# Set-Based Analysis of ML Programs*

*(Extended Abstract)*

Nevin Heintze†
School of Computer Science
Carnegie Mellon University
nch@cs.cmu.edu

## Abstract

Reasoning about program variables as sets of "values" leads to a simple, accurate and intuitively appealing notion of program approximation. This paper presents approach for the compile-time analysis of ML programs. To develop the core ideas of the analysis, we consider a simple untyped call-by-value functional language. Starting with an operational semantics for the language, we develop an approximate "set-based" operational semantics, which formalizes the intuition of treating program variables as sets. The key result of the paper is an $O(n^3)$ algorithm for computing the set based approximation of a program. We then extend this analysis in a natural way to deal with arrays, arithmetic, exceptions and continuations. We briefly describe our experience with an implementation of this analysis for ML programs.

## 1 Introduction

Information about the run-time behavior of a program is necessary for many important compiler optimizations. This information is typically computed by some kind of program analysis. Such analysis often takes the following form: given a program (or program fragment), compute *invariants* about the possible values of variables. For compilation of functional programming languages, important issues include:

**array bounds checks:** For languages with "safe" array operations, the cost of performing run-time array bounds checking can be prohibitive. Experiences in the CMU FOX project [6] (which addresses systems building in ML) suggest that this is a critical issue for ML implementations of software such as the TCP/IP network protocol suite.

**control flow:** In a language with higher-order functions, the flow of control from one program statement to another is not explicit. However, a knowledge of control flow is needed to perform many traditional loop optimization transformations.

**redundant tests:** Many tests that are performed during program execution can be eliminated using analyses that compute information about the run-time values of variables. Examples include tag-checking, run-time type checking and tests relating to pattern matching and case statements.

**inlining and specialization:** Inlining and specialization of program fragments can lead to significant program improvements. The key issue is when to apply these transformations. Information about control flow and run-time values of variables can provide important guidance.

Other areas where program analysis can be utilized are data representation, distributed computation (for which information about structure sharing and structure access are important) and program verification (program analysis can be viewed as a "weak" logic for establishing useful program invariants).

Most program analysis research targets a specific compilation optimization. As a result, when combining a variety of optimizations, it is necessary to perform a number of different analyses. Thus, from a conceptual as well as an engineering perspective, it is desirable to design a single analysis that covers a number of needs. This makes particular sense for analysis of higher-order functional languages since there are interactions between the various analyses (the interactions between control flow information and data-flow analysis are well known).

This paper describes an approach to program analysis for call-by-value higher-order functional languages that addresses a variety of program analysis/transformation needs (including those mentioned above). Unlike other works in the literature, it is designed around a single uniform definition of approximation: all dependencies between variables are ignored by treating programs variables as sets of values. No other approximations are present (in particular, there is no "abstract domain" to approximation the underlying domain of values). To develop the foundations of this *set-based* analysis, we start with an operational semantics for a small call-by-value functional language. We then modify this operational semantics so that the notion of environment is replaced by a set environment, which maps program variables into sets of values. These sets are arbitrary in the sense that no *a priori* assumptions are made about finiteness or

*This work was sponsored by the Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168.

†5000 Forbes Ave., Pittsburgh, PA 15213, USA.

representability of these sets. This set-based operational semantics is used to define the *set-based* approximation of a program. Importantly, we then show that even though the approximation is defined using a system that allows reasoning about arbitrary sets of values, the program approximation that arises is decidable, and can in fact be computed on $O(n^3)$ time. This leads to an analysis with the following properties:

- The analysis provides relatively accurate information about a program, with particular emphasis on the program's data structures.
- The underlying notion of approximation has a simple uniform definition, and the results of the analysis are intuitive and predictable.
- It is flexible and easily modified to incorporate arithmetic and operations such as assignment, continuations and exception handling.

### Inter-Variable Dependencies

To illustrate the ideas of set-based analysis and in particular to show what is meant by inter-variable dependencies, consider some example ML programs.

```
let fun mk_list (u, v) = [u, v]
in
    mk_list(1,2);
    mk_list(3,4)
end
```

**Program 1**

During the execution of Program 1, the body of the function mk_list is executed in two environments: [u ↦ 1, v ↦ 2] and [ u ↦ 3, v ↦ 4]. Inter-variable dependencies arise here in the sense that the variable u takes the value 1 exactly when v takes 2, and u takes 3 when v takes 4. In general, we say that inter-variable dependencies arise whenever the set of environments encountered at some program point is such that fixing a value for one or more variables restricts the possible values of the other variables.

Such dependencies may be ignored by treating the program variables as denoting sets of values instead of individual values. In Program 1, the sets for u and v are {1, 3} and {2, 4} respectively. If program variables are treated as sets, then the result of Program 1 is approximated by the *set* of values {[1,2], [1,4], [3,2], [3,4]}, in contrast to its actual result which is the single value [3,4].

```
let fun append(x :: xs, y) = x :: append(xs, y)
      | append(nil, y') = y'
    fun rev(z :: zs) = append(rev zs, [z])
      | rev nil = nil
in  rev [1,2,3,4]
end
```

**Program 2**

In Program 2, dependencies arise between the variables x, xs and y in the function append, and between z and zs in the function rev. If the values of variables are collected into sets, then we obtain the set {1,2,3,4} for both x and z. Using this information, a set-based interpretation of the program can be developed as follows. Consider the definition of append. From the first clause, we see that the values returned by append include values 1 :: $l$, 2 :: $l$, 3:: $l$ and 4 :: $l$ where $l$ is some list returned by append. From the second clause, the values returned by append include any value of y, and noting the call append(rev zs, [z]) in the definition of rev, these values include the singleton lists [1], [2], [3] and [4]. Combining these two observations, it is easy to see that the set-based interpretation of Program 2 yields the set of all lists constructed from 1, 2, 3 and 4.

The notion of set-based approximation can be extended in a variety of ways. For example, consider programs involving arrays. In keeping with the methodology of ignoring dependencies, we shall ignore the dependencies between subscripts and array values. In essence, we treat an array as a set of values. When the array is updated, a new value is added to this set. When the array is subscripted, the whole set is returned. For example, the set-based approximation of Program 3 yields the set of all values obtained by summing any number of 3's and 4's i.e. $\{3n + 4m : m \geq 0, n \geq 0, m + n \geq 1\}$.

```
let fun cum (arr : int array) =
    let fun f 0 = arr sub 0
          | f i = (arr sub i) + f(i − 1)
    in
        f ((length arr) − 1)
    end
    val arr = array(10, 3)
in
    update(arr, 6, 4);
    cum arr
end
```

**Program 3**

```
fun map f (x :: l) = (f x) :: (map f l)
  | map f nil = nil

val t = [1,2,3]
val d = dynamic
val u = map (fn x ⇒ (x, d)) t
val v = map (fn (x, y) ⇒ x) u
val w = map (fn (x, y) ⇒ y) u
```

**Program 4**

Set-based approximation can also be extended to deal with non-standard values. For example, to perform a binding time analysis [8, 18, 24], a non-standard value dynamic is introduced to represent a value that will not be known until "run-time". To illustrate this, consider Program 4. The set-based approximation of this program yields the following information[1] about the variables u, v and w: u is a list of pairs whose first element is either 1, 2 or 3 and whose second argument is dynamic; v is a list of 1's, 2's and 3's, and w is a list of dynamic's.

To summarize, the analysis developed here is based on the notion of ignoring inter-variable dependencies by treating variables as sets. In other words, the environments encountered at each point in a program are collapsed into a single set environment (mapping from variables into sets). Strictly speaking, there are three kinds of dependencies that

[1] This information is in fact obtained only if the analysis of map is "polyvariant" (that is, provided there can be different "versions" of map). We refer to this issue later in the paper.

are ignored in set-based analysis. First, dependencies between different variables are ignored – this was illustrated by Program 1. Second, dependencies between different occurrences of the same variable are ignored. For example the approximation of Program 5 yields {[1,1], [1,2], [2,1], [2,2]} and not {[1,1], [2,2]}. Third, dependencies between the domain and codomain of functions are ignored. For example the approximation of Program 6 yields {2,3} and not {3}.

```
let fun f x = [x,x]
in
    f 1;
    f 2;
end
```

**Program 5**

```
let fun g 1 = 2
      | g 2 = 3
in
    g 1;
    g 2;
end
```

**Program 6**

### Overview of paper

The body of this paper consists of two main components. First, we develop the underlying ideas of set-based analysis. We give a simple and natural formalization of the notion of set-based approximation, and then present an algorithm (based on constructing and solving set constraints) for computing this approximation. This is carried out in the context of a small untyped call-by-value functional language that is intended to be suggestive of a number of aspects of ML [22]. Second, we describe an implementation for the set-based analysis of ML programs that extends the basic notions of set-based analysis to arithmetic, arrays, continuations and exceptions. This implementation is built on the LAMBDA intermediate representation of the SML/NJ compiler [4]. Typical execution times are about 200-400 lines per second for programs up to several thousand lines in length. The core part of the implementation provides accurate type information for variables and functions as well as control flow information (which in turn can be used to check the conditions of "safety analysis" [27]). The implementation has also been adapted to perform polyvariant partially-static binding time analysis [8].

## 2 Related Literature

The approximation of ignoring inter-variable dependencies is used in many works on program analysis. It was identified as an important notion by Jones and Muchnick in [20], where it was called *independent attribute analysis*.

The idea of defining an approximation of a program by ignoring inter-variables dependencies and making no other approximation (that is, by treating program variables as set of values) has been used previously in the analysis of logic programs and imperative programs by Jaffar and the present author [10, 12, 13]. This paper extend the approach to analysis of functional programs. We now briefly review several areas of related work.

### Soft Typing

The aim of soft typing [7] is to infer "types" for identifiers and expressions in a dynamically typed language. These are used to remove unnecessary run-time type checks as well as help document programs. Recent work [30] describes a soft typing system for Scheme (including a treatment of assignment and callcc). From a program analysis viewpoint, soft typing incorporates a notion of approximation akin to ignoring inter-variable dependencies, in addition to other kinds of approximation. It therefore appears that, in principal, it is less accurate than set-based analysis. However, issues such as the treatment of polymorphism and the specific operations of the languages used, make comparisons somewhat problematic. At a systems level, early comparisons indicate that set-based analysis is somewhat faster than the soft typing system of Wright [30]; however again direct comparison is difficult because the two systems treat different languages.

Perhaps more closely related to our work is the soft typing system developed by Aiken et. al. [2, 3]. Their system extracts type constraints from a program and provides a normalization procedure for solving these constraints over the domain of downward closed sets of finite elements (essentially the "ideal" model of types). These constraints are similar in spirit to our constraints (in particular, they express relationships between sets of values). However, the constraints used and their simplification algorithm are very different from those used in set-based analysis. In terms of accuracy, the key observation is again that, at a conceptual level, set-based analysis represents an upper bound on the accuracy of systems that ignore dependencies between variables. Since the systems of [2, 3] effectively ignore such dependencies, they will in general be less accurate. However, other issues, such as the treatment of polymorphism, complicate this relationship. We note that, whereas the development of set-based analysis starts with an operational semantics, the construction of constraints in [2, 3] is inspired by a denotational semantics. We also note that both systems include a mechanism for reasoning about non-emptiness of sets (these are called "conditional types" in [3]).

A comparison between [3] and set-based analysis at the performance level indicates that set-based analysis is substantially faster. For example, [3] reports that the largest programs that have been analyzed are in the order of several hundred lines and for such programs, this analysis takes about 15-30 seconds. In contrast, set-based analysis has been used to analyze programs of the order of 2000-3000 lines in about 5-10 seconds. This appears to be related to the underlying complexities of the methods used. For set-based analysis, the core algorithm is $O(n^3)$; for the algorithm of [3], the complexity is exponential-time. In fact, due to efficiency problems, the implementation described in [3] does not implement the accompanying type system exactly, but makes some additional approximations. However, our implementation of set-based analysis implements the notion of set-based program approximation exactly.

The two soft typing systems described above and the set-based analysis system described in this paper were independently developed over the same period of time. One advantage of the soft typing systems is that they directly address the on-line type inference problem: given a program fragment, determine a type for this fragment. In contrast, set-based analysis is designed for global program analysis. The tradeoffs here are difficult to characterize. Finally, we note that an important difference between soft typing systems and our implementation of set-based analysis is that we address a much larger scope of analysis problems, including control flow analysis and reasoning about arithmetic tests.

### Constraints

Constraints have also been used in binding time analysis [15] and safety analysis [27]. In the former, the program ap-

proximation that arises is different from set-based approximation (and in fact less accurate), but can be computed in almost-linear time. In the latter (which is based on closure analysis), the constraints are solved over subsets of a finite domain of "closures". In contrast, our constraints are solved over an infinite domain. (Very recent work [26, 28] essentially extends the approach in [27] to data-constructors and side effects for an object oriented language).

### Grammars

Another closely related work is by Jones [17] where a grammar approach is presented for the analysis of lazy higher-order functional programs (this is one of the first treatments of control flow analysis in the literature). Also see [16, 23] for subsequent developments.

In summary, one of the main aspect of our work that sets it apart from other works is that we start with a simple, intuitive definition of approximate semantics based on an operational semantics, and only then present algorithms (using constraints) that correspond exactly to this approximation. Moreover, we extend this analysis to deal with side effects and continuations in a uniform and intuitive manner.

## 3 Set-Based Approximation

Consider a simple call-by-value functional language whose terms $e$ are defined by

$$e \quad ::= \quad x \mid c(e_1, \ldots, e_n) \mid \lambda x.e \mid e_1\ e_2 \mid \mathit{fix}\ x.e \\ \mid \mathit{case}(e_1,\ c(x_1, \ldots, x_n) \Rightarrow e_2,\ y \Rightarrow e_3)$$

where $x$ and $y$ range over program variables and $c$ ranges over a given set of (varying arity, "first-order") constants. It is convenient to adopt the usual convention that bound variables appearing in a term are distinct (that is, each variable in a term has at most one binding occurrence). The operator *case* is essentially a very restricted form of the ML case expression and provides a mechanism for branching on the result of a computation as well as "deconstructing" values. The operator *fix* serves to express recursion[2].

The operational semantics for the evaluation of the closed terms of this language is given in Figure 1. The variables $E$ and $v$ range over environments and values respectively, and these are defined mutually recursively as follows. An *environment* $E$ is a finite mapping from program variables into binding expressions. A *binding expression* is either a value or an expression of the form $\langle E, \mathit{fix}\ x.e\rangle$. A *value* $v$ is of the form $c(v_1, \ldots, v_n)$ where the $v_i$ are values, or a closure of the form $\langle E, \lambda x.e\rangle$ where $E$ is an environment. If $E$ is an environment then we write $dom(E)$ to denote the (finite) set of variables on which $E$ is defined. The notation $E[x \mapsto exp]$ denotes the environment that maps $x$ into $exp$ and all other variables $x'$ into $E(x')$. We write $\vdash e \rightarrow v$ if $E \vdash e \rightarrow v$ when $E$ is the empty environment.

This operational semantics is a fairly standard environment-based semantics. The only slightly unusual component is the treatment of *fix*. In essence, there is a choice between (a) expressing recursion directly at the level of environments and (b) extending the definition of environments so that variables map not just into values, but also to certain kinds of non-values that represent *fix* expressions. For presentational simplicity we have chosen the latter, although this issue is orthogonal to the developments in the rest of the paper. We also note that the inclusion of environments in binding expressions $\langle E, \mathit{fix}\ x.e\rangle$ is done for clarity and is not strictly necessary, given our assumed convention that each bound variable is distinct.

We now modify the operational semantics so that dependencies between variables are ignored. This is achieved by treating program variables as sets of values. To formalize this, first define that a *set environment* $\mathcal{E}$ is a finite mapping from variables into sets of binding values. The variable $V$ (possibly subscripted) will be used to denote sets of values. An expression $c(V_1, \ldots, V_n)$, where the $V_i$ are sets, denotes the set of values $\{c(v_1, \ldots, v_n) : v_i \in V_i, i = 1..n\}$. The expression $\langle \mathcal{E}, \lambda x.e\rangle$ denotes the set of closures $\{\langle E, \lambda x.e\rangle : E(x) \in \mathcal{E}(x)$ for each $x \in dom(\mathcal{E})\}$. The *set based operational semantics*, presented in Figure 2, is essentially obtained by replacing environments $E$ in the rules of Figure 1 by set environments[3] $\mathcal{E}$, and values $v$ by sets of values $V$. That is, whereas the (standard) operational semantics defines a relationship $E \vdash e \rightarrow v$, the set-based operation semantics defines a relationship $\mathcal{E} \vdash e \rightsquigarrow V$ which should be read as: in the context of set environment $\mathcal{E}$ the term $e$ "approximately" evaluates to $V$.

This replacement necessitates two kinds of changes to the rules. First, the two variable rules VAR-1 and VAR-2 are modified to accommodate the fact that $\mathcal{E}(x)$ is a set. Second, the rules that involve variable binding (APP, CASE-1, CASE-2 and FIX) are modified so that the binding information is dropped. Note that the set-based semantics is non-deterministic: in general, there will a number of sets $V$ such that $\mathcal{E} \vdash e_0 \rightsquigarrow V$. To obtain the result of the set-based execution of a closed term $e_0$ in the context of $\mathcal{E}$, we shall collect the various sets together to form the set $\{v \in V : \mathcal{E} \vdash e_0 \rightsquigarrow V\}$.

Observe that a number of the rules in Figure 1 will in general lead to an unsound approximation. That is, certain set environments $\mathcal{E}$ will be such that for some closed terms $e_0$, $\vdash e_0 \rightarrow v$ but there is no set $V$ such that $\mathcal{E} \vdash e_0 \rightsquigarrow V$ and $v \in V$. We shall however always ensure that whenever one of these rules is applied, $\mathcal{E}$ is "sufficiently large" that it contains all bindings to variables. To illustrate this issue, consider the term $(\lambda f.c(f\ a, f\ b))\ \lambda x.x$, where $a$, $b$ and $c$ are constants[4]. Denote this term by $e_0$. Now, consider three set environments $\mathcal{E}_1$, $\mathcal{E}_2$ and $\mathcal{E}_3$ satisfying:

$$\mathcal{E}_1(x) = \{\} \qquad \mathcal{E}_2(x) = \{v : v \text{ is any value}\}$$
$$\mathcal{E}_1(f) = \{\} \qquad \mathcal{E}_2(f) = \{v : v \text{ is any value}\}$$

$$\mathcal{E}_3(x) = \{a, b\}$$
$$\mathcal{E}_3(f) = \{\langle E, \lambda x.x\rangle : E \text{ is any environment}\}$$

Consider the set-based semantics of $e_0$ under $\mathcal{E}_1$, $\mathcal{E}_2$ and $\mathcal{E}_3$ in turn. For $\mathcal{E}_1$, the only derivation of the form $\mathcal{E}_1 \vdash e_0 \rightsquigarrow V$ is such that $V = \{\}$. Hence, the use of $\mathcal{E}_1$ does not lead to a safe approximation of the execution of $e_0$. Now consider $\mathcal{E}_2$. There are many derivations of the form $\mathcal{E}_2 \vdash e_0 \rightsquigarrow V$ and the set $\{v \in V : \mathcal{E}_2 \vdash e_0 \rightsquigarrow V\}$ is $\{c(v, v') : v$ and $v'$ are values$\}$. This is a safe approximation of $e_0$, but not a particularly useful one. Finally, consider $\mathcal{E}_3$. In this case, the set $\{v \in V : \mathcal{E}_3 \vdash e_0 \rightsquigarrow V\}$ is $\{c(a,a), c(a,b), c(b,a), c(b,b)\}$.

[2] In $\mathit{fix}\ x.e$, the expression $e$ shall typically be an abstraction.

[3] We remark that one reason for the explicit use of environments in the operational semantics in Figure 1 is precisely to enhance this intuition. However, the notion of set based approximation is not limited to this style of semantics. Analogous definitions can be made starting from an operational semantics that uses substitution.

[4] We shall write $a$ as an abbreviation of $a()$.

$$E \vdash x \to v \quad (v = E(x), v \neq \langle E, \mathit{fix}\ y.e\rangle) \tag{VAR-1}$$

$$\frac{E' \vdash \mathit{fix}\ y.e \to v}{E \vdash x \to v} \quad (E(x) = \langle E', \mathit{fix}\ y.e\rangle) \tag{VAR-2}$$

$$\frac{E \vdash e_1 \to \langle E', \lambda x.e\rangle \qquad E \vdash e_2 \to v' \qquad E'[x \mapsto v'] \vdash e \to v}{E \vdash e_1\ e_2 \to v} \tag{APP}$$

$$\frac{E \vdash e_1 \to v_i,\ i = 1..n}{E \vdash c(e_1, \ldots, e_n) \to c(v_1, \ldots, v_n)} \tag{CONST}$$

$$E \vdash \lambda x.e \to \langle E, \lambda x.e\rangle \tag{ABS}$$

$$\frac{E \vdash e_1 \to c(v_1, \ldots, v_n) \qquad E[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \vdash e_2 \to v}{E \vdash \mathit{case}(e_1,\ c(x_1, \ldots, x_n) \Rightarrow e_2,\ y \Rightarrow e_3) \to v} \tag{CASE-1}$$

$$\frac{E \vdash e_1 \to v' \qquad E[y \mapsto v'] \vdash e_3 \to v}{E \vdash \mathit{case}(e_1,\ c(x_1, \ldots, x_n) \Rightarrow e_2,\ y \Rightarrow e_3) \to v} \quad (v \neq c(\ldots)) \tag{CASE-2}$$

$$\frac{E[x \mapsto \langle E, \mathit{fix}\ x.e\rangle] \vdash e \to v}{E \vdash \mathit{fix}\ x.e \to v} \tag{FIX}$$

Figure 1: Operational Semantics for the Simple Language.

$$\mathcal{E} \vdash x \rightsquigarrow V \quad (V = \{v \in \mathcal{E}(x) : v \neq \langle E, \mathit{fix}\ y.e\rangle\}) \tag{VAR-1}$$

$$\frac{\mathcal{E} \vdash \mathit{fix}\ y.e \rightsquigarrow V}{\mathcal{E} \vdash x \rightsquigarrow V} \quad (\langle E, \mathit{fix}\ y.e\rangle \in \mathcal{E}(x)) \tag{VAR-2}$$

$$\frac{\mathcal{E} \vdash e_1 \rightsquigarrow V_1 \qquad \mathcal{E} \vdash e_2 \rightsquigarrow V_2 \qquad \mathcal{E} \vdash e \rightsquigarrow V_3}{\mathcal{E} \vdash e_1\ e_2 \rightsquigarrow V_3} \quad (\langle E, \lambda x.e\rangle \in V_1) \tag{APP}$$

$$\frac{\mathcal{E} \vdash e_1 \rightsquigarrow V_i,\ i = 1..n}{\mathcal{E} \vdash c(e_1, \ldots, e_n) \rightsquigarrow c(V_1, \ldots, V_n)} \tag{CONST}$$

$$\mathcal{E} \vdash \lambda x.e \rightsquigarrow \langle \mathcal{E}, \lambda x.e\rangle \tag{ABS}$$

$$\frac{\mathcal{E} \vdash e_1 \rightsquigarrow V_1 \qquad \mathcal{E} \vdash e_2 \rightsquigarrow V_2}{\mathcal{E} \vdash \mathit{case}(e_1,\ c(x_1, \ldots, x_n) \Rightarrow e_2,\ y \Rightarrow e_3) \rightsquigarrow V_2} \quad (\exists v \in V_1\ s.t.\ v = c(\ldots)) \tag{CASE-1}$$

$$\frac{\mathcal{E} \vdash e_1 \rightsquigarrow V_1 \qquad \mathcal{E} \vdash e_3 \rightsquigarrow V_3}{\mathcal{E} \vdash \mathit{case}(e_1,\ c(x_1, \ldots, x_n) \Rightarrow e_2,\ y \Rightarrow e_3) \rightsquigarrow V_3} \quad (\exists v \in V_1\ s.t.\ v \neq c(\ldots)) \tag{CASE-2}$$

$$\frac{\mathcal{E} \vdash e \rightsquigarrow V}{\mathcal{E} \vdash \mathit{fix}\ x.e \rightsquigarrow V} \tag{FIX}$$

Figure 2: Set-Based Operational Semantics.

To obtain a safe approximation, a set of local safety conditions must be satisfied. For rule APP, the required condition on $\mathcal{E}$ is $V_2 \subseteq \mathcal{E}(x)$. Similar conditions can be given for the other rules involving binding. Note that it is not appropriate to just add these conditions as side conditions to the respective rules, since side conditions have the effect of *reducing* the number of possible derivations (and therefore reducing the set $\{v : \mathcal{E} \vdash e_0 \rightsquigarrow v\}$). Instead, we require that whenever one of the potentially unsafe rules is applied in a derivation, an appropriate safety condition is satisfied. To formalize this, define that $\mathcal{E}$ is *safe* with respect to a closed term $e_0$ if every derivation of the form $\mathcal{E} \vdash e_0 \rightsquigarrow V$ satisfies the following four conditions (we follow the notation established in Figure 2):

1. In every use of APP, $V_2 \subseteq \mathcal{E}(x)$.
2. In every use of CASE-1, if $c(v_1, \ldots, v_n) \in V_1$ then $v_i \in \mathcal{E}(x_i)$, $i = 1..n$.
3. In every use of CASE-2, if $v \in V_1$ and $v \neq c(\cdots)$ then $v \in \mathcal{E}(y)$.
4. In every use of FIX, $\langle \mathcal{E}, \mathit{fix}\ x.e \rangle \subseteq \mathcal{E}(x)$.

Importantly, safety implies soundness in the following sense:

**Theorem 1 (Soundness)** *If $\mathcal{E}$ is safe wrt a closed term $e_0$ then $\{v : \vdash e_0 \rightarrow v\} \subseteq \{v \in V : \mathcal{E} \vdash e_0 \rightsquigarrow v\}$.*

**Proof:** The proof follows by structural induction on the subderivations of the derivation $\vdash e \rightarrow v$. The induction hypothesis must be strengthened slightly to include a simple property about the closures that may be encountered. □

In essence, this proves that if we guess $\mathcal{E}$ so that it is safe, then the set-based operational semantics provides a sound approximation of the execution of a term. However, given a term $e_0$, there are many correct choices for $\mathcal{E}$, and these give rise to different approximations of $e_0$. The following proposition implies that, given $e_0$, there is a canonical choice for $\mathcal{E}$, and that this choice gives rise to the most accurate approximation of $e_0$. First, define that the intersection of set environments $\mathcal{E}_1$ and $\mathcal{E}_2$, denoted $\mathcal{E}_1 \cap \mathcal{E}_2$, is given by: $(\mathcal{E}_1 \cap \mathcal{E}_2)(x) \stackrel{\text{def}}{=} \mathcal{E}_1(x) \cap \mathcal{E}_2(x)$, provided $\mathcal{E}_1(x)$ and $\mathcal{E}_2(x)$ are both defined. Then:

**Proposition 1 (Minimality)** *If $\mathcal{E}_1$ and $\mathcal{E}_2$ are safe wrt a closed term $e_0$, then so is $\mathcal{E}_1 \cap \mathcal{E}_2$. Moreover, $\mathcal{E}_1 \cap \mathcal{E}_2 \vdash e_0 \rightsquigarrow v$ implies $\mathcal{E}_1 \vdash e_0 \rightsquigarrow v$ and $\mathcal{E}_2 \vdash e_0 \rightsquigarrow v$.*

**Proof:** The proof here is straightforward and follows from the observation that any derivation $\mathcal{E}_1 \cap \mathcal{E}_2 \vdash e_0 \rightsquigarrow V$ can be replayed to give isomorphic derivations $\mathcal{E}_1 \vdash e_0 \rightsquigarrow V$ and $\mathcal{E}_2 \vdash e_0 \rightsquigarrow V$. □

This motivates the following definition[5].

**Definition 1 (Set-Based Approximation)** *Let $e_0$ be a closed term. Let $\mathcal{E}_{min}$ be the least set environment that is safe wrt $e_0$. The set-based approximation of $e_0$, denoted $sba(e_0)$, is defined by:*

$$sba(e_0) \stackrel{\text{def}}{=} \{v \in V : \mathcal{E}_{min} \vdash e_0 \rightsquigarrow V\} \quad \square$$

[5] It is possible to give a direct definition of set-based approximation, which avoids the minimization over $\mathcal{E}$. However such a definition is substantially more complex.

To summarize, the set-based operational semantics approximates the execution of a term by collapsing all environments into one single set environment. No other form of approximation is employed. In particular, no use is made of abstract domains (such as those commonly employed in abstract-interpretation styles of program analysis [9]). We remark that the results of the analysis are typically infinite sets of values, and that we make no *a priori* requirement that these sets be finitely presentable.

## 4 Main Result

We now present the main result of the paper, which is an algorithm for computing $sba(e_0)$ for any closed term $e_0$. The structure of the algorithm is as follows. First, we construct *set constraints* corresponding to the input term $e_0$. In essence, these constraints express relationships between sets of values in such a way that a model of the constraints corresponds to the set-based execution of $e_0$ in some safe set environment $\mathcal{E}$. Importantly, the least model of these constraints corresponds to execution in the smallest safe set environment, and hence to $sba(e_0)$. The second part of the algorithm is a simplification procedure for set constraints. In essence, this algorithm constructs an explicit representation of the least model of the input set constraints. This representation is in the form of a regular tree grammar. Note that no assumptions have been made about the adequacy of regular tree grammars. The fact that the least model of the set constraints (and hence $sba(e_0)$) can be represented using regular tree grammars is a corollary of the correctness proof of the algorithm.

Before describing the form of the set constraints employed by the algorithm, we first note that the environment part of closures in $sba(e_0)$ is essentially redundant. In particular, if $\mathcal{E}$ is the least set environment that is safe with respect to a closed term $e_0$, and if $sba(e_0)$ contains a closure $\langle E, \lambda x.e \rangle$, then $sba(e_0)$ must in fact contain all closures of the form $\langle E', \lambda x.e \rangle$ such that $E' \in \mathcal{E}$. This is because the set-based operational semantics collapses all environments into the single set environment $\mathcal{E}$, and moreover, the only closures generated during the set-based execution are via the (ABS) rule. In the computation of $sba(e_0)$, it is convenient to drop the redundant environment information in closures[6]. More formally, define an operator $\|v\|$ on values $v$, which forgets the environment part of closures, as follows:

$$\|v\| = \begin{cases} c(\|v_1\|, \ldots, \|v_n\|) & \text{if } v \text{ is } c(v_1, \ldots, v_n) \\ \lambda x.e & \text{if } v \text{ is } \langle E, \lambda x.e \rangle \end{cases}$$

The algorithm presented in this section computes a representation of the set $\|sba(e_0)\| = \{\|v\| : v \in sba(e_0)\}$.

### Set Constraints

The use of set constraints for analysis of programs dates back to the early works by Reynolds [29], and Jones and Muchnick [19], which employ constraints involving projection. The calculus of set constraints was first defined and studied in a general setting by Jaffar and the present author [13]. [13] also contained a decision procedure for a class of set constraints involving projection and intersection. Later works have provided algorithms for different classes of set

[6] We note that this can be recovered if needed, although it is not completely trivial to do so since values and environments are mutually dependent.

constraints (Aiken and Wimmers [1] have dealt with complementation and intersection; Jaffar and the present author have dealt with set constraint operators that are designed for analyzing logic programs and imperative programs [10, 12], and combinations of set constraint techniques and abstract interpretation techniques [14]), as well as generalizing previous results (Bachmair, Ganzinger and Waldmann [5] establish a connection between certain kinds of set constraints and a fragment of logic shown decidable by Löwenheim, and in the process give a simple proof of decidability of a class of set constraints that subsumes the earlier results in [1] and [13]).

We extend the basic set constraint calculus of [13] by adding operations to model function application and case statements. The form and meaning of these constraints is defined in the context of some given closed term $e_0$. We assume a fixed infinite class of *set variables*; set variables shall be denoted $\mathcal{W}, \mathcal{X}, \mathcal{Y}, \mathcal{Z}$. We distinguish two special disjoint subclasses of set variables. First, for each program variable $x$ in $e_0$, there is a distinct set variable $\mathcal{X}_x$ which shall be used to capture all of the values for the program variable $x$. Second, for each abstraction $\lambda x.e$ appearing in $e_0$, there is a distinct set variable $ran(\lambda x.e)$, the "range" of $\lambda x.e$, which shall be used to capture all of the values returned by applications of $\lambda x.e$ during execution. Now, in the context of the given term $e_0$, we define that a *set expression* ($se$) is either a set variable, an abstraction $\lambda x.e$ that appears in $e_0$, or of one of the forms $c(se_1, \ldots, se_2)$, $apply(se_1, se_2)$, $case(se_1, c(\mathcal{X}_1, \ldots, \mathcal{X}_n) \Rightarrow se_2, \mathcal{Y} \Rightarrow se_3)$ or $ifnonempty(se_1, se_2)$ (this is essentially the $\sigma_*$ operation of Reynolds [29]; it shall be employed later in the paper). The first form is used to model execution of expressions $c(e_1, \ldots, e_n)$, the second form models application, the third is for case statements, and the last is used to reason about emptiness. A *set constraint* is an expression of the form $\mathcal{X} \supseteq se$, and a *conjunction* $\mathcal{C}$ of set constraints is a finite collection of set constraints.

We now define the meaning of the set constraints. In essence, set expressions shall be interpreted as sets of values with the environment component of closures removed. Specifically, a *set constraint value* (sc-value) is either an abstraction $\lambda x.e$ that appears in $e_0$, or of the form $c(v_1, \ldots, v_n)$ where each $v_i$ is an sc-value. An *interpretation* is a mapping from each set variable into a set of sc-values. Such an interpretation is extended to map set expressions to sets of sc-values as follows:

1. $\mathcal{I}(c(se_1, \ldots, se_n)) = \{c(v_1, \ldots, v_n) : v_i \in \mathcal{I}(se_i)\}$
2. $\mathcal{I}(\lambda x.e) = \{\lambda x.e\}$
3. $\mathcal{I}(ifnonempty(se_1, se_2)) = \begin{cases} \mathcal{I}(se_2) & \text{if } \mathcal{I}(se_1) = \{\} \\ \{\} & \text{otherwise} \end{cases}$
4. $\mathcal{I}(apply(se_1, se_2)) = \left\{ v : \begin{array}{c} \lambda x.e \in \mathcal{I}(se_1) \\ \mathcal{I}(se_2) \neq \{\} \\ v \in \mathcal{I}(ran(\lambda x.e)) \end{array} \right\}$
   provided $\lambda x.e \in \mathcal{I}(se_1)$ implies $\mathcal{I}(se_2) \subseteq \mathcal{I}(\mathcal{X}_x)$
5. $\mathcal{I}(case(se_1, c(\mathcal{X}_1, \ldots, \mathcal{X}_n) \Rightarrow se_2, \mathcal{Y} \Rightarrow se_3)) = S_1 \cup S_2$
   provided:
   $(i)$ $S_1 = \{v : v \in \mathcal{I}(se_2) \wedge \exists v' \in \mathcal{I}(se_1)\ s.t.\ v' = c(\ldots)\}$
   $(ii)$ $S_2 = \{v : v \in \mathcal{I}(se_3) \wedge \exists v' \in \mathcal{I}(se_1)\ s.t.\ v' \neq c(\ldots)\}$
   $(iii)$ if $c(v_1, \ldots, v_n) \in \mathcal{I}(se_1)$ then $v_i \in \mathcal{I}(\mathcal{X}_i)$, $i = 1..n$
   $(iv)$ if $v \in \mathcal{I}(se_1) \ \wedge\ v \neq c(\ldots)$ then $v \in \mathcal{I}(\mathcal{Y})$

Note that the above interpretation of set expressions is somewhat unusual, because in parts 4 and 5 of the definition, the set expressions themselves impose restrictions on $\mathcal{I}$. If these conditions are not met, then the interpretation of the expression is undefined. An interpretation $\mathcal{I}$ is a *model* of a conjunction of constraints $\mathcal{C}$ if, for each constraint $\mathcal{X} \supseteq se$, it is the case that $\mathcal{I}(se)$ is defined and $\mathcal{I}(\mathcal{X}) \supseteq \mathcal{I}(se)$. It is easy to verify a model intersection property for the set constraints used in this paper, and it follows that a conjunction $\mathcal{C}$ of constraints possesses a least model, denoted $lm(\mathcal{C})$, where models are ordered as follows: $\mathcal{I}_1 \supseteq \mathcal{I}_2$ if $\mathcal{I}_1(\mathcal{X}) \supseteq \mathcal{I}_2(\mathcal{X})$, for all set variables $\mathcal{X}$.

### Constructing Set Constraints

The construction of set constraints from a term is described in Figure 3. (Strictly speaking, this is a somewhat simplified version – the complete version appears in the Appendix.) In the rules (APP), (CONST), (ABS) and (CASE), the variable $\mathcal{Y}$ is intended to be a new set variable that is not used in any other part of the derivation. Using these rules, we define

**Definition 2** *Let $e_0$ be a closed term, then $\mathcal{SC}(e_0)$ is the pair $(\mathcal{X}, \mathcal{C})$ such that $e_0 \triangleright (\mathcal{X}, \mathcal{C})$.* □

To illustrate the construction of the constraints, consider again the term $e_0 = e_1\ e_2$ where $e_1$ is $\lambda f.c(f\ a, f\ b)$, $e_2$ is $\lambda x.x$ and $a$, $b$ and $c$ are constants. For this term, we derive $e_0 \triangleright (\mathcal{X}_1, \mathcal{C})$ where $\mathcal{C}$ consists of the constraints

$$\begin{array}{lll} \mathcal{X}_1 \supseteq apply(\mathcal{X}_2, \mathcal{X}_3) & \mathcal{X}_4 \supseteq apply(\mathcal{X}_f, a) & ran(e_1) \supseteq c(\mathcal{X}_4, \mathcal{X}_5) \\ \mathcal{X}_2 \supseteq e_1 & \mathcal{X}_5 \supseteq apply(\mathcal{X}_f, b) & ran(e_2) \supseteq \mathcal{X}_x \\ \mathcal{X}_3 \supseteq e_2 & & \end{array}$$

In $lm(\mathcal{C})$, $\mathcal{X}_1 = ran(e_1) = \{c(a,b), c(b,a), c(a,a), c(b,b)\}$, $\mathcal{X}_2 = \{e_1\}$, $\mathcal{X}_3 = \{e_2\}$, $\mathcal{X}_4 = \mathcal{X}_5 = ran(e_2) = \mathcal{X}_x = \{a, b\}$

For presentational simplicity, the constraint construction given in Figure 3 does not completely correspond to $sba(e_0)$. To see this, consider the term $e_0 = e_1\ e_2$ where $e_1$ is $\lambda f.((\lambda u.f\ a)(\lambda w.f\ b))$ and $e_2$ is $\lambda x.x$. The least $\mathcal{E}$ that is safe with respect to $e_0$ maps $f$ into $\{\lambda x.x\}$, $u$ into $\{\lambda w.f\ b\}$ and $x$ into $\{a\}$, and $sba(e_0)$ is $\{a\}$. However, the set constraint construction procedure traverses *all* subexpression of $e_0$. Hence $\mathcal{SC}(e_0)$ contains the set expressions $apply(\mathcal{X}_f, a)$ and $apply(\mathcal{X}_f, b)$. As a result, $\mathcal{X}_x$ must contain both $a$ and $b$, and so the execution of $e_0$ is approximated by $\{a, b\}$. The problem is that the term $\lambda w.f\ b$ is never "executed" under the set-based semantics, but is traversed by the set constraint construction process. To rectify this situation, the constraint construction must be such that if $\lambda x.e$ appears in $e_0$, then the constraints constructed for $e$ are vacuously satisfied whenever $\mathcal{X}_x$ (the set of values for $x$) is empty. The complete constraint construction procedure appears in the Appendix. The correspondence between $sba(e_0)$ and $\mathcal{SC}(e_0)$ is given by the following Lemma[7]:

**Lemma 1** *Let $e_0$ be a closed term, let $\mathcal{SC}(e_0)$ be $(\mathcal{X}, \mathcal{C})$ and let $\mathcal{I}_{lm} = lm(\mathcal{C})$. Then $\mathcal{I}_{lm}(\mathcal{X}) = \|sba(e_0)\|$.* □

**Proof Sketch:** The proof is fairly lengthly and consists of two main parts. The first part involves modifying the

[7] We note that Lemma 1 holds using the constraint construction process described in Figure 3 if the following condition is satisfied: the least set environment $\mathcal{E}$ that is safe wrt $e_0$ is such that $\mathcal{E}(x) \neq \{\}$ for all $x$.

$$x \ \triangleright \ (\mathcal{X}_x, \ \{\}) \qquad \text{(VAR)}$$

$$\frac{e_1 \ \triangleright \ (\mathcal{X}_1, \ \mathcal{C}_1) \qquad e_2 \ \triangleright \ (\mathcal{X}_2, \ \mathcal{C}_2)}{e_1 \ e_2 \ \triangleright \ (\mathcal{Y}, \{\mathcal{Y} \supseteq \mathit{apply}(\mathcal{X}_1, \mathcal{X}_2)\} \cup \mathcal{C}_1 \cup \mathcal{C}_2)} \qquad \text{(APP)}$$

$$\frac{e_i \ \triangleright \ (\mathcal{X}_i, \mathcal{C}_i), \ i = 1..n}{c(e_1, \ldots, e_n) \ \triangleright \ (\mathcal{Y}, \{\mathcal{Y} \supseteq c(\mathcal{X}_1, \ldots, \mathcal{X}_n)\} \cup C_1 \cup \ldots \cup \mathcal{C}_n)} \qquad \text{(CONST)}$$

$$\frac{e \ \triangleright \ (\mathcal{X}, \ \mathcal{C})}{\lambda x.e \ \triangleright \ (\mathcal{Y}, \{\mathcal{Y} \supseteq \lambda x.e, \mathit{ran}(\lambda x.e) \supseteq \mathcal{X}\} \cup \mathcal{C})} \qquad \text{(ABS)}$$

$$\frac{e_1 \ \triangleright \ (\mathcal{Z}_1, \ \mathcal{C}_1) \qquad e_2 \ \triangleright \ (\mathcal{Z}_2, \ \mathcal{C}_2) \qquad e_3 \ \triangleright \ (\mathcal{Z}_3, \ \mathcal{C}_3)}{\mathit{case}(e_1, c(x_1, \ldots, x_n) \Rightarrow e_2, y \Rightarrow e_3) \ \triangleright \ (\mathcal{Y}, \mathcal{C} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3)} \qquad \text{(CASE)}$$

where $\mathcal{C} = \{\mathcal{Y} \supseteq \mathit{case}(\mathcal{Z}_1, c(\mathcal{X}_{x_1}, \ldots, \mathcal{X}_{x_n}) \Rightarrow \mathcal{Z}_2, \mathcal{X}_y \Rightarrow \mathcal{Z}_3)\}$

$$\frac{e \ \triangleright \ (\mathcal{X}, \ \mathcal{C})}{\mathit{fix} \ x.e \ \triangleright \ (\mathcal{X}_x, \{\mathcal{X}_x \supseteq \mathcal{X}\} \cup \mathcal{C})} \qquad \text{(FIX)}$$

Figure 3: Construction of Set Constraints (simplified version)

definition of $\mathcal{E} \vdash e \rightarrow v$ so that environments are removed from closures. Call this new system $\vdash'$. The proof for this part involves showing a correspondence between $\vdash$ and $\vdash'$. The second part then relates $\vdash'$ with $\mathcal{SC}(e_0)$ by showing two relationships: (a) if $\mathcal{E}$ is the least set environment that is safe wrt $e_0$ (in $\vdash'$), then $\mathcal{E}$ can be used to define a model $\mathcal{I}$ of $\mathcal{C}$ such that $\mathcal{E} \vdash' e \rightsquigarrow v$ iff $v \in \mathcal{I}(\mathcal{X})$; and (b) if $\mathcal{I}$ is a model of $\mathcal{C}$ then we can define an $\mathcal{E}$ that is safe wrt $e_0$ such that if $\mathcal{E} \vdash' e \rightsquigarrow v$ then $v \in \mathcal{I}(\mathcal{X})$. In essence, part (a) shows that $\mathcal{I}_{lm}(\mathcal{X}) \subseteq \|sba(e_0)\|$, and part (b) shows that $\mathcal{I}_{lm}(\mathcal{X}) \supseteq \|sba(e_0)\|$. □

### Set Constraint Algorithm

We first address the issue of the output format of the algorithm. What we desire is an explicit representation of the least model of the set constraints, and specifically, of $sba(e_0)$. Since these sets are typically infinite, we must deal with finite representations of infinite sets. What is needed is a representation from which simple questions such as membership, emptiness and containment can be directly determined. The representation we use is based on a restricted form of set constraints. Specifically, define that a set expression is *atomic* if it is either an abstraction $\lambda x.e$ that appears in $e_0$, a set variable, or of the form $c(ae_1, \ldots, ae_n)$ where each $ae_i$ is atomic. A constraint is in *explicit form* if it has the form $\mathcal{X} \supseteq ae$ where $ae$ is an atomic set expression that is not a set variable ($ae$ may of course *contain* set variables). A collection of constraints is in explicit form if each constraint therein is in explicit form. If $\mathcal{C}$ is a collection of constraints, then $\mathit{explicit}(\mathcal{C})$ denotes the explicit form constraints of $\mathcal{C}$. We note that explicit form constraints can be regarded as regular tree grammars by treating set variables as non-terminals and regarding a constraint $\mathcal{X} \supseteq ae$ as a production $\mathcal{X} \Rightarrow ae$.

The simplification algorithm accepts as input a collection of constraints (such as those constructed for a closed term $e_0$) and outputs an explicit form collection of constraints that has the same least model as the input collection. The main part of the algorithm involves exhaustively applying a series of simplification steps, and this serves to add new explicit form constraints so that information about $lm(\mathcal{C})$ is incrementally transferred into the explicit part of $\mathcal{C}$. The algorithm terminates exactly when all information about $lm(\mathcal{C})$ is present in $\mathit{explicit}(\mathcal{C})$. The details of the algorithm appear in Figure 4. The phrase "add $\mathcal{X} \supseteq se$ to $\mathcal{C}$" is used to mean "add the constraint $\mathcal{X} \supseteq se$ if it does not already appear". An expression of the form $lm(\mathit{explicit}(\mathcal{C}))(\mathcal{Y}) \neq \{\}$ indicates a test which can be performed as follows: construct $\mathit{explicit}(\mathcal{C})$, and (using standard algorithms), check to see if $\mathcal{Y}$ is empty in the least model of $\mathit{explicit}(\mathcal{C})$ (analogous procedures can be found in [10, 13]).

We note that the correctness of the algorithm relies on the fact that there are no "nested" set expressions. In other words, if an expression of the form $\mathit{apply}(se_1, se_2)$ appears in the constraints, then $se_1$ and $se_2$ are both set variables, and similarly for expressions involving *ifnonempty* and *case*. It is easy to see that $\mathcal{SC}(e_0)$ satisfies this property, and it is trivial to verify that the algorithm preserves this property. The next lemma establishes the correctness of the simplification algorithm, and, combined with Lemma 1, proves Theorem 2.

**Lemma 2 (Correctness of Algorithm)**
*The algorithm terminates on input $\mathcal{C}$ and outputs explicit form constraints $\mathcal{C}'$ such that $lm(\mathcal{C}') = lm(\mathcal{C})$.*

**Proof Sketch:** Termination is straightforward to verify since the algorithm adds only constraints of the form $\mathcal{X} \supseteq ae$ where both $\mathcal{X}$ and $ae$ are expressions that already appear in the constraints. The main part of the proof is to establish that the transformation steps are complete in the sense that when no further transformation steps can be applied, then $lm(\mathcal{C}) = lm(\mathit{explicit}(\mathcal{C}))$. This is achieved by showing that when no further transformation can be applied, the interpretation $lm(\mathit{explicit}(\mathcal{C}))$ is in fact a model of $\mathcal{C}$. □

**Theorem 2** *Given a closed term $e_0$, there is an $O(n^3)$ algorithm to compute an explicit representation (which is equivalent to a regular tree grammar) of $\|sba(e_0)\|$.*

**Proof:** Let $\mathcal{SC}(e_0)$ be $(\mathcal{X}, \mathcal{C})$. By Lemma 1, $lm(\mathcal{C})$ maps $\mathcal{X}$ into $\|sba(e_0)\|$. By Lemma 2, the set constraint simplification algorithm produces collection of constraints $\mathcal{C}'$ in explicit form when input with $\mathcal{C}$. Moreover, $lm(\mathcal{C}') = lm(\mathcal{C})$.

**input** a collection $\mathcal{C}$ of set constraints;
**repeat**
  **if** $\mathcal{X} \supseteq \mathit{apply}(\mathcal{X}_1, \mathcal{X}_2)$ and $\mathcal{X}_1 \supseteq \lambda x.e$ both appear in $\mathcal{C}$ **then**
    **add** $\mathcal{X} \supseteq \mathit{ran}(\lambda x.e)$ to $\mathcal{C}$;
    **add** $\mathcal{X}_x \supseteq \mathcal{X}_2$ to $\mathcal{C}$;
  **if** $\mathcal{X} \supseteq \mathit{case}(\mathcal{Y}_1, c(\mathcal{W}_1, \ldots, \mathcal{W}_n) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3)$
      and $\mathcal{Y}_1 \supseteq c(\mathcal{Z}_1, \ldots, \mathcal{Z}_n)$ both appear in $\mathcal{C}$
      and $\mathit{lm}(\mathit{explicit}(\mathcal{C}))(\mathcal{Z}_i) \neq \{\}$, $i = 1..n$, **then**
    **add** $\mathcal{X} \supseteq \mathcal{Y}_2$ to $\mathcal{C}$;
    **add** $\mathcal{W}_i \supseteq \mathcal{Z}_i$ to $\mathcal{C}$, $i = 1..n$;
  **if** $\mathcal{X} \supseteq \mathit{case}(\mathcal{Y}_1, c(\mathcal{W}_1, \ldots, \mathcal{W}_n) \Rightarrow \mathcal{Y}_2, \mathcal{W} \Rightarrow \mathcal{Y}_3)$
      and $\mathcal{Y}_1 \supseteq c'(\mathcal{Z}_1, \ldots, \mathcal{Z}_n)$ both appear in $\mathcal{C}$, where $c' \neq c$,
      and $\mathit{lm}(\mathit{explicit}(\mathcal{C}))(\mathcal{Z}_i) \neq \{\}$, $i = 1..n$, **then**
    **add** $\mathcal{X} \supseteq \mathcal{Y}_3$ to $\mathcal{C}$;
    **add** $\mathcal{W} \supseteq c'(\mathcal{Z}_1, \ldots, \mathcal{Z}_n)$ to $\mathcal{C}$;
  **if** $\mathcal{X} \supseteq \mathit{ifnonempty}(\mathcal{Y}_1, \mathcal{Y}_2)$ appears in $\mathcal{C}$ and $\mathit{lm}(\mathit{explicit}(\mathcal{C}))(\mathcal{Y}_1) \neq \{\}$ **then**
    **add** $\mathcal{X} \supseteq \mathcal{Y}_2$ to $\mathcal{C}$;
  **if** $\mathcal{X} \supseteq \mathcal{X}'$ and $\mathcal{X}' \supseteq \mathit{ae}$ both appear in $\mathcal{C}$,
      where $\mathit{ae}$ is atomic and not a set variable, **then**
    **add** $\mathcal{X} \supseteq \mathit{ae}$ to $\mathcal{C}$;
**until** no step changes $\mathcal{C}$;
**output** $\mathit{explicit}(\mathcal{C})$;

Figure 4: **Set Constraint Simplification Algorithm**

Hence $\mathit{lm}(\mathcal{C}')(\mathcal{X}) = \mathit{lm}(\mathcal{C})(\mathcal{X}) = \|\mathit{sba}(e_0)\|$, and so $\mathcal{C}'$ provides an explicit representation of $\|\mathit{sba}(e_0)\|$. The $O(n^3)$ bound can be established as follows. First, the construction of constraints is linear in the size of $e_0$. Second, at most $n^2$ new constraints can be added by the simplification algorithm, and the cost of "adding" each new constraint (i.e. determining what other new constraints need to be added, given this constraint is added) can be bounded by $0(n)$. □

In addition, the algorithm trivially has an $O(n^2)$ space bound. We remark that this algorithm not only provides a way to compute $\mathit{sba}(e_0)$, but it also computes the least set environment that is safe wrt $e_0$.

## 5 Arrays, Continuations, Exceptions and Arithmetic

Thus far we have presented a formal development of the core ideas of set based analysis. We now informally outline the extensions we have employed for dealing with arrays, exceptions and continuations. As outlined in the introduction, the set-based treatment of arrays ignores dependencies between subscripts and values. That is, an array is treated as a set of values such that when the array is updated the new value(s) are added to this set, and when the array is accessed the set of values is returned. More concretely, for each place in the program where an array can be generated, we introduce a special distinct constant $ar$ with two associated set variables $\mathit{length}(ar)$ and $\mathit{contents}(ar)$. We also introduce two new set expressions, $\mathit{contentsof}(se)$ and $\mathit{update}(se_1, se_2)$. In essence, the first denotes the union of the sets $\mathit{contents}(ar)$ such that $ar$ is an element of $se$. The second is either (i) the empty set if either $se_1$ or $se_2$ is empty, (ii) the singleton set containing the unit value provided $se_1$ and $se_2$ are non-empty and $\mathit{contents}(ar) \supseteq se_2$ for all $ar$ in $se_1$, or (iii) is undefined otherwise. The first three rules in Figure 5 are suggestive[8] of how constraints are constructed for programs involving arrays. In the first rule, $ar$ is a new constant.

Continuations are also modeled by introducing a new constant $\mathit{cont}$ for each *callcc* appearing in a program. Each new constant has an associated set variable $\mathit{contents}(\mathit{cont})$. In essence, this records the values that are thrown to the continuation. In effect, the constant $\mathit{cont}$ passes into the term $e$ a reference to the program point at which the *callcc* occurred (in fact it passes down the set variable corresponding to this point). The set expression $\mathit{throw}(se_1, se_2)$ is either (i) the empty set provided that $\mathit{contents}(\mathit{cont}) \supseteq se_2$ for each $\mathit{cont}$ in $se_1$, or (ii) undefined otherwise.

Exceptions are modeled by introducing a distinct new set variable $\mathcal{EXC}$ to capture all of the exceptions that are raised during program execution. We note exceptions could be more accurately treated by introducing a new exception variable for each expression. This would provide better "separation" of the exceptions raised by different parts of a program, but at the cost of introducing more constraints. We are currently investigating this tradeoff.

The treatment of arithmetic is described in detail in [11]. The essential idea is to compute descriptions of *how* arithmetic values are obtained. These descriptions are essentially terms built from arithmetic operations and integers. [11] also describes how the arithmetic part of the analysis can be applied to the problem of removing array bounds checks. Preliminary results indicate that this leads to useful speadups. For example, when the set-based analysis implementation is applied to itself[9], we were able to infer that key array updates and subscripts were guaranteed to be safe

[8] In particular, they are a simplification of the actual rules in the sense that Figure 3 simplifies Figure 6.

[9] We note that this implementation makes substantial use of arrays, higher order functions (for example, functions are often stored in lists and arrays) and exceptions.

$$\frac{\triangleright\ e_1 : (\mathcal{X}_1,\ \mathcal{C}_1) \qquad \triangleright\ e_2 : (\mathcal{X}_2,\ \mathcal{C}_2)}{\triangleright\ array(e_1, e_2) : (\mathcal{Y}, \{\mathcal{Y} \supseteq ar,\ contents(ar) \supseteq se_1,\ length(ar) \supseteq se_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2)} \quad \text{(ARRAY)}$$

$$\frac{\triangleright\ e_1 : (\mathcal{X}_1,\ \mathcal{C}_1) \qquad \triangleright\ e_2 : (\mathcal{X}_2,\ \mathcal{C}_2)}{\triangleright\ e_1\ sub\ e_2 : (\mathcal{Y}, \{\mathcal{Y} \supseteq contentsof(\mathcal{X}_1)\} \cup \mathcal{C}_1 \cup \mathcal{C}_2)} \quad \text{(SUBSCRIPT)}$$

$$\frac{\triangleright\ e_i : (\mathcal{X}_i,\ \mathcal{C}_i),\ i = 1..3}{\triangleright\ update(e_1, e_2, e_3) : (\mathcal{Y}, \{\mathcal{Y} \supseteq update(\mathcal{X}_1, \mathcal{X}_2)\} \cup \mathcal{C}_1 \cup \mathcal{C}_2)} \quad \text{(UPDATE)}$$

$$\frac{\triangleright\ e : (\mathcal{X},\ \mathcal{C})}{\triangleright\ callcc\ x.e : (\mathcal{Y}, \{\mathcal{Y} \supseteq \mathcal{X}, \mathcal{Y} \supseteq contents(cont), \mathcal{X}_x \supseteq cont\} \cup \mathcal{C})} \quad \text{(CALLCC)}$$

$$\frac{\triangleright\ e_1 : (\mathcal{X}_1,\ \mathcal{C}_1) \qquad \triangleright\ e_2 : (\mathcal{X}_2,\ \mathcal{C}_2)}{\triangleright\ throw(e_1, e_2) : (\mathcal{Y}, \{\mathcal{Y} \supseteq throw(\mathcal{X}_1, \mathcal{X}_2)\} \cup \mathcal{C}_1 \cup \mathcal{C}_2)} \quad \text{(THROW)}$$

$$\frac{\triangleright\ e : (\mathcal{X},\ \mathcal{C})}{\triangleright\ raise\ e : (\mathcal{Y}, \{\mathcal{E}\mathcal{X}\mathcal{C} \supseteq \mathcal{X}\} \cup \mathcal{C})} \quad \text{(RAISE)}$$

$$\frac{\triangleright\ e_1 : (\mathcal{X}_1,\ \mathcal{C}_1) \qquad \triangleright\ e_2 : (\mathcal{X}_2,\ \mathcal{C}_2)}{\triangleright\ e_1\ handle\ (\lambda x.e_2) : (\mathcal{Y}, \{\mathcal{Y} \supseteq \mathcal{X}_1, \mathcal{Y} \supseteq \mathcal{X}_2, \mathcal{X}_x \supseteq \mathcal{E}\mathcal{X}\mathcal{C}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2)} \quad \text{(HANDLE)}$$

Figure 5: Construction of Set Constraints for Arrays, Continuations and Exceptions

– exploiting this fact led to performance improvements of about 6% – 13%, depending on the program analyzed).

## 6 Implementation

An implementation of set-based analysis for ML has been developed over the last two years. The system is build on top of the SML-NJ compiler. Starting with the LAMBDA intermediate representation of a program, our system incrementally builds and solves corresponding set constraints. Many of the set constraints that are generated are trivial, and so an important part of the effort to make the analyzer efficient was directed at ensuring that such constraints are solved "on-the-fly", and are never explicitly generated.

An important aspect of the implementation is "polyvariance" (the analysis analogue of polymorphism). That is, the implementation provides a mechanism to construct different "versions" of functions. In essence this is done by constraint duplication. However, for efficiency reasons, we wish to avoid multiple passes over the input LAMBDA expression, and instead we first convert the LAMBDA expression into a compact internal format, from which multiple copies of constraints can be rapidly generated. A key aspect of polyvariance is how to control the generation of different versions of functions. One approach is to use the type information of a program (e.g. if a function is polymorphic, then it is likely to be useful to treat it as a polyvariant function). However, a goal of our implementation was to provide a generic analysis tool for functional programs, and so we did not want to commit to a typed language. Instead we chose a scheme in which the program is analyzed twice – the first pass is a "monovariant" analysis, and the second pass uses information from the first to control a polyvariant analysis.

The following table presents some preliminary empirics for the implementation. We use five programs. The first program is the intmap structure from the SML-NJ compiler, which implements a mapping from integers to integers. The second models the game life, and is written in an applicative (rather than imperative) style. The third is taken from a structure that implements an efficient form of byte array copying that is part of the FOX project's TCP/IP network protocol software. The forth is the lexer generator from the ml-lex/ml-yacc collection. The fifth is the core part of the set-based analysis implementation. All times are in seconds on an PMAX 5000/200 with 64M and running Mach and using version 0.93 of SML-NJ. For each benchmark, the number of "equations" generated is given[10] (this excludes constraints that are solved on-the-fly). Phase I is the monovariant analysis. Phase II is the polyvariant analysis (which uses information from phase I).

| | Phase I | | Phase II | |
|---|---|---|---|---|
| | time(s) | eqns. | time(s) | eqns. |
| intmap (105 lines) | 0.25 | 1053 | 0.30 | 1262 |
| life (150 lines) | 0.56 | 1461 | 2.19 | 12799 |
| copy (177 lines) | 0.23 | 1459 | 0.28 | 1484 |
| lexgen (1170 lines) | 2.38 | 6600 | 4.41 | 16674 |
| solver (2765 lines) | 6.69 | 17140 | - | - |

For some examples, the two phases have almost equal cost, but in other cases the difference is substantial. This reflects both the extent of polymorphism in the example and the performance of the heuristics that inspect the information from the first phase and control polyvariance in the second phase. The idea of these heuristics is to identify those functions that will definitely not benefit from duplication (for example, there is no reason to duplicate functions from unit to unit); all remaining functions are marked for duplication. In particular, all functions that are identified as polymorphic by the ML type system should be marked for duplication (however, note that even non-polymorphic functions can benefit from duplication).

The heuristic currently used is very simplistic and can result in many unnecessary duplications (in fact this is the reason that the results for phase II of the solver example

[10] The implementation collects all constraints with the same left-hand-side variable together, and the resulting object is effectively an equation.

are not available at this time). We expect substantial improvement in the running time of polyvariant analysis as the control of constraint duplication in further developed. Early experience suggests that, for most programs, it should be possible for phase II to run within a small constant factor of the phase I analysis.

We remark that the control flow information computed by the phase I analysis can be roughly compared to 0CFA in the terminology of [25]; the effect of phase II is go beyond 0CFA in a controlled way that avoids the combinatorial explosions of 1CFA. An underlying philosophy of our system is that the core part of an analysis should be a simple, intuitive, efficient and have well understood behavior. Then, the notion of polyvariance or polymorphism should be addressed by building on top of this core. By treating polyvariance/polymorphism as an orthogonal concept rather than "built-in" to the basic analysis, we achieve greater modularity and flexibility, as well as improved control over the cost of the analysis.

## 7 Conclusion

Starting with the simple intuition of treating program variables as sets, we have developed a powerful, general and flexible analysis for higher-order call-by-value functional languages. The contributions of the paper lie in three areas. First, we have given a very direct and appealing connection between a program's set-based approximation (which is what our algorithm computes), and its underlying operational semantics. Second, we have presented an algorithm that combines (a) an accurate treatment of data-structures, (b) modeling of side-effecting operations and (c) efficiency. Third, we have described an implementation of this analysis which provides evidence that the set-based analysis approach is practical. Applications of the implementation are being persued in a number of different areas, including partial evaluation [21] and array bounds checking [11].

## Acknowledgments

Thanks to Alex Aiken, Olivier Danvy, Matthias Felleisen, Bob Harper, Neil Jones, Peter Lee, Karoline Malmkjær and David Tarditi for comments and discussion about this work.

## References

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

[2] A. Aiken and E. Wimmers, "Type Inclusion and Type Inference", *Proc. $6^{th}$ ACM Conf. on Functional Programming and Computer Architecture*, Copenhagen, pp. 31–41, June 1993.

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

[4] A. Appel, "Compiling with Continuations", Cambridge University Press, 1992.

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

[6] E. Biagioni, R. Harper, P. Lee, and B. Milnes, "Signatures for a network protocol stack: A systems application of Standard ML", *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, Orlando, Florida, June 1994, to appear.

[7] R. Cartwright and M. Fagan, "Soft Typing", *Proc. 1991 ACM Conf. on Programming Language Design and Implementation*, Toronto, pp. 278-292, June 1991.

[8] C. Consel and O. Danvy, "Tutorial Notes on Partial Evaluation", *Proc. $20^{th}$ ACM Symp. on Principles of Programming Languages*, Charleston, pp. 493–501, January 1993.

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

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

[11] N. Heintze, "Set-Based Program Analysis and Arithmetic", Technical Report, School of Computer Science, Carnegie Mellon University, December 1993.

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

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

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

[15] F. Henglein, "Efficient Type Inference for Higher-Order Binding-Time Analysis", *Proceedings $5^{th}$ ACM-FPCA*, Cambridge MA, LNCS 523, pp. 448–472, August 1991.

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

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

[18] N. Jones, C. Gomard and P. Sestoft, "Partial Evaluation and Automatic Program Generation", "Prentice-Hall International", 1993.

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

[20] N. Jones and S. Muchnick, "Complexity of Flow Analysis, Inductive Assertion Synthesis, and a Language due to Dijkstra", *Proc. $21^{st}$ IEEE-FOCS*, Syracuse, pp. 185–190, October 1980. (Also in, *Program Flow Analysis: Theory and Applications*, N. Jones and S. Muchnick (Eds.), Prentice-Hall, 1981.)

[21] K. Malmkjær, N. Heintze and O. Danvy, "ML Partial Evaluation using Set-Based Analysis", submitted for publication.

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

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

[24] F. Nielson and H. Nielson, "Two-Level Functional Languages", Cambridge University Press, Vol 34, Cambridge Tracts in Theoretical Computer Science, 1992.

[25] O. Shivers, "Control Flow Analysis in Scheme", *Proc. 1988 ACM Conf. on Programming Language Design and Implementation*, Atlanta, pp. 164–174, June 1988.

[26] J. Palsberg, Private Communication, November 1993.

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

[28] J. Palsberg and M. Schwartzbach, "Object-Oriented Type Systems", John Wiley & Sons, to appear, 1993.

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

[30] A. Wright and R. Cartwright, "A Practical Soft Type System for Scheme", *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, Orlando, Florida, June 1994, to appear.

## Appendix: Construction of Set Constraints

Figure 6 presents the complete details of the constructions of set constraints for a term. The main difference between Figure 6 and Figure 3 is that the relation $\mathcal{Z} \vdash e \triangleright (se, \mathcal{C})$ recursively passes down a set variable which is empty if the expression under consideration is never called, and is nonempty otherwise. The key property of the relation $\mathcal{Z} \vdash e \triangleright (se, \mathcal{C})$ is that if $\mathcal{Z}$ is empty then $\mathcal{C}$ is vacuously true, and if $\mathcal{Z}$ is nonempty, then $se$ and $\mathcal{C}$ are equivalent to those constructed using the simpler deductive system in Figure 3. We now define $\mathcal{SC}(e_0)$ as follows: if $\mathcal{Z}$ is a new set variable and $\mathcal{Z} \vdash e \triangleright (\mathcal{X}, \mathcal{C})$, then $\mathcal{SC}(e_0)$ is the pair $(\mathcal{X}, \{\mathcal{Z} \supseteq e\} \cup \mathcal{C})$ where $e$ is some arbitrary sc-value. Note that all sc-values are set expressions and that the choice of $e$ is arbitrary – its only purpose is to force the variable $\mathcal{Z}$ to be nonempty, since otherwise the constraints $\mathcal{C}$ would be vacuously true.

$$\mathcal{Z} \;\vdash\; x \;\triangleright\; (\mathcal{X}_x,\ \{\}) \qquad \text{(VAR)}$$

$$\frac{\mathcal{X}_x \;\vdash\; e \;\triangleright\; (\mathcal{X},\ \mathcal{C})}{\mathcal{Z} \;\vdash\; \lambda x.e \;\triangleright\; (\mathcal{Y},\ \{\mathcal{Y} \supseteq \lambda x.e,\ ran(\lambda x.e) \supseteq \mathcal{X}\} \cup \mathcal{C})} \qquad \text{(ABS)}$$

$$\frac{\mathcal{Z} \;\vdash\; e_1 \;\triangleright\; (\mathcal{X}_1,\ \mathcal{C}_1) \qquad \mathcal{Z} \;\vdash\; e_2 \;\triangleright\; (\mathcal{X}_2,\ \mathcal{C}_2)}{\mathcal{Z} \;\vdash\; e_1\ e_2 \;\triangleright\; (\mathcal{Y},\ \{\mathcal{Y} \supseteq apply(\mathcal{Y}', \mathcal{X}_2),\ \mathcal{Y}' \supseteq ifnonempty(\mathcal{Z}, \mathcal{X}_1)\} \cup \mathcal{C}_1 \cup \mathcal{C}_2)} \qquad \text{(APP)}$$

$$\frac{\mathcal{Z} \;\vdash\; e_i \;\triangleright\; (\mathcal{X}_i, \mathcal{C}_i),\ i = 1..n}{\mathcal{Z} \;\vdash\; c(e_1, \ldots, e_n) \;\triangleright\; (\mathcal{Y},\ \{\mathcal{Y} \supseteq c(\mathcal{X}_1, \ldots, \mathcal{X}_n)\} \cup C_1 \cup \ldots \cup \mathcal{C}_n)} \qquad \text{(CONST)}$$

$$\frac{\mathcal{Z} \;\vdash\; e_1 \;\triangleright\; (\mathcal{X}_1,\ \mathcal{C}_1) \qquad \mathcal{Z} \;\vdash\; e_2 \;\triangleright\; (\mathcal{X}_2,\ \mathcal{C}_2) \qquad \mathcal{Z} \;\vdash\; e_3 \;\triangleright\; (\mathcal{X}_3,\ \mathcal{C}_3)}{\mathcal{Z} \;\vdash\; case(e_1, c(x_1, \ldots, x_n) \Rightarrow e_2, y \Rightarrow e_3) \;\triangleright\; (\mathcal{Y},\ \mathcal{C} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3)} \qquad \text{(CASE)}$$

where $\mathcal{C} = \{\mathcal{Y} \supseteq case(\mathcal{Y}', c(\mathcal{X}_{x_1}, \ldots, \mathcal{X}_{x_n}) \Rightarrow \mathcal{Z}_2, \mathcal{X}_y \Rightarrow \mathcal{Z}_3),\ \ \mathcal{Y}' \supseteq ifnonempty(\mathcal{Z}, \mathcal{X}_1)\}$

$$\frac{\mathcal{Z} \;\vdash\; e \;\triangleright\; (\mathcal{X},\ \mathcal{C})}{\mathcal{Z} \;\vdash\; fix\, x.e \;\triangleright\; (\mathcal{X}_x, \{\mathcal{X}_x \supseteq ifnonempty(\mathcal{Z}, \mathcal{X})\} \cup \mathcal{C})} \qquad \text{(FIX)}$$

Figure 6: Construction of Set Constraints (complete version)