Interprocedural Symbolic Evaluation of Ada Programs with Aliases

J. Blieberger[1], B. Burgstaller[1], and B. Scholz[2]

[1] Institute for Computer-Aided Automation (183/1),
Technical University Vienna,
Treitlstr. 1/4, A-1040 Vienna, Austria
{blieb,bburg}@auto.tuwien.ac.at

[2] Institute for Software Technology and Parallel Systems,
University of Vienna,
Leichtensteinerst. 22, A-1090 Vienna, Austria
scholz@par.univie.ac.at

Abstract. Symbolic Evaluation is a technique aimed at determining dynamic properties of programs. We extend our intraprocedural data-flow framework introduced in [3] to support interprocedural symbolic evaluation. Our data-flow framework utilizes a novel approach based on an array algebra to handle aliases induced by procedure calls. It serves as as a basis for static program analysis (e.g. reaching definitions-, alias analysis, worst-case performance estimations, cache analysis). Examples for reaching definitions- as well as alias analysis are presented.

1 Symbolic Evaluation

In this section we introduce the basics of interprocedural symbolic evaluation as it is used throughout the paper. We abstract from intraprocedural evaluation details such as conditional or repetitive statements in order to be concise. Treatment of intraprocedural symbolic analysis of Ada programs can be found in [3].

Symbolic evaluation is a form of static program analysis in which symbolic expressions are used to denote the values of program variables and computations (cf. e.g. [5]). In addition a path condition describes the impact of the program's control flow onto the values of variables and the condition under which control flow reaches a given program point. The underlying program representation for symbolic evaluation is usually the *control flow graph (CFG)*, a directed labelled graph. Its nodes are the program's basic blocks (a basic block is a single entry, single exit, sequence of statements), whereas its edges represent transfers of control between basic blocks. Each edge of the CFG is assigned a condition which must evaluate to true for the program's control flow to follow this edge. *Entry* and *Exit* are distinguished nodes used to denote start and terminal node.

Program State and Context

The *state* $\mathcal{S}$ of a program is described by a set of pairs $\{(v_1, e_1), \ldots, (v_m, e_m)\}$ where v_i is a program variable, and e_i is a symbolic expression describing the value of v_i for $1 \leq i \leq m$. For each variable v_i there exists exactly one pair (v_i, e_i) in $\mathcal{S}$.

A program consists of a sequence of statements that may change $\mathcal{S}$.

A *state condition* specifies a condition that is valid at a certain program point. It directly relates to the conditions of the CFG edges. If conditional statements are present, there may be several different valid program states at the same program point. A different state condition is associated with each of them.

States $\mathcal{S}$ and state conditions $\mathcal{C}$ specify a *program context* which is defined by

$$\bigcup_{i=1}^{k} [\mathcal{S}_i, \mathcal{C}_i] \in \mathbb{C}$$

where k denotes the number of different program states valid at a certain program point, and $\mathbb{C}$ represents the set of all possible contexts. A program context completely describes the variable bindings at a specific program point together with the associated state conditions.

Arrays

Besides being an efficient and well-established compound data structure arrays lend themselves nicely to the modelling of memory space (e.g. caches, virtual memory, cf. [4]). In the latter way they can be used to track the adverse effects of *aliasing* on interprocedural analysis.

An array is represented as an element of an array algebra $\mathbb{A}$. For sake of simplicity we describe only one-dimensional arrays with it[1].

The array algebra $\mathbb{A}$ is defined inductively as follows:

1. If n is a symbolic expression, then $\perp_n$ in $\mathbb{A}$.
2. If $a \in \mathbb{A}$ and α, β are symbolic expressions then $a \oplus (\alpha, \beta)$ is in $\mathbb{A}$.
3. Nothing else is in $\mathbb{A}$.

In the state of a context an array variable is associated to an element of the array algebra $\mathbb{A}$. Undefined array states are denoted by $\perp_n$, were n is the *size* of the array and reflects the number of array elements. A write access to an array is defined by $\oplus$-function. The semantics of $\oplus$-function is given as follows

$$a \oplus (\alpha, \beta) = (v_1, \ldots, v_{\beta-1}, \alpha, v_{\beta+1}, \ldots, v_n),$$

where β denotes the index where the value α is written.

An element a in $\mathbb{A}$ with at least one $\oplus$-function is a $\oplus$-chain. Every $\oplus$-chain can be written as $\perp_n \bigoplus_{k=1}^{m} (\alpha_k, \beta_k)$. The length of a chain (also written as $|a|$) is the number of $\oplus$-functions in the chain.

[1] Extending the algebra to the multi-dimensional case is left to the reader.

Simplification Intuitively, a simplification is needed to keep the symbolic description of an array as short as possible. Although the equivalence of two symbolic expressions is undecidable in general [7], a wide class of equivalence relations can be solved in practice.

A partial simplification operator θ is introduced to simplify $\oplus$-chains. It is defined as follows

$$\theta\left(a\bigoplus_{l=1}^{m}(\alpha_l,\beta_l)\right)=\begin{cases}\perp_n\bigoplus_{l=1,l\neq i}^{m}(\alpha_l,\beta_l), & \text{if } \exists 1\leq i<j\leq m:\beta_i=\beta_j\\ \perp_n\bigoplus_{l=1}^{m}(\alpha_l,\beta_l), & \text{otherwise}\end{cases}$$

The partial simplification operator θ seeks for two equal β-expressions in a $\oplus$-chain. If a pair exists, the result of θ will be the initial $\oplus$-chain without the $\oplus$-function, which refers to the β-expression with the smaller index. If no pair exists, the operator returns the initial $\oplus$-chain; the argument could not be simplified. Semantically, the outer β-expression relates to the later assignment; if a previous assignment exists and the index of the previous statement is equal to the current one, the inner $\oplus$-function can be reduced. The value of the array element is overwritten by the outer assignment statement. The partial simplification operator θ can only reduce one redundant $\oplus$-function. Moreover, each $\oplus$-function in the chain is a potentially redundant one. Therefore, the chain can be potentially simplified in less than $|a|$ applications of θ. A complete simplification is an iterated application of the partial simplification operator and it is written as $\theta^*(a)$. If $\theta^*(a)$ is applied to a further application of a partial simplification operator θ will not simplify a anymore: $\theta(\theta^*(a))=\theta^*(a)$.

Please note that for ease of readability we have presented array simplification as this iterated application of the θ operator. However, in practice more efficient algorithms to do this exist.

Array access The ρ operator accesses an element of an array a, which is described as an element of the array algebra $\mathbb{A}$.

If $a=\perp_n\bigoplus_{l=1}^{m}(\alpha_l,\beta_l)$ is element of $\mathbb{A}$, then

$$\rho\left(\perp_n\bigoplus_{l=1}^{m}(\alpha_l,\beta_l),i\right)=\begin{cases}\alpha_l, \text{if } \exists l=\max\{\,l\,|\,1\leq l\leq m\wedge\beta_l=i\}\\ \perp, \text{ otherwise}\end{cases}$$

If index i cannot be found in the $\oplus$-chain, ρ yields the undefined value $\perp$ – this means that either the element was never written or that we have failed to prove the equality of index i with some β inside a $\oplus$-function. If a is not completely simplified, more than one β can be found in the $\oplus$-chain. The β with the highest index is taken.

A Data-Flow Framework for Symbolic Evaluation

We define the following set of equations for the symbolic evaluation framework:

$$\mathrm{SymEval}(B_{\mathrm{entry}})=[\mathcal{S}_0,\mathcal{C}_0]$$

where $\mathcal{S}_0$ denotes the initial state containing all variables which are assigned their initial values, and $\mathcal{C}_0$ is true,

$$\mathrm{SymEval}(B) = \bigcup_{B' \in \mathrm{Preds}(B)} \mathrm{PrpgtCond}(B', B, \mathrm{SymEval}(B')) \mid \mathrm{LocalEval}(B) \tag{1}$$

where $\mathrm{LocalEval}(B) = \{(v_{i_1}, e_{i_1}), \ldots, (v_{i_m}, e_{i_m})\}$ denotes the symbolic evaluation local to basic block B. The variables that get a new value assigned in the basic block are denoted by $v_{i_1}, \ldots, v_{i_m}$. The new symbolic values are given by $e_{i_1}, \ldots, e_{i_m}$. The *propagated conditions* are defined by

$$\mathrm{PrpgtCond}(B', B, \mathrm{PC}) = \mathrm{Cond}(B', B) \odot \mathrm{PC}$$

Denoting by PC a program context, the operation $\odot$ is defined as follows:

$$\begin{aligned}\mathrm{Cond}(B', B) \odot \mathrm{PC} &= \mathrm{Cond}(B', B) \odot [\mathcal{S}_1, p_1] \cup \ldots \cup [\mathcal{S}_k, p_k] \\ &= [\mathcal{S}_1, \mathrm{Cond}(B', B) \wedge p_1] \cup \ldots \cup [\mathcal{S}_k, \mathrm{Cond}(B', B) \wedge p_k]\end{aligned}$$

Definition 1. *The semantics of the $\mid$ operator is as follows:*

1. *We replace*
$$\{\ldots, (v, e_1), \ldots\} | \{\ldots, (v, e_2), \ldots\}$$
by
$$\{\ldots, (v, e_2), \ldots\}.$$
The pair (v, e_1) is not contained in the new set.
2. *Furthermore*
$$\{\ldots, (v_1, e_1), \ldots\} | \{\ldots, (v_2, e_2(v_1)), \ldots\},$$
where $e(v)$ denotes an expression involving variable v, is replaced with
$$\{\ldots, (v_1, e_1), \ldots, (v_2, e_2(v_1)), \ldots\}.$$
For the situations discussed above it is important to apply the rules in the correct order, which is to elaborate the elements of the right set from left to right.
3. *If a situation like*
$$[\{\ldots, (v, e), \ldots\}, C(\ldots, v, \ldots)]$$
is encountered during symbolic evaluation, we replace it with
$$[\{\ldots, (v, e), \ldots\}, C(\ldots, e, \ldots)].$$
4. *For arrays $A \in \mathbb{A}$*
$$\{\ldots, (A, \perp), \ldots\} | \{\ldots, (A, A \oplus (\alpha, \beta)), \ldots\}$$
is replaced by
$$\{\ldots, (A, \perp \oplus (\alpha, \beta)), \ldots\}.$$

And for the general case,

$$\{\ldots,(A, A\bigoplus_{l_1=1}^{m_1}(\alpha_{l_1},\beta_{l_1}),\ldots\}|\{\ldots,(A, A\bigoplus_{l_2=1}^{m_2}(\alpha_{l_2},\beta_{l_2}),\ldots\}$$

is replaced by

$$\{\ldots,(\theta^*\left(A, A\bigoplus_{l_1=1}^{m_1}(\alpha_{l_1},\beta_{l_1})\bigoplus_{l_2=1}^{m_2}(\alpha_{l_2},\beta_{l_2})\right)\ldots\}.$$

This data-flow framework has been introduced in [2], cf. also [3]. The array algebra has been introduced in [4].

2 Aliasing

Call-by-Reference parameter passing between procedures introduces *aliases*, an effect where two or more *l-values* (cf. [1]) refer to the same storage location at the same program point. [10] shows that solving the may-alias problem for $k > 1$ level pointers is undecideable. However, from this proof it follows that determining $k = 1$ (aka single) level pointers is almost trivial - [9] solves this problem with polynomial effort whereas the algorithm we use (cf. [11]) is almost linear w.r.t. time.

Aliases and Ada95

In [8] (6.2) it is stated that parameters in Ada95 are either passed by-copy or by-reference. If a parameter is passed by-copy, any information transfer between formal and actual parameter occurs only before and after execution of the subprogram. By-copy parameter types are elementary types, or descendants of a private type whose full type is a by-copy type.

All other types (e.g. tagged types) are either passed by reference (in which case reads and updates of the formal parameter directly reference the actual parameter object) or the parameter passing mechanism is undefined.

Access types also contribute to the generation of aliases. Line 10 of our *Main* example procedure (cf. Fig. 3) generates a second access path to variable V from within procedure *Do_It*. For this particular invocation of *Do_It* the updates of both V and *P.all* refer to the same storage location.

Treatment

We model the semantics of access values by treating the memory space in which the program is symbolically evaluated as an array A, element of the array algebra $\mathbb{A}$. The address of a given variable V is denoted by $\$V$. The example given in Section 4 utilizes this notation. For the ease of reading we not only give the address and the corresponding symbolic value of an access of A, but also the variable affected by that access. Only for that reason we introduce the ^ operator used for dereferencing access values. Note that ^$V = V$. This translates to a slightly modified use of the $\oplus$ function as

$$a \oplus (\alpha, \beta, \gamma) = (v_1, \ldots, v_{\beta-1}, \gamma, v_{\beta+1}, \ldots, v_n),$$

where α denotes the entity being updated, β denotes the corresponding address, and γ represents the update value.

3 Interprocedural Analysis

Each procedure call may change a symbolic context in two ways:

1. By passing back values from the callee to the caller (e.g. by means of *out* parameters).
2. Through side effects within the callee (e.g. by assigning values to variables not local to the callee).

Topic 1 is achieved by devoting a CFG node to each procedure call. Within this node, parameter passing between caller and callee is handled.

Moreover, the effect of the callee on the current symbolic context (Topic 2) is incorporated at this node. This requires intraprocedural symbolic evaluation of the callee (for details cf. [2]). It is denoted as $\mathrm{LocalEval}(B) = Proc(ap_1, .., ap_n)$ (cf. Equation 1, Section 1) in our data-flow framework (ap_n denoting the *actual* procedure parameters). Intraprocedural symbolic evaluation of the callee utilizes the callee's *formal* parameters, it results in a functional description of the callee's effects on the symbolic context that is then appended to the callers context under consideration of the parameter passing mechanism. Since we restrict ourselves to acyclic procedure call graphs[2], it is possible to evaluate every callee before its callers (post order traversal of the procedure call graph).

The parameter passing mechanism is modelled as it is specified in [8]. A parameter that is passed by-copy is treated as follows:

- Mode *in*: This entity is not allowed to be used as an l-value. Thus it is sufficient to replace such a formal parameter by the corresponding actual parameter (or *default_expression*) at the CFG node representing the call site (cf. the example in Section 4).
- Mode *out*: A formal parameter of mode *out* introduces a new l-value in the program context of the callee. [8] (6.4.1(13)-(15)) defines how such an entity is initialized. For elementary types we have considered so far this is $\perp$ (they are uninitialized).
  To pass back the value of the formal parameter to the actual parameter at the call site we replace the l-value occurrences of the formal parameter in the symbolic description of the callee by the corresponding actual parameter.

[2] Thus excluding recursive calls.

```
package P1 is

   type Int_Pointer is access all Integer;
   V: aliased Integer := 1;

   procedure Do_It(P : Int_Pointer);

end P1;
```

Fig. 1. Package Spec P1

```
package body P1 is

   I: Integer := 0;

   procedure Do_It(P : Int_Pointer) is
   begin
      P.all := P.all + 1;
      V := V + 1;
      I := I + 1;
      P.all := P.all + V + I;
   end Do_It;

end P1;
```

Fig. 2. Package Body P1

- Mode *in out*: Basically the sum of the two modes given above: l-value occurrences as well as occurrences within (right-hand side) expressions of the formal parameter are replaced by the actual parameter.

A parameter that is passed by-reference involves a single-level pointer to the entity of the parameter itself. This can be treated by the approach introduced in Section 2.

4 Example

X_{Exit} of procedure *Do_It* contains the functional description in terms of the *formal* parameter on the symbolic context. For procedure $Main$ we give the SymEval equations for each of its CFG nodes. X_1 corresponds to the entry to the program where all entities are assigned their initial value. X_2 corresponds to lines 6 to 8 of $Main$. X_3 calculates the effect of the first procedure call (line 9), X_4 of the second call. The solution of this set of equations is depicted in X_{Exit} of $Main$. Note that Y stays initialised to *null* throughout this program due to

```
1    with P1, Text_IO;

2    procedure Main is
3       X: P1.Int_Pointer := new Integer'(1);
4       Y: P1.Int_Pointer;
5    begin
6       if False then
7          Y := X;
8       end if;
9       P1.Do_It(X);
10      P1.Do_It(P1.V'access);
11   end Main;
```

Fig. 3. Procedure Body Main

the *False* condition[3] in line 6. This fact is handled correctly by our data-flow framework. For that reason we differ from e.g. [6] where the above case would introduce a so-called *may-alias*.

It is shown in [3] how our framework can be exploited for reaching definition analysis where we can detect if Y was referenced somewhere in the code.

Do_It

$$\begin{aligned} X_{Exit} &= X_1 \\ &= X_{Entry} \mid \{(A, A \oplus (\hat{\ }P, P, \rho(A,P)+1) \oplus (V, \$V, \rho(A,\$V)+1)), \\ &\quad (I, I+1), (A, A \oplus (\hat{\ }P, P, \rho(A,P)+\rho(A,\$V)+I)\} \end{aligned}$$

$Main$

$$X_1 = X_{Entry} \mid \{(A, \perp \oplus (V, \$V, 1) \oplus (\hat{\ }X, X, 1) \oplus (\hat{\ }Y, Y, \perp)), (I, 0)\}$$

$$X_2 = \mathit{False} \odot X_1 \mid \{(A, A \oplus (Y, \$Y, X))\}$$

$$\begin{aligned} X_3 &= (\mathit{True} \odot X_1 \cup \mathit{False} \odot X_2) \mid P1.Do\_It(X) = X_1 \mid P1.Do\_It(X) = \\ &= X_1 \mid \{(A, A \oplus (\hat{\ }X, X, \rho(A,X)+1) \oplus (V, \$V, \rho(A,\$V)+1)), \\ &\quad (I, I+1), (A, A \oplus (\hat{\ }X, X, \rho(A,X)+\rho(A,\$V)+I)\} \end{aligned}$$

$$\begin{aligned} X_4 &= X_3 \mid \; P1.Do\_It(\$V) \\ &= X_3 \mid \{(A, A \oplus (\hat{\ }\$V, \$V, \rho(A,\$V)+1) \oplus (V, \$V, \rho(A,\$V)+1)), \\ &\quad (I, I+1), (A, A \oplus (\hat{\ }\$V, \$V, \rho(A,\$V)+\rho(A,\$V)+I)\} \end{aligned}$$

[3] Contrary to Dead Code Elimination, Symbolic Evaluation can handle arbitrary complex expressions as conditions. Only to keep the example simple and expressive we chose *False*.

$X_{Entry}, X_1, X_2, X_3, X_4 \to X_{Exit}$

$$\begin{aligned}
X_{Exit} = X_{Entry} \mid \{&(A, \perp \oplus (V, \$V, 1) \oplus (\hat{}X, X, 1) \oplus (\hat{}Y, Y, \perp)), (I, 0)\} \\
&\mid \{(A, A \oplus (\hat{}X, X, \rho(A, X) + 1) \oplus (V, \$V, \rho(A, \$V) + 1)), (I, I+1), \\
&\quad (A, A \oplus (\hat{}X, X, \rho(A, X) + \rho(A, \$V) + I)\} \\
&\mid \{(A, A \oplus (\hat{}\$V, \$V, \rho(A, \$V) + 1) \oplus (V, \$V, \rho(A, \$V) + 1)), \\
&\quad (I, I+1), (A, A \oplus (\hat{}\$V, \$V, \rho(A, \$V) + \rho(A, \$V) + I)\} \\
= X_{Entry} \mid \{&(I, 2), (A, \perp \oplus (\hat{}X, X, 5) \oplus (V, \$V, 10) \oplus (\hat{}Y, Y, \perp))\}
\end{aligned}$$

By solving these SymEval equations according to the algorithm presented in [11] we derive for X_{Exit} the program context valid after executing procedure $Main$. Note that this corresponds exactly to the result obtained by executing $Main$ on a *real* CPU.

5 Conclusion

We have presented an extension of the intraprocedural data-flow framework introduced in [2] to support interprocedural symbolic evaluation. Aliases due to by-reference parameter passing are handled by using an array that models the underlying memory. At present we are investigating the use of this concept for general pointers. A prototype implementation of the framework presented is under way.

References

1. A. V. Aho, R. Seti, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.
2. J. Blieberger. Data-flow Frameworks for Worst-Case Execution Time Analysis. (submitted), 1997.
3. J. Blieberger and B. Burgstaller. Symbolic Reaching Definitions Analysis of Ada Programs. Proceedings of the Ada-Europe International Conference on Reliable Software Technologies, 238-250, June 1998.
4. J. Blieberger, T. Fahringer, and B. Scholz. An Accurate Cache Prediction for C-Programs with Symbolic Evaluation. (submitted), 1999.
5. T. E. Cheatham, G. H. Holloway, and J. A. Townley. Symbolic Evaluation and the Analysis of Programs. *IEEE Trans. on Software Engineering*, 5(4):403–417, July 1979.
6. J. D. Choi, M. Burke, and P. Carini. Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects. *ACM PoPL*, 1/93:232–245, 1993.
7. M. Haghighat, C. Polychronopoulos. Symbolic Analysis for Parallelizing Compilers. *ACM Trans. Prog. Lang. Sys.*, 18(4):477-518, July 1996.
8. ISO/IEC 8652. *Ada Reference manual*, 1995.
9. W. Landi, and B. G. Ryder. Pointer-induced Aliasing: A Problem Classification. *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, 235-248, 1992

10. G. Ramalingam. The Undecidability of Aliasing. *ACM Trans. Prog. Lang. Sys.*, 16(5):1467–1471, 1994.
11. V. C. Sreedhar. *Efficient Program Analysis Using DJ Graphs.* PhD thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, 1995.