Set Based Analysis of Arithmetic

NEVIN HEINTZE

December 1993

CMU-CS-93-221

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

(Also appears as Fox Memorandum CMU-CS-FOX-93-06.)

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.

Keywords: program analysis, ML, set based analysis, set constraints, range analysis, array bounds checks.

Abstract

Set based analysis is an approach to compile-time program analysis that is based on a simple approximation: all dependencies between variables are ignored. In effect, program variables are treated as sets of values. Thus far, set based analysis techniques have focussed on data-constructor languages. The main reason for this is algorithmic: the equality for data-constructor values is structural (that is, two values $f(v_1, \ldots, v_n)$ and $f'(v'_1, \ldots, v'_n)$ are equal if and only if f and f' are identical constructors and $v_i = v'_i$, $i = 1..n$). This has important implications for how sets of such values can be represented during the computation of a set based analysis.

In contrast, the equality theory of arithmetic is much richer. Two terms with very different structure can be equal. Correspondingly, the manipulation and representation of sets of arithmetic values is significantly more complex. In this paper we extend the ideas of set based analysis to arithmetic expression in such a way that the analysis yields descriptions of how arithmetic values are computed. Importantly, this extended analysis yields useful information about the arithmetic components of a program while maintaining the efficiency of the basic set constraint approach. We show how this information can be exploited during compilation with two examples involving array bounds elimination. While this work is carried out in the context of the ML, the techniques developed appear to be applicable to other languages.

1 Introduction

Set based analysis [5, 7, 8] is an approach to compile-time program analysis that is based on a simple notion of approximation: all dependencies between variables are ignored. This notion is formalized by treating program variables as denoting sets of values instead of individual values. For example, if at some point in a program, the environments $[x \mapsto 1, y \mapsto 2]$ and $[x \mapsto 3, y \mapsto 4]$ can be encountered, then the set based analysis of the program will introduce set variables $\mathcal{X}$ and $\mathcal{Y}$ to represent the respective values of x and y at the given point, and $\mathcal{X}$ will contain both 1 and 3, and $\mathcal{Y}$ will contain 2 and 4. In effect, the x–y dependency that "x is 1 when y is 2" and "x is 3 when y is 4" are ignored; instead only the sets of values for x and y are retained. Call the approximation that arises from this interpretation the program's *set based approximation*.

Computationally, set based analysis proceeds by extracting *set constraints* from a program such that the least solution of the constraints corresponds exactly to the program's set based approximation. These constraints are then input to a solver that computes an explicit representation of the least solution of the constraints (the constraint solving process is the main algorithmic component of the analysis).

To date, set based analysis techniques have focussed on data-constructor languages. The main reason for this is that two values $f(v_1, \ldots, v_n)$ and $f'(v'_1, \ldots, v'_n)$ are equal if and only if f and f' are identical constructors and $v_i = v'_i$, $i = 1..n$. This has important implications for the constraint solving process. In particular, it means that the least solution of the constraints can be incrementally built up in a form such that at any time, the structure of the partial solution constructed thus far is explicit, and questions about membership and emptiness can be directly answered.

However, when set based analysis is extended to arithmetic, a problem arises: the equality theory of arithmetic is much richer than the structure equality of data-constructors. Two terms with very different structure can be equal (for example, 3 and $(42 - 48) + 9$ represent the same value). The essential problem is: how can we explicitly represent and incrementally build up solutions of constraints involving arithmetic?

One approach to the arithmetic problem is to employ an abstract interpretation style approximation of the arithmetic component of the language. That is, we could use set constraint techniques to reason about the data-constructor component of a program, and abstract interpretation techniques to reason about its arithmetic components. (With similar motivations, a hybrid combination of set constraints and abstract interpretation is developed for logic programs in [9].)

Such an approach has two disadvantages. First, we would like to avoid increasing the complexity of the algorithm beyond the complexity of the core set constraint algorithm (for analysis of ML, the complexity is $O(n^3)$ where n is the size of the input program; in practice, it can be engineered to be nearly linear). This means that we have to place severe limits on the complexity of the abstract domain, with a resulting loss of information. Second, adding an abstract interpretation mechanism would involve adding extra machinery to the algorithm, particularly if the techniques of narrowing and widening [3] are employed.

In this paper we develop an alternative approach that effectively computes a representation of *how* a value is obtained in terms of the basic arithmetic operations. In section 2 we describe the basic extensions of set based analysis needed to compute descriptions of arithmetic values. In section 3 we describe an important extension of the basic approach that significantly increases the accuracy of the analysis. In section 4 we discuss some example programs and provide some preliminary data on how the information obtained from the analysis can be used to improve program performance. Finally, in section 5 we outline some future directions.

We conclude with a brief discussion of related work. This paper is heavily dependent on a companion paper [6], which describes the set based analysis of a call-by-value functional language with data-constructors, references, arrays, exceptions and callcc. Hence there are close connections with this work and work related to set based analysis (for example, [1, 11, 12, 14, 15]). There has been little work on the analysis of arithmetic in languages with higher-order functions, side-effects and continuations. However, there has been much work in the general area of analysis of programs that contain arithmetic. Some of these works focus on obtaining accurate information about complex relationships between program variables. For example [4] and [10] on obtaining information about linear relationships between variables. Another example is array data dependency analysis, which is a local analysis directed towards detecting loop level parallelism in numeric programs. At the other extreme, type analysis has been used to inexpensively obtain very simple information about arithmetic expressions. Another important classification is range analysis, based on abstract interpretation [3]. Range analysis typically ignores dependencies between variable values, but is somewhat more accurate than type analysis approaches. The motivation for this kind of analysis is to determine when arithmetic tests (such as those for array bounds checking) can be safely ignored. Our work is most closely related to range analysis, and our motivations are similar.

2 Arithmetic Expressions

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

$$\begin{aligned} e \;::=\;& x \mid c(e_1, \ldots, e_n) \mid \lambda x.e \mid e_1\; e_2 \mid \mathbf{case}(e_1,\; c(x_1, \ldots, x_n) \Rightarrow e_2,\; y \Rightarrow e_3) \mid \mathit{fix}\; x.e \\ & \mid i \mid e_1\; \mathit{arithop}\; e_2 \mid \mathbf{if}\; x\; \mathit{relop}\; y\; \mathbf{then}\; e_1\; \mathbf{else}\; e_2 \end{aligned}$$

where $x, x_1, \ldots, x_n, y$ range over program variables, c ranges over a given set of (varying arity, "first-order") constants, i ranges over integers[1], *arithop* ranges over arithmetic operations such as $+, -, *, /$, and *relop* ranges over comparison operations such as $=, <>, \leq, <$, etc. It is convenient to adopt the usual convention that each bound variable is distinct. The operator *fix* serves to express recursion[2]. This language is an extension of the language used in [6], and the operational semantics we shall use is an extension of the one given there. Specifically, *environments* are finite mappings from program variables to values. *Values* are either (i) integers, (ii) of the form $c(v_1, \ldots, v_n)$ where the v_i are values, or (iii) closures of the form $\langle E, \lambda x.e \rangle$ where

[1] The restriction to integers is for convenience; the implementation, described later in the paper, deals with floating point numbers as well as a variety of integer data types.

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

E is an environment. We briefly outline evaluation of expressions. Evaluation of variables, abstractions, applications and fix expressions proceeds in the usual way. Evaluation of a case statement $\mathbf{case}(e_1,\ c(x_1,\ldots,x_n) \Rightarrow e_2,\ y \Rightarrow e_3)$ proceeds by first evaluating e_1. If the result has the form $c(v_1,\ldots,v_n)$, then the variables x_i are bound to v_i and e_2 is evaluated. Otherwise, y is bound to the result of evaluating e_1 and then e_3 is evaluated. Evaluation of $c(e_1,\ldots,e_n)$ proceeds by evaluating each e_i, say to v_i, and then constructing the value $c(v_1,\ldots,v_n)$. Arithmetic expressions e_1 *arithop* e_2 are evaluated by first evaluating e_1 and then e_2, and then combining the results using *arithop*. A conditional statement $\mathbf{if}\ x\ \mathit{relop}\ y\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$ is evaluated by first evaluating x and y, applying relop to the results, and then appropriately branching to either e_1 or e_2. If the evaluation of an arithmetic operation causes an exception, then the computation is aborted. We shall write $E\ \vdash\ e \rightarrow v$ if e evaluates to v in the context of environment E. If E is the empty environment[3] then we omit it, and just write $\vdash\ e \rightarrow v$.

In [6], we gave a *set based semantics* for the data-constructor subset of the above language. This was achieved by replacing the use of environments in the (exact) operation semantics, by set environments. A set environment is like an environment except that it maps variables to sets of values instead of individual values. By this means, we formalized the notion of ignoring dependencies between variables, and defined the *set based approximation* $sba(e_0)$ of a program e_0 (which is a conservative approximation of $\{v : \vdash\ e_0 \rightarrow v\}$[4]). We then extracted constraints from a program and presented an algorithm to solve these constraints such that the output of the algorithm is a representation of the (possibly infinite set) $sba(e_0)$. Strictly speaking, the output of the set constraint algorithm was not $sba(e_0)$, but a set of descriptions from which $sba(e_0)$ could be reconstructed. (The algorithm also computes, for each program variable, a set that conservatively describes all values that the variable can be bound to during program execution.)

We now extend this procedure to the arithmetic component of the language. The key idea is to generalize the notion of "description" to include arithmetic operations. For example consider the power program.

```
let fun power(0, n) = 1
      | power(m, n) = n × power(m-1, n)
in
    power(3, 4)
end
```

Program 1

When our analysis is applied to this program, the following regular tree grammar is output:

$$P \Rightarrow (4 \times P)$$
$$P \Rightarrow 1$$

This represents the set of expressions $\{1,\ \ 4 \times 1,\ \ 4 \times (4 \times 1),\ \ 4 \times (4 \times (4 \times 1)), \ldots\}$, and indicates that the program returns a value that is obtained by starting with 1 and repeatedly multiplying by 4 some arbitrary number of times.

[3] That is, its domain is the empty set of variables.

[4] This set is either empty or a singleton set.

We begin by describing the set constraints we shall employ. The calculus described here is an extension of the calculus described in [6], which in turn is based on the earlier works [8, 12, 15]. The form and meaning of the 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, an integer, or of one of the forms $se_1\ arithop\ se_2$, $c(se_1, \ldots, se_2)$, $apply(se_1, se_2)$, $case(se_1, c(\mathcal{X}_1, \ldots, \mathcal{X}_n) \Rightarrow se_2, \mathcal{Y} \Rightarrow se_3)$ or $ifnonempty(se_1, se_2)$. The first form is used to model arithmetic expressions, and is the main change from [6]. The second form models expressions $c(e_1, \ldots, e_n)$, the third models application, the fourth 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 outline the meaning of set constraints. Define that a *set constraint value* (sc-value) is either an abstraction $\lambda x.e$ that appears in e_0, an integer i, or of the form $c(u_1, \ldots, u_n)$ or $u_1\, arithop\, u_2$ where each u_i is an sc-value. In essence, sc-values are *description* of values. These descriptions differ from values in two respects: (i) the descriptions contain arithmetic operations, and (ii) the descriptions omit the environment component of closures[5]. Set expressions are interpreted as sets of these descriptions of values. Specifically, 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. The rules for this interpretation are essentially identical to those in [6]. Two new rules are needed for the new kinds of set expressions:

- $\mathcal{I}(i) = \{i\}$, where i is an integer, and
- $\mathcal{I}(se_1\ arithop\ se_2)) = \{u_1\ arithop\ u_2 : u_1 \in \mathcal{I}(se_1) \wedge u_2 \in \mathcal{I}(se_2)\}$.

The complete definition is given in Appendix I. 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 $\mathcal{X}$. For example the least model of the constraint $\mathcal{X} \supseteq (4 \times \mathcal{X}) \cup 1$ maps the set variable $\mathcal{X}$ into the set of values $\{1,\ 4 \times 1,\ 4 \times (4 \times 1),\ 4 \times (4 \times (4 \times 1)), \ldots\}$ (which is same as the language generated by the grammar given earlier in this section).

We now outline how set constraints are constructed for a program. Since this construction is an extension of that given in [6], we shall only discuss the additional rules required (these correspond to the arithmetic expressions of the language). The following is a slightly simplified version of the new rules (the complete set of rules appears in Appendix II). The judgements[6]

[5]Importantly, these can be reconstructed from the output of the set constraint algorithm.

[6]Strictly speaking, the judgements used in Appendix II have a slightly more complex form: $\mathcal{Z} \vdash e \triangleright (\mathcal{X}, \mathcal{C})$. The additional set variable $\mathcal{Z}$ is used to reason about non-emptiness. In effect, it defers the actions of a function until it is determined that the function may be called.

of this system have the form $e \rhd (\mathcal{X}, \mathcal{C})$ where e is a term, $\mathcal{X}$ is a program variable and $\mathcal{C}$ is a collection of set constraints.

$$i \rhd (i, \{\}) \tag{INTEGER}$$

$$\frac{e_1 \rhd (\mathcal{X}_1, \mathcal{C}_1) \qquad e_2 \rhd (\mathcal{X}_2, \mathcal{C}_2)}{e_1 \; arithop \; e_2 \rhd (\mathcal{Y}, \{\mathcal{Y} \supseteq \mathcal{X}_1 \; arithop \; \mathcal{X}_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2)} \tag{ARITHOP}$$

$$\frac{e_1 \rhd (\mathcal{X}_1, \mathcal{C}_1) \qquad e_2 \rhd (\mathcal{X}_2, \mathcal{C}_2)}{\mathbf{if}\, x_1 \; relop \; x_2 \;\mathbf{then}\; e_1 \;\mathbf{else}\; e_2 \rhd (\mathcal{Y}, \{\mathcal{Y} \supseteq \mathcal{X}_1 \cup \mathcal{X}_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2)} \tag{IF}$$

Note that the rule for conditional statements merely joins together the results from the two branches of the conditional statement. In other words, information about the "test" part of the statement is ignored. Clearly this loss of information is unacceptable; we address this issue in the next section. Let $\mathcal{SC}(e_0)$ denote the pair $(\mathcal{X}, \mathcal{C})$ constructed for a term e_0 according to the constraint construction procedure ($\mathcal{C}$ is the collection of constraints constructed for e_0, and $\mathcal{X}$ is effectively a pointer into the constraints that indicates which set variable captures the set of values corresponding to e_0).

Meanwhile, we describe the changes to the set constraint algorithm of [6] that are needed to deal with the new constraints. In effect the algorithm is extended so that set expressions of the form $se_1 \; arithop \; se_2$ are treated in an identical manner to set expressions of the form $c(se_1, \ldots, se_2)$. That is, arithmetic operations are treated like data-constructors. The only difference is that we have no de-construction operation for arithmetic operations - we can build them up, but we can't break them down. Formally, this is achieved by adapting the definition of *atomic set expressions* to include expressions of the form $ae_1 \; arithop \; ae_2$ where ae_1 and ae_2 are atomic set expressions. With this change of definition, the set constraint simplification algorithm of [6] can be directly applied. When input with a collection of constraints $\mathcal{C}$, this $O(n^3)$ algorithm computes an explicit representation of the least model of the constraint $\mathcal{C}$. This representation takes the form of a regular tree grammar.

We now formalize the connection between the descriptions of values computed by the set constraint algorithm and the values of the operational semantics. Define a meaning function, which maps an arbitrary sc-value into an sc-value that does not contain arithmetic operations, as follows:

$$[\![u]\!] = \begin{cases} \lambda x.e & \text{if } u \text{ is } \lambda x.e \\ i & \text{if } u \text{ is } i \\ c([\![u_1]\!], \ldots, [\![u_n]\!]) & \text{if } u \text{ is } c(u_1, \ldots, u_n) \\ [\![u_1]\!] \; \underline{arithop} \; [\![u_2]\!] & \text{if } u \text{ is } u_1 \; arithop \; u_2, \; [\![u_1[\![\text{ and } [\![u_2[\![\text{ are both integers} \end{cases}$$

In the last line of the definition, $[\![u_1]\!] \; \underline{arithop} \; [\![u_2]\!]$ denotes the result of applying the arithmetic operation to the integers $[\![u_1]\!]$ and $[\![u_2]\!]$. For example, $[\![(3 + 4) - 1]\!]$ is 6. Finally, given a set S

of sc-values, define that $[\![S]\!]$ is $\{[\![u]\!] : u \in S \wedge \ [\![u]\!]$ is defined$\}$.

Next, we extend the operator $\|v\|$ on values v (defined in [6]), whose purpose is to ignore the environment part of closures:

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

Finally, we can state the correspondence between the set constraints constructed from a program and the operational behaviour of the program:

Theorem 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 $\{\|v\| : \vdash \ e_0 \to v\} \subseteq [\![\mathcal{I}_{lm}(\mathcal{X})]\!]$.*

The correctness of the analysis procedure described in this section follows from the above theorem and the correctness of the set constraint simplification algorithm presented in [6]. Note that complexity of the whole analysis process (constructing set constraints from a program and then solving them) is $O(n^3)$ where n is the size of the input program. This is because the translation of a program into its set constraints is $O(n)$ in the size of the program and the set constraint algorithm is an $O(n^3)$ algorithm.

3 Conditional Statements

In the previous section, we outlined a basic analysis for a call-by-value functional language involving arithmetic. A limitation of this analysis was that it ignored information in the tests of conditional statments. To see the specific ways where the treatment of conditional statements loses information, consider the statement $\mathbf{if} x_1$ *relop* x_2 $\mathbf{then}$ e_1 $\mathbf{else}$ e_2. Now, if the test x_1 *relop* x_2 always succeeds or always fails, then the analysis will infer inferior information because it unnecessarily joins together the values obtained from *both* e_1 and e_2. In contrast, the analysis of a case statement $case(e_1, c(x_1, \ldots, x_n) \Rightarrow e_2, y \Rightarrow e_3)$ ignores the values from the e_2 and e_3 until they are *activated* (i.e. until it is determined that e_1 may contain values that necessitate that the particular arm be executed). This is modeled by the case set expression. Unfortunately, it is difficult to treat conditional statements in an analogous manner. This is because the set constraints reason about *descriptions* of arithmetic computations, and from these descriptions there is no simple way to determine whether a branch will be taken. (The key point is that this test must potentially be done at each step of the algorithm; it is therefore imperative for efficiency that it be a local test. Note that for case statements, the corresponding test – also potentially done at each step – is a trivial constant time test.) This is a direct reflection of the difference between the simple structural equality of data-constructors, and the more complex equality of arithmetic.

However, there is a much more significant loss of information than the issue of "dead" branches in conditional statements. Consider the statement $\mathbf{if}\ x > y\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$, and for simplicity, suppose that the only free variables in e_1 and e_2 are x and y. Now, inside expression e_1 it will always be the case that $x > y$ and inside e_2 it will be the case that $x \leq y$. In other words, the conditional statement serves to restrict the possible bindings to variables in the scope of its body. From an analysis perspective, it is therefore useful to think of conditional statements as a form of variable binding mechanism. More concretely, suppose that the conditional statement is to be executed in environment ρ and let e'_1 and e'_2 be the result of renaming x and y into new variables x' and y' in e_1 and e_2. Then, at an informal level, the evaluation of e_1 and e_2 can be viewed as proceeding in the following environments:

environment for e_1	environment for e_2
$x' \mapsto \rho([> y](x))$	$x' \mapsto \rho([\leq y](x))$
$y' \mapsto \rho([< x](y))$	$y' \mapsto \rho([\geq x](y))$

where $[> y](x)$ denotes the value of x if $x > y$ and is undefined otherwise, $[< x](y)$ denotes the value of y if $y < x$ and is undefined otherwise, and similarly for $[\leq y](x)$ and $[\geq x](y)$. More generally, define an expression $[relop\ y](x)$ which, in the context of some environment ρ, denotes $\rho(x)$ if $\rho(x)\ relop\ \rho(y)$ and is undefined otherwise.

Such a view of conditional statements could be formalized to give an alternative operational semantics that emphasizes the latent binding effects of conditional statements. Rather than spell out the details of this, we shall instead focus on how these intuitions can be employed in analysis. In effect, we shall treat expressions such as $[> y](x)$ in much the same way as we treat expressions such as $1 + 3$. As before, we shall construct set constraints from a program that shall reason about descriptions of values (rather than the actual values), but now these descriptions shall involve expressions of the form $[relop\ u_1](u_2)$ in addition to $u_1\ arithop\ u_2$.

To this end, we first introduce a new kind of set expression which has the form $[relop\ se_1](se_2)$. We shall also extend the definition of sc-value to include expressions of the form $[relop\ u_1](u_2)$. The new form of set expression shall be interpreted as follows:

$$\mathcal{I}([relop\ se_1](se_2)) = \{[relop\ u_1](u_2) : u_1 \in \mathcal{I}(se_1) \wedge u_2 \in \mathcal{I}(se_2)\}.$$

Accordingly, the definition of the function $[\![\ ]\!]$, which maps an arbitrary sc-values into its meaning, is extended as follows: $[\![[relop\ u_1](u_2)]\!] = u_2$ if $u_2\ relop\ u_1$ and is undefined otherwise.

Finally, the construction of constraints is modified so that when an expression of the form $\mathbf{if}\, x\ relop\ y\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$ is encountered, occurrences of x and y in e_1 and e_2 are treated specially:

- when x is encountered in e_1, the set expression $[relop \mathcal{X}_y](\mathcal{X}_x)$ is generated instead of $\mathcal{X}_x$;
- when y is encountered in e_1, the set expression $[op(relop)\mathcal{X}_x](\mathcal{X}_y)$ is generated instead of $\mathcal{X}_y$;
- when x is encountered in e_2, the set expression $[neg(relop)\mathcal{X}_y](\mathcal{X}_x)$ is generated instead

of $\mathcal{X}_x$, and

- when x is encountered in e_2, the set expression $[op(neg(relop))\mathcal{X}_x](\mathcal{X}_y)$ is generated instead of $\mathcal{X}_y$.

where op maps a relational operator into its opposite (for example $op(<)$ is $>$), and neg maps a relational operator into its negation (for example $neg(<)$ is $\geq$). Strictly speaking, we shall cascade this process, so that at the occurrence of x in the expression $x + 1$ of the conditional statement

if $x > y$ **then if** $x < z$ **then** $x + 1$ **else** 1 **else** 2

the set expression $[< \mathcal{X}_z]([> \mathcal{X}_y](\mathcal{X}_x))$ is generated.

We conclude this section with an example illustrating the kinds of output generated by this modification to the analysis. Consider the following ML program for adding up the elements in an array. The functions update, sub and length are the update, subscript and array length operations; the expression array(10, 3) creates an array of size 10 with each element initialized to 3 (the valid subscripts of the array are 0..9).

```
let fun cum (arr : int array) =
    let  fun f  i =  if  (i = 0)  then  arr sub 0
                     else  (arr sub i) + f(i – 1)
    in
         f ((length arr) – 1)
    end
in
    cum (array(10, 3))
end
```

Program 2

Of particular interest is the set of values for i. If we can determine that i is always in the range 0..9 then the array bounds check can be eliminated during the subscript operation. When applied to program 2, the analysis described in this section outputs the following tree grammar corresponding to the program variable i, where $\mathcal{I}$ is the set variable corresponding to i:

$$\mathcal{I} \Rightarrow 10 - 1$$
$$\mathcal{I} \Rightarrow ([\neq 0](\mathcal{I})) - 1$$

That is, the set of values obtained for i is the set consisting of the integer 9 as well as any integer obtained by subtracting 1 from any non-zero integer described by $\mathcal{I}$. It is easy to verify[7] from this description that the possible values for i must lie in the range 0..9, and so the array bounds check is not required.

[7] In general, some postprocessing is needed to obtain the required properties from the descriptions output by the analyzer. We address this issue further in the Section 5.

For the purposes of comparison, we now outline how the abstract interpretation approach based on the domain of intervals with widening and narrowing would perform on this example. The abstract values of the interval domain take the form $[x, y]$, $[x, \infty]$, $[-\infty, y]$ or $[-\infty, \infty]$, where x and y are integers. An interval $[x, y]$ indicates the set of all integers i such that $x \leq i \leq y$. (See [3] for further details.) We now trace out the behaviour of the analysis and focus on the values obtained for the program variable i. The first value for i is 9, and this is represented by the interval $[9, 9]$. During the next iteration, a new value 8 is obtained by i, and this is represented by the interval $[8, 8]$. When these two intervals are combined using the widening operator typically employed for this domain[8] the result obtained is $[-\infty, 9]$. This completes the widening phase of the analysis (at least as far as i is concerned). Next, the narrowing phase of the analysis begins. However, for this program, it is not possible to obtain a tighter bound on i using narrowing, and so the final result is $[-\infty, 9]$, which does not imply that the array bounds check can be removed. Observe that if the test in the conditional statement were changed to $x \leq 0$ instead of $x = 0$, then the abstract interpretation approach would obtain the description $[0, 9]$ for i.

4 Examples

We now consider two more substantial examples to illustrate the kinds of information that our analysis can obtain. The first example is a small package (about 100 lines) that implements two-dimensional arrays using the one-dimensional arrays of SML of New Jersey. The package provides subscript and update operations, as wall as a matrix multiplication operation. This package was analyzed in the context of a benchmark that multiplies two 100×100 matrices. The time for analysis was approximately 0.06s[9]. Of particular interest are the manual bounds check operations for each dimension of the two-dimensional arrays. A typical example of the results obtained by the analysis for an index involved in bounds checking, is given by the set variable $\mathcal{X}$ in the following regular tree grammar:

$$\begin{array}{lcl} \mathcal{X} & \Rightarrow & [< 100](\mathcal{Y}) \\ \mathcal{Y} & \Rightarrow & ([< 100](\mathcal{Y})) + 1 \\ \mathcal{Y} & \Rightarrow & 0 \end{array}$$

Focus first on the set variable $\mathcal{Y}$. The values described by $\mathcal{Y}$ consist of the integer 1 and any value obtained by adding 1 to some value in $\mathcal{Y}$ that is strictly less than 100. If is easy to see that the values described by $\mathcal{Y}$ are exactly the interval $0..100$. If follows that the set described by $\mathcal{X}$ is the interval $0..99$, and therefore the bounds checks can be eliminated. In fact the analysis determines that all of the two-dimensional bounds checks can be eliminated for this program. This resulted in an approximately 40% speedup in the benchmark.

The second example is the core part of the set based analysis implementation (approximately 2700 lines). It relies less heavily on array operations, although it does contain substantial symbolic and imperative aspects. It also make significant use of higher-order functions: in

[8] In general, widening is needed to obtain a terminating abstract interpretation analysis using the interval domain.

[9] All execution times reported in this section are in seconds on an PMAX 5000/200 with 64M and running Mach. The analysis is implemented in Standard ML of New Jersey [2], Version 0.93.

particular functions are stored in lists, data-structures and arrays and then recovered and called when necessary. An important part of the design of the system is the use of integers to provide a compact and efficient representation of certain objects call *terms*. The class of terms is partitioned into *identifiers* and *compound-terms*. Negative integers are used for identifiers and positive integers are used for compound-terms. Both identifiers and compound-terms are allocated using global references that contain the respective next integers to be allocated. Finally, both kinds of terms are used to index into arrays. The following is suggestive of the allocation mechanism for compound-terms.

```
val last_compound_term = ....
val next_compound_term = ref 1
fun new_compound_term () =
    let val i = !next_compound_term in
        next_compound_term := i + 1;
        i
    end
```

where .... indicates the computation of the maximum size of integers used for compound-terms (it is computed from some parameter describing the problem size). Any arrays indexed by compound-terms are created with size last_compound_term + 1. There is a similar mechanism for the allocation of identifiers, except that identifiers are allocated in the opposite direction, starting from -1. These term objects appear in almost all parts of the program, are stored in arrays and data-structures and also appear in the closures of higher-order functions.

The analysis of this second example was performed in about 7.6s. The results of the analysis for variables that range over term objects are typically of the following form (the set variable $\mathcal{X}$ describes the possible values of the program variable of interest).

$$
\begin{array}{lcl}
\mathcal{X} & \Rightarrow & [\neq -((20000/3)+1)](\mathcal{Y}) \\
\mathcal{X} & \Rightarrow & [\neq ((20000/3)+1)](\mathcal{Z}) \\
\mathcal{X} & \Rightarrow & 0 \\
\mathcal{Y} & \Rightarrow & -1 \\
\mathcal{Y} & \Rightarrow & [\neq -((20000/3)+1)](\mathcal{Y}) - 1 \\
\mathcal{Y} & \Rightarrow & 0 \\
\mathcal{Z} & \Rightarrow & 1 \\
\mathcal{Z} & \Rightarrow & [\neq ((20000/3)+1)](\mathcal{Z}) + 1 \\
\mathcal{Z} & \Rightarrow & 0
\end{array}
$$

Note that the expression $(20000/3) + 1$ corresponds to the initial computation to determine the bounds on the allocation of identifiers and compound-terms. The above description clearly reflects the allocation of compound-terms (starting from 1 and incrementing) and the allocation of identifiers (starting from -1 and decrementing). The possible value of 0 is used to represent an uninitialized term and is used in a part of the program at which a variable must be defined before it can be given its proper value.

Most importantly, the descriptions for the variables that are used for subscripting into arrays have one of the following two forms: $[> 0](\mathcal{X})$ or $-[< 0](\mathcal{X})$. In either case, the description establishes that the array operation can be done without bounds checking. Preliminary results indicate that when these array operations are done without array bounds checking, overall performance of the analyzer improves by about 6% – 13%, depending on the program analyzed.

5 Conclusion

We have presented an extension to set based analysis for the treatment of arithmetic expressions in an ML-like language. The analysis yields revealing descriptions of how arithmetic values are obtained during the execution of a program. We have investigated the use of this information to remove array bounds checks for two different styles of programs. Preliminary data suggests that the approach presented here will scale up to large programs.

We observe that the information computed by the analysis described by this paper is not explicit in the sense that properties cannot be directly read from it. Rather, some post-processing is required. In practice, the information that is relevant to a particular program variable is usually quite small and fairly easy to reason about, even for moderate sized programs. In general, however, it may involve complex recurrence relations[10]. One area of future work is the development of tools to reason about the output of the analyzer.

Although the use of a somewhat implicit representation of values is a disadvantage in the sense that postprocessing is required, it is also an advantage because it means that we can compute a much richer variety of properties that is possible in other approaches (for example, in abstract interpretation, we fix in advance an explicit representation used in the analysis, but in the process we restrict the properties that can be inferred). In effect, our approach defers the choice of "interesting properties" until the post-processing stage. This leads to an important level of modularity, since when we wish to analyze for a new property, we typically only need to construct a new postprocessing stage. For example, suppose that for alignment purposes, we wished to determine whether a byte array was always created with a size that was a multiple of 4. Such information could be determined from the output of our analysis by adapting the postprocessing stage for the property of interest. In comparison, consider an abstract interpretation analysis based on intervals (such as used in the example at the end of Section 3). In order to modify the abstract interpretation analysis for this new property, we would need to completely redesign the analysis to appropriately enrich the abstract domain.

Future Work

Much work remains in the use of the analysis during compilation. The results relating to array reported in Section 4 are very preliminary. Further applications involving data representation,

[10] In some sense, the approach outlined in this paper can be viewed as a technique for decomposing the large problem of reasoning about the arithmetic operations of an entire program (which contains other aspects such as higher order functions, continuations and data-structures) into a number of smaller "local" arithmetic problems. Since these problems are typically much smaller than the original problem, and are often important for optimization purposes, we may wish to use more expensive analysis procedures on these subproblems than we could justify for analysis of the entire program.

binding time analysis and flat types [13] are being investigated. The approach of computing descriptions of how a value is obtained also seems relevant to other kinds of complex values such as strings, sets and Lisp futures.

Acknowledgments

Thanks to Peter Lee, Greg Morrisett, and David Tarditi for many useful discussions. Thanks also to David for the use of the two-dimensional array benchmark and to Chris Colby for comments on a draft of this paper.

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. Appel, "Compiling with Continuations", Cambridge University Press, 1992.

[3] 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.

[4] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program" *Proc.* 5^{th} *ACM Symp. on Principles of Programming Languages*, Tuscon, Arizona, pp. 84–97, January 1978.

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

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

[7] 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.)

[8] 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.)

[9] 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.

[10] P. Granger, "Static Analysis on Linear Congruence Equalities Among Variables of a Program", TAPSOFT'91, LNCS 493, pp. 162–192, 1991.

[11] 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.

[12] 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.

[13] J. G. Morrisett, "Data Representation and Polymorphic Languages", thesis proposal, School of Computer Science, Carnegie Mellon University, December 1993.

[14] 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.

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

Appendix I : Interpretation of Set Expressions

1. $\mathcal{I}(i) = \{i\}$;
2. $\mathcal{I}(se_1 \; arithop \; se_2)) = \{v_1 \; arithop \; v_2 : v_1 \in \mathcal{I}(se_1) \wedge v_1 \in \mathcal{I}(se_1)\}$;
3. $\mathcal{I}(c(se_1, \ldots, se_n)) = \{c(v_1, \ldots, v_n) : v_i \in \mathcal{I}(se_i), i = 1..n\}$;
4. $\mathcal{I}(\lambda x.e) = \{\lambda x.e\}$;
5. $\mathcal{I}(\mathit{ifnonempty}(se_1, se_2)) =$ if $\mathcal{I}(se_1) = \{\}$ then $\{\}$ else $\mathcal{I}(se_2)$;
6. $\mathcal{I}(\mathit{apply}(se_1, se_2)) = \Big\{ v : \;\; \lambda x.e \in \mathcal{I}(se_1) \; \wedge \; \mathcal{I}(se_2) \neq \{\} \; \wedge \; v \in \mathcal{I}(ran(\lambda x.e)) \;\Big\}$
   provided $\lambda x.e \in \mathcal{I}(se_1)$ implies $\mathcal{I}(se_2) \subseteq \mathcal{I}(\mathcal{X}_x)$
7. $\mathcal{I}(\mathit{case}(se_1, c(\mathcal{X}_1, \ldots, \mathcal{X}_n) \Rightarrow se_2, \mathcal{Y} \Rightarrow se_3)) = S_1 \; \cup \; S_2$,
   where (i) $\quad S_1 = \{v : v \in \mathcal{I}(se_2) \; \wedge \; \exists c(v_1, \ldots, v_n) \in \mathcal{I}(se_1)\}$
   (ii) $\quad S_2 = \{v : v \in \mathcal{I}(se_3) \; \wedge \; \exists v' \in \mathcal{I}(se_1) \; s.t. \; v' \neq c(v_1, \ldots, v_n)\}$
   (iii) $\quad c(v_1, \ldots, v_n) \in \mathcal{I}(se_1)$ implies $v_i \in \mathcal{I}(\mathcal{X}_i), \; i = 1..n$
   (iv) $\quad v \in \mathcal{I}(se_1) \; \wedge \; v \neq c(v_1, \ldots, v_n)$ implies $c'(v_1, \ldots, v_n) \in \mathcal{I}(\mathcal{Y})$

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.

Appendix II : Construction of Set Constraints

$$\mathcal{Z} \vdash x \triangleright (\mathcal{X}_x, \{\}) \tag{VAR}$$

$$\mathcal{Z} \vdash i \triangleright (i, \{\}) \tag{INT}$$

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

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

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

$$\frac{\mathcal{Z} \vdash e_1 \triangleright (\mathcal{X}_1, \mathcal{C}_1) \qquad \mathcal{Z} \vdash e_2 \triangleright (\mathcal{X}_2, \mathcal{C}_2)}{\mathcal{Z} \vdash e_1\ arithop\ e_2 \triangleright (\mathcal{Y}, \{\mathcal{Y} \supseteq \mathcal{X}_1\ arithop\ \mathcal{X}_2\} \cup C_1 \cup \mathcal{C}_2)} \tag{ARITHOP}$$

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

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

$$\frac{\mathcal{Z} \vdash e_1 \triangleright (\mathcal{X}_1, \mathcal{C}_1) \qquad \mathcal{Z} \vdash e_2 \triangleright (\mathcal{X}_2, \mathcal{C}_2)}{\mathcal{Z} \vdash if\, x_1\ relop\ x_2\ then\ e_1\ else\ e_2 \triangleright (\mathcal{Y}, \{\mathcal{Y} \supseteq \mathcal{X}_1 \cup \mathcal{X}_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2)} \tag{IF}$$

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

Figure 1: Construction of Set Constraints

Figure 1 presents the complete details of the constructions of set constraints for a term. The judgement $\mathcal{Z} \vdash e \triangleright (se, \mathcal{C})$ recursively passes down a set variable which is empty if the expression under consideration is never called, and is non-empty otherwise. A key property of the judgement $\mathcal{Z} \vdash e \triangleright (se, \mathcal{C})$ is that if $\mathcal{Z}$ is empty then $\mathcal{C}$ is vacuously true. 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 $(\mathcal{X}, \{\mathcal{Z} \supseteq u\} \cup \mathcal{C})$ where u is some arbitrary sc-value. Note that all sc-values are set expressions and that the choice of u 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.