Set-Based Program Analysis

(Extended Abstract)

Nevin Heintze* and Joxan Jaffar†

1 January 1991

SUMMARY

Program analysis involves approximating the semantics of programs. One important kind of approximation, that of ignoring intervariable dependencies, is widely used for reasons of decidability and efficiency. This paper addresses program analysis wherein ignoring intervariable dependencies is the *only* kind of approximation used.

Starting with the collecting semantics of a program, it is shown that there is a natural formalization of this approximation in which program variables are associated with sets of values. Further, this formalization is applicable to both the imperative and logic programming paradigms. An algorithm is then presented which inputs an imperative or logic program and constructs an explicit representation of its approximate semantics. It therefore provides the *most accurate* analysis possible, given that intervariables dependencies are ignored during analysis.

*School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

†IBM Thomas J. Watson Research Center, PO Box 704, Yorktown Heights, NY 10598.

1 Introduction

In program analysis, the notion of program approximation is fundamental because it is typically infeasible or impossible to obtain exact information about a program. In general, an approximation has several components, the complexity and number of which determine the simplicity and accuracy of the analysis. For example, one component of an approximation may involve the underlying domain of computation (to reason about integers, one may reason about their signs but ignore their specific values). Another component may involve intervariable relationships, for example, intervariable dependencies may be ignored (to reason about two variables x and y, one may reason about the collection of values each variable may assume, but ignore the collection of tuples (x, y)). For decidability reasons, program analyzers must limit their reasoning about intervariable dependencies. For efficiency reasons, analyzers often completely omit reasoning about intervariable dependencies [11]. This is in addition to making other approximations. What has not been addressed is whether or not program analysis is decidable when ignoring intervariable dependencies is the *only* approximation used.

The starting point of this paper is the use of *environment constraints* to specify the exact collecting semantics of a program. Such a formalization is concise because it states just the needed relationships between program points, without resort to the intricate details within a particular operational model. It is then shown that the approximation of ignoring intervariable dependencies can be naturally formalized by a *set-based* interpretation of these constraints, that is, by interpreting variables over sets of values. More formally, it is shown that the approximate semantics of a program can be defined to be the least set-based model of its environment constraints.

The main result proves that incorporating just the notion of ignoring intervariable dependencies into program semantics results in a decidable approximation. This is presented in the form of an algorithm whose first part rewrites environment constraints into a simpler set-based calculus in such a way that set-based models are preserved. The second part solves the resulting constraints and outputs an explicit representation of the least model. Because this model characterizes the approximate semantics of the program at hand, the algorithm provides the most accurate analysis possible in the context of the approximation considered.

In what follows, both the imperative and logic programming paradigms are considered to emphasize the generality of the result.

1.1 Related Work

The early works [14], [10] and [12] use set formulas on imperative and functional programs to conservatively estimate variable values. In addition to the approximation of ignoring intervariable dependencies, these formulas omit other information (such as that contained in **if-then** conditions), and the algorithms do not accommodate the notion of intersection between set expressions. Consequently, it is not clear how to relate these formulas with an independent definition of approximate semantics. However, their algorithms are considerably simpler, though less accurate, than the one presented in this paper.

In logic programming, the notion of program approximation has been used to formalize types [13]. Here, superset approximations of a program's least model were sought. In [7], where the focus was program analysis and not types, the notion of set-based models was introduced, and it was shown that the resulting program approximation was both more accurate and decidable. The logic programming aspect of the present paper advances this work significantly. In [7] program approximation was defined in terms of the immediate consequence operator, and hence was limited to the approximation of bottom-up program execution. It is, for example, not applicable for the analysis of procedure calls in a top-down execution. The present paper, in contrast, is based upon a general formalism for collecting semantics, and hence can also be used for top-down execution strategies. Consequently the present algorithm is more general.

In abstract interpretation [1] approximation is typically formalized by means of a semantic function describing single step approximate computation. The least fixpoint gives the approximate semantics (see, for example, [4] and [5]), and so program analysis reduces to iterative fixpoint computation. Such an approach is not applicable in this paper because the appropriate semantic function does not converge. For example, while the our algorithm reasons exactly on the class of monadic logic programs, no abstract interpretation algorithm can. Note that abstract interpretation approaches typically ignore intervariable dependencies, in addition to making other approximations, for efficiency reasons (see for example [2, 3]).

1.2 Preliminary Definitions

The domain of computation is a Herbrand Universe with functors Σ hereafter denoted by small letters, and Σ^{-1} containing the *projections* of Σ: $\{f_{(i)}^{-1} : f$ is an n-ary symbol from Σ, $1 \leq i \leq n\}$. Program variables $\mathcal{V}$ are hereafter denoted by W, X, Y, Z. A *term* is constructed from $\mathcal{V}$, Σ and Σ^{-1}; a *simple term* is constructed from $\mathcal{V}$ and Σ only. A *ground* term or simple term contains no variables.

A *logic program* also uses predicate symbols Π, and it is defined to be a finite collection of *rules* of the form $H \leftarrow B_1, \cdots, B_n$, $n \geq 0$, where $H, B_1, \cdots, B_n$ are atoms built from simple terms and Π. H is called the *head* of the rule while each B_i is called a *body atom*. There are two further conditions: first, a program contains one special rule, called the *initial goal*, which has no head atom; second, no variable is common to two rules in a program. Let $head_P$ and $body_P$ respectively denote the head and body atoms in P.

An *imperative program* is a finite sequence $Stat^\star$ of statements $Stat$:

$$Stat \ ::= \ X := t \ \Big| \ \textbf{if}\ cond\ \textbf{then}\ Stat^\star \ \Big| \ \textbf{while}\ cond\ \textbf{do}\ Stat^\star$$

where t is a term, and *cond* (a *program condition*) ranges over boolean combinations of two basic relations: (a) $X = s$ where s is either a program variable or a ground simple term, and (b) $match_s(X)$ where s a template for a set of terms, that is, s is constructed from Σ and a new class of *template* variables. The relation $match_s(X)$ is *true* if there is a substitution θ on the template variables such that $\theta(s)$ is equal to the ground term bound to X. Intuitively, the basic operations of imperative programs are data construction and destruction, and the basic tests involve term matching.

An *environment*, hereafter denoted by ρ, is a mapping from $\mathcal{V}$ into ground simple terms. We write $\rho[X \mapsto t]$ to denote the environment which is identical to ρ except that it maps X to t. For example, if ρ maps X to $a.b.nil$[1] and Y to c, then execution of the imperative program $Z_1 := Y.X$; $Z_2 := cdr(X)$; $Z_3 := c.(car(X)).nil$ starting from ρ results in the environment $\rho[Z_1 \mapsto c.a.b.nil][Z_2 \mapsto b.nil][Z_3 \mapsto c.a.nil]$. In what follows, we use notation such as $\{(X \mapsto t_1, Y \mapsto t_2), (X \mapsto s_1, Y \mapsto s_2)\}$ as an abbreviation for the collection of environments $\{\rho : \rho(X) = t_1 \wedge \rho(Y) = t_2 \ \text{ or } \ \rho(X) = s_1 \wedge \rho(Y) = s_2\}$.

2 Collecting Semantics

The execution model of a programming language can be viewed as defining an execution graph for each program. Nodes in this graph represent execution states and edges represent transitions between states. A collecting semantics is obtained by finitely classifying the execution states, and then reasoning about the classes of states. Typically, each class of execution states corresponds to a unique part of the program syntax; hence the classes are called *program points*.

In imperative programs, program points can be defined by first labelling each statement occurrence S_i in the program, and then, for each such S_i, associating two program points $\uparrow i$ and $\downarrow i$ which indicate the execution states just before and just after S_i is executed. In logic programs, program points can be obtained by labelling each rule and each occurrence of a body atom. A rule label indicates the execution state just after the rule has finished executing[2]; a body atom label indicates the state just before execution of the body atom[3]. A collecting semantics for both imperative and logic programs associates to each program point a collection of environments.

In what follows we use positive integers for labels. For technical convenience, we

[1] $a.b.nil$, car and cdr are syntactic sugar for $cons(a, cons(b, nil))$, $cons^{-1}_{(1)}$ and $cons^{-1}_{(2)}$ respectively.

[2] That is, every body atom (and sub-goals thereof) has been solved.

[3] That is, just before the body atom is selected.

assume that the first statement of an imperative program has label 1. For logic programs, we also assume that the first rule of a logic program is the initial goal and has label 1. Consider the following imperative program IP and logic program LP:

1. $X := a.b.nil;$	
2. $Y := c;$	1. $\leftarrow loop(a.b.nil, Y_1)^{④}$.
3. **while** $(X \neq nil)$ **do**	2. $loop(Y_2.nil, Y_2).$
4. $\quad Y := car(X);$	3. $loop(Z.X, Y_3) \leftarrow loop(X, Y_3)^{⑤}$.
5. $\quad X := cdr(X);$	

Note that the label 3 corresponds to the entire **while-do** statement in IP. There are 10 program points for IP and 5 for LP. (Sometimes we will write labels n as ⓝ to avoid confusion with footnotes.) During execution of IP we have the following (in which we omit mention of the program points $\uparrow 1$, $\downarrow 1$, $\uparrow 2$ and $\downarrow 2$ for brevity):

	before
$\uparrow 3$	$\{(X \mapsto a.b.nil, Y \mapsto c)\}$
$\uparrow 4$	$\{(X \mapsto a.b.nil, Y \mapsto c), (X \mapsto b.nil, Y \mapsto a)\}$
$\uparrow 5$	$\{(X \mapsto a.b.nil, Y \mapsto a), (X \mapsto b.nil, Y \mapsto b)\}$

	after
$\downarrow 3$	$\{(X \mapsto nil, Y \mapsto b)\}$
$\downarrow 4$	$\{(X \mapsto a.b.nil, Y \mapsto a), (X \mapsto b.nil, Y \mapsto b)\}$
$\downarrow 5$	$\{(X \mapsto b.nil, Y \mapsto a), (X \mapsto nil, Y \mapsto b)\}$

During execution of LP we have the following (in which we omit Z for brevity):

	before (atom)
4	all environments
5	$\{(X \mapsto b.nil), (X \mapsto nil)\}$

	after (rule)
1	$\{(Y_1 \mapsto b)\}$
2	$\{(Y_2 \mapsto b)\}$
3	$\{(X \mapsto b.nil, Y_3 \mapsto b)\}$

Let $\mathcal{M}$ denote an interpreter for imperative programs P which executes P in the obvious sequential manner.

Definition 1 *The* <u>collecting semantics</u>[4] *of an imperative program P is the mapping:*

$$\uparrow i \quad into \quad \left\{ \rho : \begin{array}{l} \textit{when } \mathcal{M} \textit{ executes } P \textit{ starting with some environment,} \\ \textit{the environment } \rho \textit{ is encountered just before } S_i \textit{ is executed} \end{array} \right\},$$

$$\downarrow i \quad into \quad \left\{ \rho : \begin{array}{l} \textit{when } \mathcal{M} \textit{ executes } P \textit{ starting with some environment,} \\ \textit{the environment } \rho \textit{ is encountered just after } S_i \textit{ is executed} \end{array} \right\}$$

for all program points $\uparrow i$ and $\downarrow i$ in P. □

Before defining the collecting semantics for logic programs, we use the terminology of CLP [9] and recall some basic definitions. A *goal* is of the form (C, G) where C is a satisfiable conjunction of term equations, and G is a finite sequence of atoms. There is a *derivation step* from (C, G) to (C', G') if (i) A is selected[5] from G; (ii) $H \leftarrow B_1, \cdots, B_n$

[4] For technical convenience, execution of an imperative program is defined to start from any environment.

[5] According to the atom selection rule at hand.

is a rule and σ is a renaming into new variables; (iii) G' is the result of replacing A in G by its *descendents* $\sigma(B_1), \cdots, \sigma(B_n)$, and (iv) C_{n+1} is $C_n \wedge (A = \sigma(H))$. A *derivation* is a sequence of goals such that there is a derivation step from any two successive goals in the sequence. An atom in a goal is *solved* in a derivation if it has no descendents or all its descendents are solved. In the following definition, the only important information contained in the environments of a program point μ pertains to the variables in the rule at hand.

Definition 2 *The* collecting semantics *of a logic program P, whose initial goal is G, is the following mapping: for each rule $R^\mu = H \leftarrow B_1^{\mu_1}, \cdots, B_n^{\mu_n}$, we have*

$$\mu_i \quad \mapsto \quad \left\{ \rho \circ \sigma : \begin{array}{l} \textit{there is a derivation } (true, G), \cdots, (C, G') \\ \textit{such that the renaming } \sigma(B_i^{\mu_i}) \textit{ of } B_i^{\mu_i} \\ \textit{is selected from } G' \textit{ and } \rho \models C. \end{array} \right\}, \quad 1 \leq i \leq n;$$

$$\mu \quad \mapsto \quad \left\{ \rho \circ \sigma : \begin{array}{l} \textit{there is a derivation } (true, G), \cdots, (C, G') \\ \textit{in which the renamings } \sigma(B_1^{\mu_1}), \cdots, \sigma(B_n^{\mu_n}) \textit{ of } B_1^{\mu_1}, \cdots, B_n^{\mu_n} \\ \textit{have just}^6 \textit{ been solved and } \rho \models C. \end{array} \right\} \qquad \square$$

Note that a more general definition of the collecting semantics for logic programs is obtained by collecting the constraints C encountered during execution instead of the environments $\rho \circ \sigma$ above. However in this paper we deal only with the values that variables may assume, and so the present (ground) definition suffices.

We now introduce a more flexible representation of the collecting semantics of a program. This representation makes explicit the essential properties of a collecting semantics without resort to an underlying definition of the operational model. Intuitively the representation captures the minimal notion of when a collecting interpretation is consistent. That is, if an environment ρ is associated with a program point μ, and a single step of program execution results in an environment ρ' at program point μ', then ρ' must be associated with program point μ'. The advantage of this representation is that, by varying the interpretation of the constraints, different approximations of the collecting semantics may be obtained, including approximations defined via abstract interpretation, and those defined in this paper.

An *environment expression* is either an *environment variable*, hereafter denoted by Ψ, or one of the following:

- $\Psi[X \mapsto t]$;
- $\Psi[cond]$;
- $[t_1 \in S_1, \ \ldots \ , t_n \in S_n]$, $n \geq 0$ or
- the union of two environment expressions

[6] Strict super-sequences satisfying this property are discounted.

where X is a program variable, $t, t_1, \ldots, t_n$ are terms, and $S_1, \ldots, S_n$ are expressions for sets of terms and are each of the form $s_1.\Psi_1 \cup s_2.\Psi_2 \cup \ldots s_m.\Psi_m$ where $s_1, \ldots, s_m$ are terms. Intuitively, the first two expressions denote operations on a variable Ψ, while the third expression denotes the construction of a collection of environments. More formally, an *interpretation* $\mathcal{I}$ maps variables Ψ into collections of substitutions, maps expressions of the form $s.\Psi$ into $\{\rho(s) : \rho \in \mathcal{I}(\Psi)\}$, maps $\cup$ into set union, and finally, maps:

- $\Psi[X \mapsto t]$ into $\{\rho[X \mapsto \rho(t)] : \rho \in \mathcal{I}(\Psi)\}$;
- $\Psi[cond]$ into $\{\rho \in \mathcal{I}(\Psi) : \rho \models cond\}$;
- $[t_1 \in S_1, \ \ldots \ , t_n \in S_n]$ into $\{\rho : \rho(t_1) \in \mathcal{I}(S_1), \ \ldots \ , \rho(t_n) \in \mathcal{I}(S_n)\}$[7].

An *environment constraint* is of the form $\Psi \supseteq \mathcal{EE}$ where Ψ is an environment variable and $\mathcal{EE}$ an environment expression. An interpretation $\mathcal{I}$ maps such a constraint into *true* iff $\mathcal{I}(\Psi) \supseteq \mathcal{I}(\mathcal{EE})$.

Definition 3 *The environment constraints EC_P for an imperative program P are:*

- $\Psi^{\uparrow 1} \supseteq [\,]$;
- $\Psi^{\uparrow j} \supseteq \Psi^{\downarrow i}$, *for all sequences of statements $S_i; S_j$ appearing in P;*
- $\Psi^{\downarrow i} \supseteq \Psi^{\uparrow i}[X \mapsto t]$ *for all S_i in P of the form* X :=t;
- $\left.\begin{array}{l} \Psi^{\uparrow j} \supseteq \Psi^{\uparrow i}[cond] \\ \Psi^{\downarrow i} \supseteq \Psi^{\uparrow i}[\neg cond] \cup \Psi^{\downarrow k} \end{array}\right\}$ *for all S_i in P of the form* **if** cond **then** Stat$_j$; ... ;Stat$_k$;
- $\left.\begin{array}{l} \Psi^{\uparrow j} \supseteq \Psi^{\uparrow i}[cond] \cup \Psi^{\downarrow k}[cond] \\ \Psi^{\downarrow i} \supseteq \Psi^{\uparrow i}[\neg cond] \cup \Psi^{\downarrow k}[\neg cond] \end{array}\right\}$ *for all S_i in P of the form* **while** cond **do** Stat$_j$; ... ;Stat$_k$;

The (top-down) environment constraints EC_P for a logic program P are:

- $\Psi^{\eta} \supseteq [H \in Call]$[8]
  for each body atom B^{η} appearing in a rule $H \leftarrow \cdots B^{\eta} \cdots$ in P;
- $\Psi^{\mu} \supseteq [H \in Call, B_1 \in Ret, \ldots, B_n \in Ret]$[9]
  for each rule $H^{\mu} \leftarrow B_1, \ldots, B_n$ in P labelled with μ;

where Call denotes $\left(\bigcup_{B^{\eta} \in body_P} B^{\eta}.\Psi^{\eta} \right)$ *and Ret denotes* $\left(\bigcup_{H^{\mu} \in head_P} H^{\mu}.\Psi^{\mu} \right)$. □

The logic program constraints defined above correspond to a top-down parallel interpreter. They can be easily adapted to the case of bottom-up execution, or to top-down execution with a fixed atom selection rule.

[7] In case $n = 0$, this reduces to $\mathcal{I}([\,]) = \{\rho : \ true\}$, the set of all environments.
[8] This reduces to [] in case the rule is the initial goal.
[9] This reduces to $[B_1 \in Ret, \ldots, B_n \in Ret]$ in case the rule is the initial goal.

Recall the previous example imperative program IP and logic program LP; the environment constraints corresponding to these programs are respectively

$$\begin{array}{lll}
\Psi^{\uparrow 1} & \supseteq & [\,] \\
\Psi^{\downarrow 1} & \supseteq & \Psi^{\uparrow 1}[X \mapsto a.b.c.nil] \\
\Psi^{\uparrow 2} & \supseteq & \Psi^{\downarrow 1} \\
\Psi^{\downarrow 2} & \supseteq & \Psi^{\uparrow 2}[Y \mapsto c] \\
\Psi^{\uparrow 3} & \supseteq & \Psi^{\downarrow 2} \\
\Psi^{\uparrow 4} & \supseteq & \Psi^{\uparrow 3}[X \neq nil] \ \cup \ \Psi^{\downarrow 5}[X \neq nil] \\
\Psi^{\downarrow 4} & \supseteq & \Psi^{\uparrow 4}[Y \mapsto cons_{(1)}^{-1}(X)] \\
\Psi^{\uparrow 5} & \supseteq & \Psi^{\downarrow 4} \\
\Psi^{\downarrow 5} & \supseteq & \Psi^{\uparrow 5}[X \mapsto cons_{(2)}^{-1}(X)] \\
\Psi^{\downarrow 3} & \supseteq & \Psi^{\uparrow 3}[X = nil] \cup \Psi^{\downarrow 5}[X = nil]
\end{array}$$

$$\begin{array}{lll}
\Psi^{①} & \supseteq & [loop(a.b.nil, Y_1) \in Ret] \\
\Psi^{②} & \supseteq & [loop(Y_2.nil, Y_2) \in Call] \\
\Psi^{③} & \supseteq & [loop(Z.X, Y_3) \in Call, loop(X, Y_3) \in Ret] \\
\Psi^{④} & \supseteq & [\,] \\
\Psi^{⑤} & \supseteq & [loop(Z.X, Y_3) \in Call]
\end{array}$$

where *Call* is $loop(a.b.nil, Y_1).\Psi^{④} \ \cup \ loop(X, Y_3).\Psi^{⑤}$
and *Ret* is $loop(Y_2.nil, Y_2).\Psi^{②} \ \cup \ loop(Z.X, Y_3).\Psi^{③}$

Lemma 1 *Let P be an imperative or logic program. Then EC_P has a least model which coincides with the collecting semantics of P.* ☐

3 Approximate Semantics

To formalize the notion of ignoring inter-variable dependencies, a collection of environments is approximated by a *single* mapping from $\mathcal{V}$ into sets. That is, a (possibly infinite) collection of tuples is approximated by a single sequence of (possibly infinite) sets. A *set-based approximation* of a collection of environments $\rho, \rho', \cdots$ is the sequence of sets $(\{\rho(X), \rho'(X), \cdots\}, \{\rho(Y), \rho'(Y), \cdots\}, \cdots)$, where $X, Y, \cdots$ is some listing of $\mathcal{V}$. We call such a sequence a *set substitution* because it associates a set to each variable. We write $\alpha = \mathcal{A}(\{\rho, \rho', \cdots\})$ to denote that the set substitution α is the set-based approximation of $\rho, \rho', \cdots$.

A set substitution may be applied to expressions in a similar manner as in the case for a normal substitution. More formally,

$$\alpha(t) = \begin{cases} \alpha(X) & \text{if } t \text{ is a variable } X; \\ \{f(x_1, \cdots, x_n) : x_i \in \alpha(t_i)\} & \text{if } t \text{ is of the form } f(t_1, \cdots, t_n); \\ \{x_i : f(x_1, \cdots, x_n) \in \alpha(t')\} & \text{if } t \text{ is of the form } f_{(i)}^{-1}(t'). \end{cases}$$

We write $\alpha[X \mapsto S]$ to denote the set substitution which is identical to α except that it maps X to the set S. We also write $\alpha \supseteq \alpha'$ to denote that $\alpha(X) \supseteq \alpha'(X)$ for all $X \in \mathcal{V}$.

Next, define that a *set-based interpretation* $\mathcal{I}$ of environment constraints is like a normal interpretation except for one fundamental change: $\mathcal{I}$ maps variables Ψ into set substitutions, as opposed to collections of environments. Accordingly, $\mathcal{I}$ now maps:

- expressions $t.\Psi$ into the set $\alpha(t)$ where α is $\mathcal{I}(\Psi)$;

- $\Psi[X \mapsto t]$ into $\alpha[X \mapsto \alpha(t)]$ where α is $\mathcal{I}(\Psi)$;
- $\Psi[cond]$ into $\mathcal{A}\left(\left\{\rho : \begin{array}{l} \rho \models cond \ \wedge \\ \forall X\ (\rho(X) \in (\mathcal{I}(\Psi))(X)) \end{array}\right\}\right)$;
- $[t_1 \in S_1, \ \ldots \ , t_n \in S_n]$ into $\mathcal{A}\left(\left\{\rho : \begin{array}{ccc} \rho(t_1) & \in & \mathcal{I}(S_1) \\ & \cdots & \\ \rho(t_n) & \in & \mathcal{I}(S_n) \end{array}\right\}\right)$;
- $\mathcal{EE}_1 \cup \mathcal{EE}_2$ into α such that $\alpha(X) = \mathcal{I}(\mathcal{EE}_1)(X) \cup \mathcal{I}(\mathcal{EE}_2)(X)$ for all $X \in \mathcal{V}$;
- constraints $\Psi \supseteq \mathcal{EE}$ into *true* iff $\mathcal{I}(\Psi) \supseteq \mathcal{I}(\mathcal{EE})$.

In the previous section, the exact semantics of a program was obtained via the least model of the environment constraints at hand. We now obtain

Definition 4 *The* <u>approximate semantics</u> *of an imperative or logic program P is the least*[10] *set-based model of $\mathcal{EC}_P$.* $\Box$

Consider the following logic program LP_2 and its environment constraints:

1. $\leftarrow p(W, X)^{⑤}$.
2. $p(a, a)$.
3. $p(b, b)$.
4. $p(Y, f(f(Z))) \leftarrow p(Y, Z)^{⑥}$.

$$\begin{array}{lcl} \Psi^{①} & \supseteq & [p(W, X) \in Ret] \\ \Psi^{②} & \supseteq & [p(a, a) \in Call] \\ \Psi^{③} & \supseteq & [p(b, b) \in Call] \\ \Psi^{④} & \supseteq & [p(Y, f(f(Z))) \in Call,\ p(Y, Z) \in Ret] \\ \Psi^{⑤} & \supseteq & [\,] \\ \Psi^{⑥} & \supseteq & [p(Y, f(f(Z))) \in Call] \end{array}$$

The following compares the value of $\Psi^{④}$ (the return bindings of the last rule above) in the least model and the least set-based model of the environment constraints. Note that the intervariable dependency between Y and Z exhibited in the least model is absent in the least set-based model.

$\Psi^{④}$, exact semantics	$\Psi^{④}$, approximate semantics
$Y \mapsto a \quad Z \mapsto a$ $Y \mapsto b \quad Z \mapsto b$ $Y \mapsto a \quad Z \mapsto f^2(a)$ $Y \mapsto b \quad Z \mapsto f^2(b)$ $\ldots \quad \ldots$	$Y \mapsto \{a, b\}$ $Z \mapsto \{a, b\} \cup f^2(\{a, b\}) \cup f^4(\{a, b\}) \cup \ldots$

The rest of this paper deals with the problem of how to compute the approximate semantics of a given program. Toward this end, we transform the problem of finding the least set-based model of environment constraints into finding the least model of a

[10] $\mathcal{I} \leq \mathcal{I}'$ if $\mathcal{I}(\Psi) \subseteq \mathcal{I}'(\Psi)$ for all Ψ.

more explicit class of constraints, the set constraints. The following definitions of set expressions and constraints are made with respect to variables of a given program, say $X_1, \cdots, X_N$.

A *simple set expression* is a union and intersection of terms constructed using Σ and a new class of *set variables*, hereafter denoted by $\mathcal{W}, \mathcal{X}, \mathcal{Y}, \mathcal{Z}$. A *quantified set expression* is of the form

$$\{X_i : \exists X_1 \ldots X_{i-1} X_{i+1} \ldots X_N \ \ (F_1 \wedge \cdots \wedge F_n)\}$$

where F_i is either (a) of the form $t \in sexp$, where t is a term and $sexp$ a simple set expression, or (b) of the form $cond$, where $cond$ is an (imperative) program conditional. A *set expression* is a union of simple and quantified set expressions. Finally, a *set constraint* is of the form $\mathcal{X} \supseteq \mathcal{SE}$ where $\mathcal{X}$ is a set variable and $\mathcal{SE}$ a set expression. An interpretation of such a constraint maps each set variable into a set of ground simple terms, and it maps the constraint into *true* and *false* in the obvious way.

We now define how to obtain set constraints $\mathcal{SC}_P$ corresponding to environment constraints $\mathcal{EC}_P$. Intuitively, we will use the set variables $\mathcal{X}_1^\mu, \ldots, \mathcal{X}_N^\mu$ to represent the program variables $X_1, \cdots, X_N$, and accordingly, environment expressions will be rewritten using set variables in place of program variables. Let ω^μ be a renaming substitution which maps each program variable X_i into its set variable counterpart $\mathcal{X}_i^\mu$ w.r.t μ. Also define that Ω maps each expressions $t.\Psi^\mu$ in $\mathcal{EC}_P$ into its set expression counterpart $\omega^\mu(t)$. For example, $\Omega(f(X_1, X_2).\Psi^\mu \cup f(X_2, X_2).\Psi^\nu)$ is $f(\mathcal{X}_1^\mu, \mathcal{X}_2^\mu) \cup f(\mathcal{X}_2^\nu, \mathcal{X}_2^\nu)$. Now, for each environment constraint in $\mathcal{EC}_P$ of the form:

- $\Psi^\mu \supseteq \Psi^\eta$, obtain the set constraints $\displaystyle\bigwedge_{1 \le i \le N} \mathcal{X}_i^\mu \supseteq \mathcal{X}_i^\eta$;
- $\Psi^\mu \supseteq \Psi^\eta[X_i \mapsto t]$, obtain the set constraints

$$\bigwedge_{1 \le j \le N} \mathcal{X}_j^\mu \supseteq \left\{ X_j : \exists X_1 \ldots X_{j-1} X_{j+1} \ldots X_N \begin{pmatrix} X_1 \ \in \ \mathcal{X}_1^\eta \\ \cdots \\ X_i \ \in \ \omega^\eta(t) \\ \cdots \\ X_N \ \in \ \mathcal{X}_N^\eta \end{pmatrix} \right\};$$

- $\Psi^\mu \supseteq \Psi^\eta[cond]$, obtain the set constraints

$$\bigwedge_{1 \le j \le N} \mathcal{X}_j^\mu \supseteq \left\{ X_j : \exists X_1 \ldots X_{j-1} X_{j+1} \ldots X_N \begin{pmatrix} X_1 \ \in \ \mathcal{X}_1^\eta \\ \cdots \\ X_N \ \in \ \mathcal{X}_N^\eta \\ cond \end{pmatrix} \right\};$$

- $\Psi^{\mu} \supseteq [t_1 \in S_1, \ldots, t_n \in S_n]$, obtain the set constraints

$$\bigwedge_{1 \le j \le N} \mathcal{X}_j^{\mu} \supseteq \left\{ X_j : \exists X_1 \ldots X_{j-1} X_{j+1} \ldots X_N \begin{pmatrix} t_1 & \in & \Omega(S_1) \\ & \cdots & \\ t_N & \in & \Omega(S_N) \end{pmatrix} \right\};$$

- $\Psi \supseteq \mathcal{E}\mathcal{E}_1 \cup \mathcal{E}\mathcal{E}_2$, continually reduce such a constraint into $\Psi \supseteq \mathcal{E}\mathcal{E}_1$ and $\Psi \supseteq \mathcal{E}\mathcal{E}_2$ until the resulting constraints fall into the above cases.

Definition 5 *The set constraints $\mathcal{SC}_P$ corresponding to a imperative or logic program P is obtained from $\mathcal{EC}_P$ as described above.* $\Box$

Consider the previous example logic program LP_2. The constraints in $\mathcal{SC}_P$ corresponding to the program points 4, 5 and 6 are:

$$\begin{array}{lcl}
\mathcal{Y}^{④} & \supseteq & \left\{ Y : \exists Z \begin{pmatrix} p(Y, f(f(Z))) & \in & p(\mathcal{W}^{⑤}, \mathcal{X}^{⑤}) \cup p(\mathcal{Y}^{⑥}, \mathcal{Z}^{⑥}) \\ p(Y, Z) & \in & p(a, a) \cup p(b, b) \cup p(\mathcal{Y}^{③}, f(f(\mathcal{Z}^{③}))) \end{pmatrix} \right\} \\
\mathcal{Z}^{④} & \supseteq & \left\{ Z : \exists Y \begin{pmatrix} p(Y, f(f(Z))) & \in & p(\mathcal{W}^{⑤}, \mathcal{X}^{⑤}) \cup p(\mathcal{Y}^{⑥}, \mathcal{Z}^{⑥}) \\ p(Y, Z) & \in & p(a, a) \cup p(b, b) \cup p(\mathcal{Y}^{③}, f(f(\mathcal{Z}^{③}))) \end{pmatrix} \right\} \\
\mathcal{W}^{⑤} & \supseteq & \{ W : \exists X \ \mathit{true} \} \\
\mathcal{X}^{⑤} & \supseteq & \{ X : \exists W \ \mathit{true} \} \\
\mathcal{Y}^{⑥} & \supseteq & \left\{ Y : \exists Z \left(p(Y, f(f(Z))) \in p(\mathcal{W}^{⑤}, \mathcal{X}^{⑤}) \cup p(\mathcal{Y}^{⑥}, \mathcal{Z}^{⑥}) \right) \right\} \\
\mathcal{Z}^{⑥} & \supseteq & \left\{ Z : \exists Y \left(p(Y, f(f(Z))) \in p(\mathcal{W}^{⑤}, \mathcal{X}^{⑤}) \cup p(\mathcal{Y}^{⑥}, \mathcal{Z}^{⑥}) \right) \right\}
\end{array}$$

A *model* of $\mathcal{SC}_P$ defined in the obvious way. Note that each model $\mathcal{I}$ of $\mathcal{SC}_P$ defines a collecting interpretation for P because it associates to each program point μ the set of environments $\{\rho : \rho(X_i) \in \mathcal{I}(\mathcal{X}_i^{\mu}), 1 \le i \le N\}$.

Lemma 2 *The least model of $\mathcal{SC}_P$ defines the approximate semantics of P.*

4 The Algorithm

Partition set constraints $\mathcal{X} \supseteq \mathit{exp}$ into the cases: (a) exp is a simple set expression, in which case the constraint is called *explicit*, and (b) exp is a quantified set expression. Let the notation *explicit*($\mathcal{SC}$) denote the conjunction of just the explicit constraints in a conjunction $\mathcal{SC}$.

A simplified overview of the main algorithm is that it starts with constraints $\mathcal{SC}_P$ obtained from a program P, and by performing a series of transformations, incrementally converts the constraints into explicit form. In each transformation step $\mathcal{SC} \Longrightarrow \mathcal{SC}'$, a condition on *explicit*($\mathcal{SC}$) must be satisfied before the step is applied. Thus some key subroutines of the algorithm are for reasoning about explicit constraints. These subroutines can be adapted from [8] which contained an algorithm to construct a representation[11] of the least model of given explicit constraints.

Let $\mathcal{SC}$ denote the conjunction of set constraints at hand. Let $\tilde{\mathcal{X}}^\mu \neq \emptyset$ abbreviate $\mathcal{X}_1^\mu \neq \emptyset \wedge \cdots \wedge \mathcal{X}_N^\mu \neq \emptyset$ where $X_1, \cdots, X_N$ are the program variables at hand. Let $\tilde{\Psi} \in \tilde{\Phi}$ abbreviate a conjunction of the form $\Psi_1 \in \Phi_1 \wedge \cdots \wedge \Psi_n \in \Phi_n$, and in particular, let $\tilde{X} \in \tilde{\mathcal{X}}^\mu$ abbreviate $X_1 \in \mathcal{X}_1^\mu \wedge \cdots \wedge X_N \in \mathcal{X}_N^\mu$. Further, let $\tilde{\exists}_{\bar{i}}$ abbreviate the sequence of quantifiers $\exists X_1 \cdots X_{i-1} X_{i+1} \cdots X_N$. The following outlines some of the transformation steps applied to $\mathcal{SC}$ *in the context* of a conjunction of explicit constraints $\mathcal{ESC}$:

- If $\mathcal{ESC} \models \tilde{\mathcal{X}}^\mu \neq \emptyset \;\wedge\; \mathcal{X}_j^\mu \cap \mathcal{X}_k^\mu \neq \emptyset$ then replace any expression $\left\{X_i : \tilde{\exists}_{\bar{i}} \left(\tilde{X} \in \tilde{\mathcal{X}}^\mu \wedge X_j = X_k\right)\right\}$ in $\mathcal{SC}$ by $\mathcal{X}_j^\mu \cap \mathcal{X}_k^\mu$ if i is j or k, and $\mathcal{X}_i^\mu$ otherwise.
- If $\mathcal{ESC} \models \tilde{\mathcal{X}}^\mu \neq \emptyset \;\wedge\; |\mathcal{X}_j^\mu| \geq 2 \;\wedge\; |\mathcal{X}_k^\mu| \geq 2$ and $j \neq k$ then replace any expression $\left\{X_i : \tilde{\exists}_{\bar{i}} \left(\tilde{X} \in \tilde{\mathcal{X}}^\mu \wedge X_j \neq X_k\right)\right\}$ in $\mathcal{SC}$ by $\mathcal{X}_i^\mu$.
- Replace any expression $\left\{X_i : \tilde{\exists}_{\bar{i}} \left(\tilde{\Psi} \in \tilde{\Phi} \;\wedge\; s \in t_1 \cup t_2\right)\right\}$ in $\mathcal{SC}$ by $\left\{X_i : \tilde{\exists}_{\bar{i}} \left(\tilde{\Psi} \in \tilde{\Phi} \wedge s \in t_1\right)\right\} \cup \left\{X_i : \tilde{\exists}_{\bar{i}} \left(\tilde{\Psi} \in \tilde{\Phi} \wedge s \in t_2\right)\right\}$
- Replace any expression $\left\{X_i : \tilde{\exists}_{\bar{i}} \left(\tilde{\Psi} \in \tilde{\Phi} \;\wedge\; f(\tilde{s}) \in g(\tilde{t})\right)\right\}$ in $\mathcal{SC}$ by $\left\{X_i : \tilde{\exists}_{\bar{i}} \left(\tilde{\Psi} \in \tilde{\Phi} \;\wedge\; \tilde{s} \in \tilde{t}\right)\right\}$ if $f = g$ and by $\emptyset$ otherwise.
- If $\mathcal{ESC} \models s_{j,1} \cap \cdots \cap s_{j,m_j} \neq \emptyset$, $1 \leq j \leq N$, then replace any expression
$$\left\{X_i : \tilde{\exists}_{\bar{i}} \left(\begin{array}{c} X_1 \in s_{1,1} \\ \cdots \\ X_1 \in s_{1,m_1} \end{array} \bigwedge \cdots \bigwedge \begin{array}{c} X_N \in s_{N,1} \\ \cdots \\ X_N \in s_{N,m_N} \end{array} \right)\right\}$$ in $\mathcal{SC}$ by $s_{i,1} \cap \cdots \cap s_{i,m_i}$.

We first remark that there are in fact a considerable number of additional transformations; they are omitted because they require lengthy preliminary definitions. Further, the transformations can only be applied in certain orders. This is because the interactions between them is particularly problematic for the guarantee of termination. The details are considerable and are left to the full version of this paper.

Let REDUCE($\mathcal{SC}$, $\mathcal{ESC}$) denote the result of exhaustively applying the above transformations to $\mathcal{SC}$ in the context of $\mathcal{ESC}$. The main algorithm, whose simplified format is in figure 1, is essentially a repeated application of REDUCE. In each iteration, a collection $\mathcal{ESC}$ of explicit constraints is extracted from the current set constraints $\mathcal{SC}$. REDUCE is then applied on $\mathcal{SC}$ and $\mathcal{ESC}$ to obtain new set constraints for the next iteration.

[11] This explicit representation consists of a regular set of terms [6] for each variable at hand.

input an imperative or logic program P;
obtain the environment constraints $\mathcal{EC}_P$ of P;
obtain the set constraints $\mathcal{SC}_P$ from $\mathcal{EC}_P$;
$i\ :=\ 0$;
$\mathcal{SC}_0\ :=\ \mathcal{SC}_P$;
repeat
$\mathcal{ESC}_{i+1}\ :=\ \mathit{explicit}(\mathcal{SC}_i)$;
$\mathcal{SC}_{i+1}\ :=\ \text{REDUCE}(\mathcal{SC}_i, \mathcal{ESC}_{i+1})$;
$i\ :=\ i+1$;
until ($\mathcal{SC}_i$ is syntactically the same as $\mathcal{SC}_{i-1}$);
output the least model of $\mathcal{ESC}_i$;

Figure 1: **Main Algorithm**

Theorem: *The main algorithm, when input with an imperative or logic program P:*

- *terminates, and outputs some $\mathcal{I}$;*
- *$\mathcal{I}$ is the approximate semantics of P.* □

Termination is proved by establishing a metric on the form of quantified set expressions that appear in $\mathcal{SC}_i$. That the transformations are correct, in the sense that they preserve models, can be easily verified. The essence of the proof of global correctness is in showing that when the algorithm terminates, say with $i = n$, all information about the least model of $\mathcal{SC}_0$ is contained in the definite constraints of $\mathcal{SC}_n$.

We finally remark that the initial set of environments for an imperative program, presently predefined as the set of all environments, can be any set Ψ_0 which is definable by a mapping of each variable into a regular set of terms. All that is required is to replace the set constraints corresponding to $\Psi^{\uparrow 1} \supseteq [\,]$ by set constraints corresponding to $\Psi^{\uparrow 1} \supseteq \Psi_0$, and then conjoining appropriate set constraints which characterize Ψ_0.

5 Conclusion

The problem considered was formalizing approximate program semantics which (a) most naturally captures the notion of ignoring intervariable dependencies, (b) is as accurate as possible, and (c) is decidable. The first step taken was to define the approximate semantics using environment constraints. The following alternative characterizations of the approximation at hand justify this constraint approach. In the case of both imperative and logic programs, our approximation essentially corresponds to an abstract interpretation in which arbitrary set substitutions are used as the descriptions of collections of environments. Note that such an abstract interpretation approach cannot be directly used to compute the approximation. Another view, for the case of imperative programs, uses a restricted Hoare-style proof system. Recall that the exact collecting semantics of a program P can be characterized as just those environments consistent with a (Hoare-)provable assertion at that point. Accordingly, our approximate semantics essentially corresponds to provability in the context of a *restricted* assertion language in which no intervariable dependencies may be expressed. An example language: conjunctions of the form $\Phi_1(X_1) \wedge \ldots \wedge \Phi_N(X_N)$ where each $\Phi_i(X_i)$ is some first-order formula in which the program variable X_i is the only free variable.

Next addressed was the computation of the approximate semantics. Toward this end, it was first shown that the approximate semantics of an imperative or logic program can be represented uniformly and concisely by a conjunction of set constraints. More specifically, the least model of these constraints characterized the approximate semantics of the program. Finally, an algorithm for constructing a representation of the least model of given set constraints was outlined.

What was thus presented was a precise analyzer for reasoning about programs in the light of the program approximation considered. Consequently, it is strictly more accurate than previous analysis techniques for imperative, functional and logic programming which ignore intervariable dependencies. An additional advantage of the present algorithm is that it is extendible to other kinds of analyses, for example, structure sharing and mode analysis for logic programs. This is essentially because the algorithm preserves certain runtime properties; for example, if X and Y do not share the same *occurrence* of a term, then X and Y cannot share structure.

Work remains to be done in addressing efficiency considerations. The algorithm, as outlined, emphasized clarity. An implementation in progress reorganizes the algorithm to maintain constraints in a minimal form which helps to substantially reduce redundant computations. Further work also includes considering other classes of programs, $CLP(\mathcal{R})$ for example. Note that there are classes of programs where ignoring intervariable dependencies does not yield a recursive approximation. An example is where program conditions may be arbitrary integer arithmetic expressions. We finally remark that there are important kinds of analysis for which this work does not seem applicable, for example, that which deals with variable aliasing.

References

[1] S. Abramsky and C. Hankin (Eds.), *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.

[2] M. Bruynooghe and G. Janssens, "An Instance of Abstract Interpretation integrating Type and Mode Inference", *Proceedings* 5^{th} *Int. Conf. on Logic Programming*, K.A. Bowen and R.A. Kowalski (eds), MIT Press, pp 669 - 683, August 1988.

[3] S.K. Debray, "Global Optimization of Logic Programs", Ph.D. Thesis, Department of Computer Science, SUNY at Stony Brook, 233 pages, December 1986.

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

[5] P. Cousot and R. Cousot, "Systematic Design of Program Analysis Frameworks", *Proceedings* 6^{th} *ACM Symposium on Principles of Programming Languages*, 1979, pp 269 - 282.

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

[7] N.C. Heintze and J. Jaffar, "A Finite Presentation Theorem for Approximating Logic Programs", *Proceedings* 17^{th} *ACM Symposium on Principles of Programming Languages*, 1990, pp 197 - 209.

[8] N.C. Heintze and J. Jaffar, "A Decision Procedure for a Class of Herbrand Set Constraints", *Proceedings* 5^{th} *IEEE Symposium on Logic in Computer Science*, June 1990, pp 42 - 51.

[9] Jaffar, J. and Lassez, J-L. "Constraint Logic Programming", Technical Report 86/73, Dept. of Computer Science, Monash University (June 1986). [An abstract appears in the *Proceedings* 14^{th} *ACM Symposium on Principles of Programming Languages*, Munich (January 1987), 111–119.]

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

[11] N.D. Jones and S.S. Muchnick, "Complexity of Flow Analysis, Inductive Assertion Synthesis, and a Language due to Dijkstra", *Proceedings* 21^{st} *Conference on Foundations of Computer Science*, 1980, pp 185 - 190, also in, *Program Flow Analysis: Theory and Applications*, N.D. Jones and S.S. Muchnick (eds), Prentice-Hall, 1981.

[12] N.D. Jones and S.S. Muchnick, "A Flexible Approach to Interprocedural Dataflow Analysis and Programs with Recursive Data Structures", *Proceedings* 9^{th} *ACM Symposium on Principles of Programming Languages*, 1982, pp 244 - 256.

[13] F. Pfenning (Ed.), *Types in Logic Programming*, MIT Press Series in Logic Programming, 1991, forthcoming.

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