{out}(f^{-1}_{(i)}(t))$. By definition, there must exist a term $f(x_1, \cdots, x_n) \in \mathcal{I}_{out}(t)$. Now, t is either a variable or a non-variable atomic expression. If it is a variable, say $\mathcal{X}$, then we can show there must exist an atomic expression $f(t_1, \cdots, t_m)$ in the rhs of the equation for X in $\mathcal{C}'$ such that $x_j \in t_j$, $1 \le j \le n$. Now, the projector transformation, when applied to $f^{-1}_{(i)}(t)$ and $f(t_1, \cdots, t_n)$ would add t_i to $rhs(\mathcal{X})$, and so t_i is in fact one of $s_1, \cdots, s_n$. Hence $x_i \in t_i \subseteq \mathcal{I}_{out}(s_1 \cup \cdots \cup s_n)$. The case where t is a non-variable atomic expression is similar. $\square$

We conclude this section with a brief example illustrating the execution of the algorithm.

$$
\begin{array}{rcl}
\mathcal{W} & = & \mathcal{X} \cup f^2(\mathcal{W}) \cup f^{-1}(\mathcal{W}), \\
\mathcal{X} & = & f(\mathcal{Y}) \cap \mathcal{Z}, \\
\mathcal{Y} & = & a, \\
\mathcal{Z} & = & f(a) \cup g(a),
\end{array}
$$

A. Add $f(\mathcal{W})$ to $rhs(\mathcal{W})$;
B. Add $f(\mathcal{V}_{\{\mathcal{Y},a\}})$ to $rhs(\mathcal{X})$, and add the equation $\mathcal{V}_{\{\mathcal{Y},a\}} = \mathcal{Y} \cap a$;
C. Add $f(\mathcal{V}_{\{\mathcal{Y},a\}})$ to $rhs(\mathcal{W})$;
D. Add a to $rhs(\mathcal{V}_{\{\mathcal{Y},a\}})$;

When input with the above constraints, the algorithm performs steps A, B, C and D, corresponding to applications of transformations 2, 3, 1 and 3 respectively. The output of the algorithm is $\mathcal{W} = f^2(\mathcal{W}) \cup f(\mathcal{W}) \cup f(\mathcal{V}_{\{\mathcal{Y},a\}})$, $\mathcal{X} = f(\mathcal{V}_{\{\mathcal{Y},a\}})$, $\mathcal{Y} = a$, $\mathcal{Z} = f(a) \cup g(a)$, and $\mathcal{V}_{\{\mathcal{Y},a\}} = a$.

4 Outline of Implementation

The implementation of the preprocessing phase is fairly straightforward. We shall instead concentrate on the implementation of the core part of the algorithm. We first classify the kinds of expressions that can appear on the rhs of an equation. First, there are non-variable atomic expression, and we refer to these as *constructed* terms. In essence, these are the central objects which are manipulated by the algorithm. Second, there are the projectors, and these can be viewed as operators on constructors. For example, if $rhs(\mathcal{X})$ contains the constructed term $f(t_1, t_2)$, then $f^{-1}_{(2)}(\mathcal{X})$ produces the term t_2 (which must either be a constructed term or a variable). Specifically, corresponding to each projector of the form $f^{-1}_{(i)}(\cdots)$, we define a partial function $proj_{f,i}$ which maps $f(t_1, \cdots, t_n)$ into t_i and is undefined if t is not of the form $f(t_1, \cdots, t_n)$. The third class of expressions consists of the intersectors. Again these can be viewed as operators on constructors, and corresponding to each n-ary intersection expression, we define a partial function $intersect_n$ as follows: for each function symbol f, $intersect_n(f(t_{1,1}, \cdots, t_{1,n}), \cdots, f(t_{m,1}, \cdots, t_{m,n})) = f(\mathcal{V}_{S_1}, \cdots, \mathcal{V}_{S_n})$ where each S_j is $\mathcal{N}(s_{1,j}) \cup \cdots \cup \mathcal{N}(s_{m,j})$, and when applied to arguments which are not of this form, $intersect_n$ is undefined. The final kind of expressions that can occur in rhs are simply variables, and their main effect in the algorithm is to directly transfer constructed terms from one equation to another.

On the basis of this classification, we represent the array rhs as three separate arrays, con, var and op. Let $con(\mathcal{X})$ denote the set of construced terms in $rhs(\mathcal{X})$, let $var(\mathcal{X})$ denote the set of variables in $rhs(\mathcal{X})$ and let $op(\mathcal{X}_i)$ denote the set of projectors and intersectors in $rhs(\mathcal{X})$. For example, the equation $\mathcal{X} = f^2(\mathcal{X}) \cup f(\mathcal{X}) \cup f^{-1}(\mathcal{X}) \cup \mathcal{Y}$ is represented as $con(\mathcal{X}) = \{f^2(\mathcal{X}), f(\mathcal{X})\}$, $op(\mathcal{X}) = \{f^{-1}(\mathcal{X})\}$, and $var(\mathcal{X}) = \{\mathcal{Y}\}$. Note that the preprocessing phase ensures that initially $op(\mathcal{X}_i)$ is either empty or a singleton set, and that the algorithm never alters $op(\mathcal{X}_i)$. Hence we shall treat $op(\mathcal{X}_i)$ as either the empty set, or a projector or an intersector. Corresponding to the convention that $rhs(s)$ is just $\{s\}$ for any non-variable atomic expression s, we define that $con(s)$ is $\{s\}$. Now, the transformation steps of the algorithm can be rephrased as:

- If $op(\mathcal{X})$ is $f_i^{-1}(\mathcal{Y})$, $t \in con(\mathcal{Y})$ and $nonempty(t)$, then if $proj_{f,i}(t)$ is a variable, add it to $var(\mathcal{X})$, and otherwise add it to $con(\mathcal{X})$.
- If $op(\mathcal{X})$ is $s_1 \cap \cdots \cap s_m$ and $t_i \in con(s_i)$, then add $intersect_n(t_1, \cdots, t_m)$ to $con(\mathcal{X})$, and construct appropriate equations for any new composite variables introduced.
- If $\mathcal{Y} \in var(\mathcal{X})$ and $t \in con(\mathcal{Y})$, then add t to $con(\mathcal{X})$.

where "add" is used informally to mean that if the operation is defined, then add the resulting atomic expression to the specified set if it does not already appear there. Recall that the algorithm consists of the exhaustive applications of *productive* steps, which add something new to some set. Note that the sets $con(\mathcal{X})$, $var(\mathcal{X})$ are monotonically increasing and that $op(\mathcal{X})$ does not change.

Our implementation is essentially based on the above formulation of transformations. Two key observations about this formulation motivate a number of major implementation design decisions. First, a frequent operation in the algorithm involves comparison of constructed terms, and in particular, the determination of whether a new constructed term is already an element of a set. Second, at any particular instance, there may be many possible steps, but only a few of these are likely to be productive. It is therefore important to be able to focus on those steps which are likely to be productive. We now elaborate.

Term Representation

To provide the cheap comparison for constructed terms, we code each atomic term as an integer in the following simple manner. As each variable is encountered, a unique positive integer is allocated for it. Notationally we shall write $\#\mathcal{X}$ to denote the integer associated with the variable $\mathcal{X}$. Function and constant symbol are also identified by positive integers; we write $\#f$ to denote the integer for the function or constant symbol f. The coding of constructed terms is essentially performed by incrementally building a mapping ζ from sequences of integers into negative integers (ζ is initially empty). Specifically, the coding $\#t$ of a constructed term t is achieved as follows. Let t be $f(t_1, \cdots, t_n)$ and first compute the sequence $(\#f, \#t_1, \cdots, \#t_n)$. Then, return $\zeta(\#f, \#t_1, \cdots, \#t_n)$ if it is already defined, and otherwise allocate a negative integer say j, update ζ appropriately and then return j. An array is also maintained to map each coding into the sequence it represents. This is used particularly by the projection operation.

Coding constructed terms as integers provides a very compact representation of atomic terms, since there is maximal sharing between atomic terms which have common subterms. It also allows the set $con(\mathcal{X})$ to be represented efficiently. Specifically, we represent the relationship $t \in con(\mathcal{X})$ using ordered pairs $(\#t, \#\mathcal{X})$, and this in turn is implemented using a hash table. Similar comments apply to a number of other operations of the algorithm, and hashing techniques are heavily used throughout. Although fairly simple methods have proved effective for moderate sized problems, it is likely that specialized hashing techniques will be important for handling very large analysis problems.

However the use of the coding represents a tradeoff. Whilst operations such as term comparision are dramatically improved, the operation of projection becomes slightly more expensive and also there are overheads in initializing and maintaining the data structures used. Thus far, these costs appear to be insignificant compared to the substantial improvements over a very early implementation using an explicit representation of terms.

Dependency Directed Updating

We now address the issue of focussing on productive steps. Note that once a particular instance of a step has been applied, it never needs to be applied again. For example if $op(\mathcal{X})$ is the projector $f_{(1)}^{-1}(\mathcal{Y})$ and the constructed term $f(a,b)$ appears in $con(\mathcal{Y})$, then once a has been added to $con(\mathcal{X})$, we never again need to apply $op(\mathcal{X})$ to $f(a,b)$. We exploit this by using a dependency directed updating scheme. In other words we only apply operations to newly constructed terms. Specifically, we define an array dep such that $dep(\mathcal{X})$ is the set consisting of

$$\begin{array}{ll} & \{var\_dep(\mathcal{Y}) : \mathcal{X} \in var(\mathcal{Y})\} \\ \bigcup & \{proj\_dep(\mathcal{Y}) : f_{(i)}^{-1}(\mathcal{X}) \in var(\mathcal{Y})\} \\ \bigcup & \{\cap\_dep(\mathcal{Y}) : (s_1 \cap \cdots \cap s_n) \in var(\mathcal{Y}) \text{ where some } s_i \text{ is } \mathcal{X}\ \} \end{array}$$

We refer to items of the form $var\_dep(\mathcal{Y})$, $proj\_dep(\mathcal{Y})$ or $\cap\_dep(\mathcal{Y})$ as *dependencies*. The set of dependencies for a variable $\mathcal{X}$ indicate where any new constructed terms for $\mathcal{X}$ have to be propagated. For example, consider the following constraints and their representation in terms of arrays *con*, *var op* and *dep*.

$$\begin{array}{lcl} \mathcal{X} & = & b \cup f(\mathcal{X}) \cup \mathcal{Z} \\ \mathcal{Y} & = & c \cup (\mathcal{X} \cap \mathcal{Z}) \\ \mathcal{Z} & = & d \cup f_{(1)}^{-1}(\mathcal{X}) \end{array}$$

	con	*var*	*op*	*dep*
$\mathcal{X}$	$b, f(\mathcal{X})$	$\mathcal{Z}$		$\cap\_dep(\mathcal{Y}), proj\_dep(\mathcal{Z})$
$\mathcal{Y}$	c		$\mathcal{X} \cap \mathcal{Z}$	
$\mathcal{Z}$	d		$f_{(1)}^{-1}(\mathcal{X})$	$var\_dep(\mathcal{X}), \cap\_dep(\mathcal{Y})$

Using this representation, the algorithm can now be described as

```
update(t, dep, X) {
   if dep is var_dep(Y) then add(t, Y);
   if dep is proj_dep(Y) and op(Y) is f^{-1}_{(i)}(X) then add(proj_{f,i}(t), Y);
   if dep is ∩_dep(Y) and op(Y) is s_1 ∩ ··· ∩ s_m then
      let X be s_j;     /* note that X must appear in s_1 ···, s_m */
      foreach sequence (t_1, ···, t_n) such that t_i ∈ rhs(s_i), j ≠ i, and t_i = t
         add(intersect_n(t_1, ···, t_n), X);
         construct appropriate equations for any new composite variables;
}

add(t, X) {
   if t ∉ con(X) then
      con(X) := con(X) ∪ {t};
      foreach dep' ∈ dep(X)
         update(t, dep', X);
}
```

The algorithm is initiated by inspecting each variable $\mathcal{X}$ and calling $add(proj_{f,i}(t), \mathcal{X})$ for each

$t \in rhs(s)$ if $op(\mathcal{X})$ is $f_{(i)}^{-1}(s)$, and calling $add(intersect_n(t_1, \cdots, t_n))$ for each sequence $(t_1, \cdots, t_n)$ such that $t_i \in rhs(s_i)$, if $op(\mathcal{X})$ is $s_1 \cap \cdots \cap s_n$.

Composite Variables and Nonempty

We conclude by discussing composite variables and the function *nonempty*. The generation and management of composite variables is essentially straightforward. Corresponding to the function $\mathcal{N}$, an array is used to record, for each new composite variable allocated, the set of terms corresponding to this variable. The mapping from a set of terms $\{s_1, \cdots, s_n\}$ into a composite variable is performed by first constructing $\mathcal{N}(s_1) \cup \cdots \cup \mathcal{N}(s_n)$. The elements of this set are then sorted and the resulting sequence is looked up in a hash table, where each entry of this table consists of a sorted list of atomic expressions and the corresponding composite variable identifier. If the sequence is found in the table, the appropriate variable identifier is returned. If the sequence is not found, then a new variable $\mathcal{V}_S$ is generated and the table is updated appropriately. Also a new equation is generated by setting $var(\mathcal{V}_S) := \emptyset$, $con(\mathcal{V}_S) := \emptyset$ and $op(\mathcal{V}_S) := \{s_1 \cap \cdots \cap s_n\}$ where S is $\{s_1, \cdots, s_n\}$, and updating the dependencies for each variable $\mathcal{X}$ in S using $dep(\mathcal{X}) := dep(\mathcal{X}) \cup \{op\_dep(\mathcal{V}_S)\}$, and then finally, if each s_i is a non-variable atomic expression, calling $add(intersect_n(s_1, \cdots, s_n), \mathcal{V}_S)$

We now address the issue of the nonempty condition in the projector step. We first describe the implementation of the *nonempty* function. Clearly we do not want to recompute *nonempty* each time it is needed. Instead *nonempty* may be incrementally computed as follows. Let *nonempty* be an array which maps each atomic expression t to a boolean value which is *true* if t is currently known to be nonempty. Initially each entry in *nonempty* is set to *false*. The value of $nonempty(f(t_1, \cdots, t_n))$ is updated to *true* if $nonempty(t_1)$ is *true* for all $1 \leq i \leq n$. The value of $nonempty(\mathcal{X})$ for a variable $\mathcal{X}$ is updated to *true* if $con(\mathcal{X})$ contains a nonempty atomic expression. Again this updating is managed using lists of dependencies. In this case, for each atomic expression t, we maintain the list of the atomic expressions directly dependent on $nonempty(t)$. When $nonempty(t)$ changes from *false* to *true*, the terms dependent on t are recomputed.

We now address the issue of how to incorporate the nonempty condition in the projector updating step. Recall that the step is: if $op(\mathcal{X})$ is $\Pi_{\mathcal{Y}}$, $t \in con(\mathcal{Y})$ and $nonempty(t)$, then add $\Pi_{\mathcal{Y}}(t)$ to $con(\mathcal{X})$ or $var(\mathcal{X})$. Now, there is no difficulty if we find that $nonempty(t)$ is *true*, since we can then proceed with the updating step. However, if $nonempty(t)$ is *false* then we need to ensure that if $nonempty(t)$ ever becomes *true*, then the projector updating step is eventually performed. This is achieved by adding a new kind of dependency to the nonempty dependency lists.

5 Intersectors

The previous section outlined a rudimentary implementation, which although a significant improvement over a naive implementation, is still impractical. The main reason for this is intersectors, which are perhaps the most problematic part of the implementation. Not only are they responsible for introducing new variables, but updating intersectors has a combinatorial nature. For example consider an intersector $t_1 \cap \cdots \cap t_n$, and suppose that s has been freshly added to $con(t_i)$. Then, in updating this intersector using this constructed term, we must consider all possible combinations of $s_j \in con(t_j)$, $j \neq i$.

A first step in dealing with this problem is to partition the terms in $con(\mathcal{X})$ according to their principal function symbol. For example, consider updating the intersector $\mathcal{X} \cap \mathcal{Y} \cap \mathcal{Z}$ using a newly constructed term $f(\mathcal{Y})$ for $\mathcal{Z}$. If $con(\mathcal{X}) = \{f(\mathcal{X}), f(\mathcal{W}), g(\mathcal{X})\}$ and $con(\mathcal{Y}) = \{f(\mathcal{Z}), g(\mathcal{Y})\}$, then clearly we only need to consider the intersections $f(\mathcal{X}) \cap f(\mathcal{Z}) \cap f(\mathcal{Y})$ and $f(\mathcal{W}) \cap f(\mathcal{Z}) \cap f(\mathcal{Y})$, and we can ignore intersections such as $f(\mathcal{X}) \cap g(\mathcal{Y}) \cap f(\mathcal{Y})$, since they will always be empty. Although this simple modification is extremely important, more fundamental changes are required to implement intersection efficiently. In essence we need to exploit the substantial redundancy which is typically present in set constraints.

Minimal DNF Expansion

Consider the intersector $\mathcal{X} \cap \mathcal{Y} \cap \mathcal{Z}$ where $con(\mathcal{X})$, $con(\mathcal{Y})$ and $con(\mathcal{Z})$ are respectively $\{A, B, C\}, \{A, C, D\}$ and $\{A, B, D\}$, where A, B, C and D are distinct constructed terms. In essence, we wish to compute

$$(A \cup B \cup D) \cap (A \cup C \cup D) \cap (A \cup B \cup D).$$

A naive expansion of this expression would lead to 27 intersections: $A \cap A \cap A$, $A \cap A \cap C$, etc. However for this example, we only need to compute the expressions A, $B \cap D$, $C \cap B$ and $C \cap D$, and so we can reduce 27 intersections to just three. The problem of minimizing the number of expressions which must be computed can be viewed as a special case of computing a minimal DNF form, given an expression in CNF. A number of different approaches were tested. The essence of the current approach is to precompute information about the pattern of occurrences of constructed terms which appear in more than one "disjunction", and then to use this information to sequentially build up minimal conjunctions. For space reasons we omit the details of this algorithm.

We now outline how this approach to intersection is incorporated. It is clear that updating intersectors is very expensive, and should therefore be done as infrequently as possible. This can be achieved by giving precedence to updates involving projectors. Specifically, the updating is split into two phases. The first consists of the exhaustive updating using projectors. The second consists of updating using intersectors. The algorithm then proceeds by repeatedly alternating these two phases. As a side effect, this organization of updating steps also enhances the performance of the DNF minimization algorithm, since by delaying its application, the amount of redundancy increases. The algorithm can now be described as

```
repeat
   exhaustively apply all possible projector steps;
   recompute each intersector;
until no change;
```

Approximating Subsumption Relationships

We conclude this section by describing an enhancement the performance of intersector recomputation. This modification is perhaps the most important described so far. Consider an intersector $\mathcal{X} \cap \mathcal{Y}$. If $con(\mathcal{X}) = \{A, B\}$ and $con(\mathcal{Y}) = \{C, D\}$ and in the least model of the equations, $A \subseteq B$ and $C \subseteq D$, then clearly the recomputation of the intersector can be reduced to computing $B \cap D$. However, establishing whether subsumption relationships such as $A \subseteq B$ hold in the least model can only be done, in general, once the least model is known. The approach used here involves approximating the least model using the information contained in the array *con*. Specifically, this array defines an equation for each variable $\mathcal{X}$:

$\mathcal{X} = t_1 \cup \cdots \cup t_n$, where $con(\mathcal{X}) = \{t_1, \cdots, t_n\}$.

Hence *con* can be considered to define a collection of equations in explicit form, and this in turn defines an interpretation, call it $\mathcal{I}_{con}$. This interpretation repesents that part of the least model which has been computed by the algorithm so far. If $\mathcal{I}_{lm}$ denotes the least model of the constraints input to the algorithm, then $\mathcal{I}_{con}(\mathcal{X}) \subseteq \mathcal{I}_{lm}(\mathcal{X})$ for all variables $\mathcal{X}$ appearing in the input constraints. Now, we can use $\mathcal{I}_{con}$ to approximate the subsumption relationships which hold in $\mathcal{I}_{lm}$, and hence simplify the intersector computation, as follows.

> **repeat**
>   **exhaustively apply** `all possble projector steps;`
>   **for** `each variable` $\mathcal{X}$
> $max\_con(\mathcal{X}) := \{t \in con(\mathcal{X}) : \neg\exists s \in con(\mathcal{X})$ `s.t.` $\mathcal{I}_{con}(s) \supset \mathcal{I}_{con}(t)\}^a$`;`
>   **recompute** `each intersector using` *max_con* `instead of` *con*`;`
> **until** `no change;`

[a]If this set contains terms which are equal under $\mathcal{I}_{con}$, then we pick one of them and delete the others.

Note that there is no formal connection between the subsumption relationships that hold in $\mathcal{I}_{lm}$ and $\mathcal{I}_{con}$. Hence, in general, $max\_con(\mathcal{X})$ is just a subset of $con(\mathcal{X})$ for each $\mathcal{X}$. In practice the correlation between the subsumption relationships which hold in $\mathcal{I}_{lm}$ and $\mathcal{I}_{con}$ is close, especially after the early stages of the algorithm (this issue is discussed further in the next section). The correctness of using *max_con* instead of *con* follows from the fact that $\mathcal{I}_{con} = \mathcal{I}_{max\_con}$, where $\mathcal{I}_{max\_con}$ is the model defined by *max_con*. We omit the details.

6 Conclusion

The implementation described in this paper has been developed over a period of three years. A very early version used an explicit representation for constructed terms. However term comparision was prohibitively expensive and only very small programs such as naive reverse could be analysed. The next version employed the term representation described here and exploited redundancy using a dnf minimization algorithm and an early form of the subsumption approximation algorithm. Most of the basic notions described in this paper were contained in this version. A subsequent version reimplemented these notions more efficiently. Greater use was made of dependency directed update, and the subsumption algorithm was completely redesigned using bit vectors. This reimplementation resulted in average speedups of at least a factor of 10, and a reduction in the number of equations generated by a factor of 4. The preceeding sections contain only an outline of some of the major design decisions of this implementation. For space reasons we have ommitted many details, including a large number of small modifications which, in sum, make an important contribution to the system's performance.

We now present a number of empirical results, with particular emphasis on illustrating the importance of some of the design decisions outlined in the previous sections. The current implementation is approximately 4,500 lines of Standard ML (this includes programs to construct the set constraints corresponding to a program as well as the set constraint solver and its front end). All timings are on a Sun Sparc 1+ (24MB) running Mach and using version 0.75 of Standard ML of New Jersey. Table 1 presents some execution statistics for a number of example set constraints. The nrev program is the standard naive reverse program. The dnf program is a program to con-

		time				space				
		total (secs)	proj (%)	sub (%)	$\cap$ (%)	input eqns	ouput eqns	intersectors	con terms	model size
nrev	bottom-up	0.01	0	100	10	10	10	0 + 0	11	10
	top-down l-r	0.21	10	76	14	20	35	8 + 5	20	26
dnf	bottom-up	0.65	29	68	3	88	96	3 + 4	82	159
	top-down l-r	12.59	6	59	35	156	440	53 + 254	313	1739
lcm constraints		13.86	0	99	1	3	145	3 + 142	171	148
hcf constraints		27.54	1	99	0	1	34	0 + 0	148	4

Table 1: Time and space measurements for six set constraint benchmarks.

		subsumption				no subsumption			
		time	eqns	terms	mem	time	eqns	terms	mem
nrev	bottom-up	0.01	10	11	16	< 0.01	10	11	16
	top-down l-r	0.21	35	20	75	0.11	35	20	75
dnf	bottom-up	0.65	96	82	1689	1.01	184	126	5309
	top-down l-r	12.59	440	313	9317	≈ 218	2669	1510	280K
lcm constraints		13.86	145	171	292	0.26	145	171	292
hcf constraints		27.54	34	148	1644	0.31	34	148	1644

Table 2: Effects of the subsumption component of the algorithm.

vert a propositional logic formula into a formula in disjunctive normal form. It contains 10 facts and 22 rules (with an average of about 2 body atoms per rule) and contains a large number of mutually recursive calls. For both of these programs we consider the set constraints corresponding both bottom-up and top-down left-to-right. The final two example constraints, lcm and hcf, are essentially pathological cases designed for testing the intersector and projector parts of the algorithm respectively. These constraints are $\mathcal{X} = a \cup f^{11}(\mathcal{X}), \mathcal{Y} = a \cup f^{13}(\mathcal{Y}), \mathcal{Z} = \mathcal{X} \cap \mathcal{Y}$ and $\mathcal{X} = a \cup f^{-33}(\mathcal{X}) \cup f^{144}(\mathcal{X})$ respectively, where $f^{11}(\mathcal{X})$ denotes 11 applications of f to $\mathcal{X}$ and $f^{-33}(\mathcal{X})$ denotes 33 applications of the projector $f^{-1}_{(1)}$ to $\mathcal{X}$. For each example, we give timing information, equation counts and an indication of the space used. The first column of the table is total time, and the next three columns break this down into time spent on projectors, inferring subsumption relationships and intersectors, each expressed as a percentage of total time. Columns 5, 6 and 7 respectively give the number of equations input to the algorithm, the total equations generated during execution, and the number of equations which deal with intersections (entries in this column are of the form $x + y$ where x is the number of intersectors in the input equations, and y is the number of intersectors introduced during execution). Columns 8 is the total number of distinct constructed terms. Column 9 provides a measure of the complexity of the least model constructed by the algorithm, and is obtained as follows. First, list all of the sets $max\_con(\mathcal{X})$, for each set varaible $\mathcal{X}$. Some of these sets will be the same (in which case the corresponding variables are equal in the least model). The complexity measure is then obtained by summing the cardinalities of the distinct sets which appear in this list. These results show that bottom-up constraints are substantially easier to solve than top-down left-to-right constraints. This general relationship holds because, by their nature, top-down constraints are more complicated and more accurate than the bottom-up constraints. The subsumption approximation part of the algorithm accounts for the majority of execution time. The lcm and hcf examples show extreme behaviour – in fact the subsumption approximation component of the algorthm is not used at all during the execution of the main body of the algorithm, although it is responsible for producing a compact representation of the output constraints. In the case of lcm, the output for $\mathcal{X}$ is $\mathcal{X} = a \cup f^{143}(\mathcal{X})$, and for hcf the output is $\mathcal{X} = a \cup f^{3}(\mathcal{X})$.

	Iterations								
	1	2	3	4	5	6	7	8	9
% time	2	8	18	31	47	61	81	91	100
% $\cap$ eqns	19	24	40	67	85	96	100	100	100
% model	23	43	58	68	78	86	93	100	100

Table 3: Cumulative consumption of resources during execution.

The driving example used during the development of this implementation was the top-down left-to-right constraints for the dnf program. A number of features of this program lead to top-down constraints which are unusually expensive to solve. In particular, it was this program that motivated the subsumption approximation part of the implementation. Table 2 illustrates the effects of the subsumption component of the algorithm on all of the example constraints. For the analysis of nrev, and for the lcm and hcf constraints, there is little redundancy and the subsumption component is expensive and provides no direct benefit. However for the more substantial dnf program, it is clearly of benefit, and in the case of top-down analysis, it is crucial. Specifically, in this case the difference in time is nearly a factor of 20, and for one measure of memory usage (the number of entries in the central hash table of the implementation), the difference is a factor of 30. (In fact the measurements for this entry in the table had to be made on a 64MB DECstation 5000, and then converted to equivalent Sparc 1+ times).

Table 3 illustrate the dynamic behaviour of the algorithm by giving cumulative measures of the consumption of resources during execution of the top-down dnf constraints. One "iteration" corresponds to an exhaustive application of the projector transformation followed by a recomputation of each intersector. The measure of the amount of the least model computed at each stage is obtained as follows. Let $max\_con(\mathcal{X})|_{final}$ denote the value of the set $max\_con(\mathcal{X})$ on termination of the algorithm. Let $\delta(\mathcal{X})$ denote the proportion of the elements in $max\_con(\mathcal{X})|_{final}$ which currently appear in $con(\mathcal{X})$, and this represents a pessimistic estimate of the proportion of $\mathcal{X}$ that has been currently computed. Finally, the measure of the amount of the least model computed is just the average of $\delta(\mathcal{X})$ over all variables $\mathcal{X}$ currently appearing in the equations. The behaviour of this measure indicates that the model grows quickly in the early stages, and that most of the time is spent obtaining the last few components of the model. This supports the notion that during most of the execution of the algorithm, $max\_con(\mathcal{X})$ provides a fairly good approximation to the least model for the purposes of obtaining subsumption relationships.

To illustrate the effects of the minimal dnf expansions algorithm, we again use the top-down dnf constraints. When applied to these constraints, the implementation constructs 1270 conjunctions. In comparison, a simple expansion of these formulas would have led to 1921 disjunctions. Although this only represents a direct saving of 649 disjunction, or about a third, the indirect savings, in terms of intersectors which never need to be constructed, is more significant.

In summary then, we have worked with a moderate sized program (dnf) which exhibits properties which are problematic for analysis such as substantial mutual recursion and intersection. Very significant progress has been made and this example can now be analysed within a reasonable time and space bound. We believe that this provides strong evidence that set based analysis can be made practical. However to realize this goal, much work remains. Our experience suggests that a number of components of the algorithm can be further improved. Currently the major expense is the subsumption algorithm, and one reason for this is that it is not by nature incremental. We are investigating ways of overcoming this, including trading some of its effectiveness for the ability to reuse results from previous iterations. Another avenue for improvement lies in the use of specialized hashing techniques.

Thus far we have focussed extensively on efficiency aspects of the analysis. However we have now reached a stage where moderate sized programs can be analyzed. We therefore plan to use this implemenetation to investigate set based analysis, with particular emphasis on the quality of the information obtained[3] and its relevance to compilation. Practical program analysis must also deal with operations such as call, assert and retract. Preliminary work suggests that it may be possible to directly modeled these operations in a set constraint framework. One extension that is in the current implementation, but which for space reasons has not been described, is a limited form of mode analysis. For example, when run on the nrev program where the initial goal has its first argument ground, our implementation computes that both arguments will be ground lists after goal execution, and that all calls to append are such that the first argument is always a ground list. Although this extension has been incorporated into the implementation with very little additional overhead, it is somewhat restricted because interdependencies between variables are ignored, and such dependencies can be important for mode analysis. An algorithm in [7] may provide a basis for more accurately dealing with modes whilst retaining the important advantages of structural accuracy.

We conclude with a discussion of related work. Most of the work on implementation of analysis for logic programs is based on abstract interpretation and the algorithms used are fundamentally different in nature. Our work is more closely connected to work on types for logic programs in which types are defined by ignoring inter-variable or inter-argument dependencies. Implementations have been reported in [11] and [14], but the former does not focus on practical issues, and the latter is not directly comparable to our work since it deals with type checking rather than type inference. The most closely related work to ours is [1], which describes an implementation of type inference for the functional language FL. In very general terms their observations about the complexity of the intersection operation are similar to ours. However the two algorithms are completely different in nature. In particular their algorithm does not include the projection operation, and this appears to substantially alter the tradeoffs and design decisions that are made. Furthermore, our implementation has been specifically designed to deal with constraints where there is substantial mutual dependency between variables. Such constraints are typical in the constraints generated for the top-down analysis of logic programs.

[3] As an example of the output of the algorithm, the top-down analysis of the dnf program constructs the following set $\mathcal{X}$ to describe the set of terms which are the result of putting an arbitrary input formula into dnf: $\mathcal{X} = or(\mathcal{X}, \mathcal{X}) \cup and(\mathcal{X}, \mathcal{X}) \cup not(\mathcal{C})$ where $\mathcal{C}$ describes the set of propositional constants.

References

[1] A. Aiken and B. Murphy, "Implementing Regular Trees", *Proceedings* 4^{th} *ACM Conference on Functional Programming and Computer Architecture*, pp 2 - 16, 1991.

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

[3] N. Heintze and J. Jaffar, "A Finite Presentation Theorem for Approximating Logic Programs", *Proceedings* 17^{th} *ACM-POPL*, 1990, pp. 197–209.

[4] N. Heintze and J. Jaffar, "A Decision Procedure for a Class of Herbrand Set Constraints", *Proceedings* 5^{th} *IEEE-LICS*, June 1990, pp 42 - 51.

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

[6] N. Heintze and J. Jaffar, "Semantic Types for Logic Programs" in *Types in Logic Programming*, F. Pfenning (Ed.), MIT Press Series in Logic Programming, 1992, forthcoming.

[7] N. Heintze and J. Jaffar, "An Engine for Logic Program Analysis", *Proceedings* 7^{th} *IEEE-LICS*, to appear.

[8] N.D. Jones and S.S. Muchnick, "Flow Analysis and Optimization of LISP-like Structures", *Proceedings* 6^{th} *ACM-POPL*, 1979, pp 244 - 256.

[9] N. Heintze, "Set Based Program Analysis", forthcoming Ph.D. thesis, School of Computer Science, Carnegie Mellon University, June 1992 (expected).

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

[11] C. Pyo and U.S. Reddy, "Inference of Polymorphic Types for Logic Programs" *Logic Programming, Proceedings of the North American Conference, 1989*, MIT Press, pp 1115 - 1134, October, 1989.

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

[13] T. Fruhwirth, E. Shapiro, M.Y. Vardi, E. Yardeni, "Logic Programs as Types for Logic Programs", *Proceedings* 6^{th} *IEEE Symposium on Logic in Computer Science*, June 1991.

[14] E. Yardeni and E.Y. Shapiro, "A Type System for Logic Programs", *Journal of Logic Programming*, Vol. 10, pp 125 - 153, 1991.