Elaborating on the alias calculus

Alexander Gerasimov
Saint Petersburg State University,
Computer Science Chair
and
Saint Petersburg National Research University of
Information Technologies, Mechanics and Optics,
Software Engineering Laboratory
Email: asgerasimov@gmail.com

***Abstract*—In this research-in-progress report, we elaborate on the alias calculus introduced in [1], [2]. The alias calculus is to determine whether two reference expressions at the given program point may address the same object at the run-time (in other words, whether one expression may be an alias of the other expression). The main intended application of the alias calculus is to support deductive verification of object-oriented programs. We show how aliases may be used in Hoare-style reasoning, hence derive at what program points we are to compute aliases and propose an algorithm that computes the required aliases. We also state future work directions.**

I. Introduction

To perform deductive verification of an object-oriented program we need to know whether two reference expressions at the given program point may address the same object at the run-time (in other words, whether one expression may be an alias of the other expression). For example, we should take care of such a situation. If x and y are aliased and an operation modifies the value of the attribute $x.f$, then the opertation also modifies $y.f$, though y is not mentioned in the text of the operation. The alias calculus introduced in [1], [2] is to give an answer on the formulated question.

[2] suggests to represent possible aliases at a program point as a symmetric and irreflexive relation over a set of reference expressions (such a relation is called an *alias relation*) and formulates rules of the alias calculus for a model of a programming language.

There are following instructions in the model of a programming language: skip, forget, create, assignment, compound, conditional, cut, loop, and procedure call.

Each rule is of the form

$$(a \gg p) \;=\; A,$$

where A denotes an alias relation that may hold just after the execution of a program instruction p provided an alias relation a holds just before the execution. Let us formulate some of the rules.

The rule for an assignment in the non-object-oriented alias calculus (where every reference expression is a variable) is

$$(a \gg x := y) \;=\; a[x : y],$$

where

- a is an alias relation, x and y are variables,
- $a[x : y] = b \cup \overline{(\{x\} \times (b/y))}$,
- $b = a \setminus\!\!- \{x\} \;=\; a \setminus \{\langle u, v\rangle \in a \mid u = x \vee v = x\}$,
- $b/y \;=\; \{y\} \cup \{u \mid \langle u, y\rangle \in b\}$,
- $\overline{c} = (c \cup \{\langle v, u\rangle \mid \langle u, v\rangle \in c\}) \setminus \{\langle u, u\rangle \mid \langle u, u\rangle \in c\}$ for a relation c.

The rule for a compound instruction is

$$(a \gg (p; q)) \;=\; (a \gg p) \gg q,$$

where p and q are instructions.

A conditional instruction in the model language is of the form "$\mathbf{then}\ p\ \mathbf{else}\ q$" (there is no boolean condition in it). The rule for it is

$$(a \gg \mathbf{then}\ p\ \mathbf{else}\ q) \;=\; (a \gg p) \cup (a \gg q).$$

The alias calculus may say that some expressions are aliased though in fact they cannot because the boolean condition in a real conditional and loop is not taken into account. Such an imprecision may be handled by means of a cut instruction in the model language. The instruction "$\mathbf{cut}\ e_1, e_2$" asserts that the expressions e_1 and e_2 are not aliased at the given program point. In the non-object-oriented alias calculus the rule for "$\mathbf{cut}\ e_1, e_2$" is

$$(a \gg \mathbf{cut}\ e_1, e_2) \;=\; a \setminus \overline{e_1, e_2},$$

where $\overline{s} = \overline{s \times s}$ for a set s of expressions and the braces enclosing a list of the elements of s may be omitted.[1]

A. Related work.

The following approaches to handling references for program verification are known (see [2], [3]): separation logic [4], shape analysis [5], ownership types [6], and dynamic frames [7]. However separation logic and shape analysis try to reveal a more detailed structure of pointers than it is necessary for alias analysis [2]. Separation logic, ownership types, and dynamic frames require a programmer to write additional annotations besides Hoare assertions.

The alias calculus is formulated in terms of a model of a high-level programming language and additional annotations (i. e., cut instructions) are rarely required.

[1] Thus $\overline{e_1, e_2} = \{\langle e_1, e_2\rangle, \langle e_2, e_1\rangle\}$. The notation $\overline{s}$ is also used below.

B. Our work.

We analyse the alias calculus [2] and its prototype implementation (mentioned in [2]) and elaborate on some aspects of the alias calculus in order to provide a solid basis for further implementation and integration of the alias calculus into Eiffel Verification Environment [8]. These aspects are:

- Hoare-style reasoning with aliases;
- computing aliases for calls to (mutually) recursive procedures;
- coping with the infinity of some alias relations in the object-oriented alias calculus.

These aspects and some of our results achieved are described in the subsequent sections.

II. On Hoare-style reasoning with aliases

Consider the following Eiffel code fragment:

---

if b **then** $y := x$
$--$ $true$
$x.set$
$--$ $x.f = c$

Suppose the precondition and postcondition of the qualified procedure call $x.set$ are given in the comments above (the comments are the lines starting with "$--$"). Let us try to infer the weakest precondition of this fragment for the postcondition $y.f = c$.

If we do not know that x and y may be equal just after the call $x.set$, then we cannot weaken the postcondition $x.f = c$ to $y.f = c$.

Suppose we have computed aliases that may hold just before and just after the call $x.set$: x and y may be aliased at these program points. Then we add $x = y$ to the precondition and postcondition of $x.set$ via conjunction. Now we obtain the precondition $b \vee x = y$ of the whole code fragment using Hoare rules [9]:

$--$ $b \vee x = y$
if b **then** $y := x$
$--$ $true \wedge x = y$
$x.set$
$--$ $x.f = c \wedge x = y$
$--$ **implies**
$--$ $y.f = c$

Thus in order to perform Hoare-style reasoning for a program, whose routines are specified with their preconditions and postconditions, we propose (at least) to compute aliases that may hold just before and just after each routine call. (Other details of Hoare-style reasoning with aliases are to be elaborated in future.)

III. Handling procedure calls in the non-object-oriented alias calculus

[2] introduces the following rule for a call **call** $r(l)$ to a procedure r with a list l of actual arguments:

$$(a \gg \textbf{call } r(l)) \;=\; (a[r^\bullet : l] \gg \underline{r}),$$

where

- $r^\bullet$ is the list of formal arguments of the procedure r,
- $\underline{r}$ is the body of the procedure r,
- $a[\tilde{l} : l] \;=\; (\ldots(a[\tilde{x}_1 : x_1])\ldots)[\tilde{x}_m : x_m]$ for lists of variables $\tilde{l} = (\tilde{x}_1, \ldots, \tilde{x}_m)$ and $l = (x_1, \ldots, x_m)$ (see the definition of $a[x : y]$ for variables x, y in Section I).

We elaborate on how to extend this rule to handle possibly mutually recursive procedures. First of all, from Section II we know that we must compute alias relations that may hold just before and just after each procedure call of a program. Next we introduce some definitions and notation and then propose an algorithm that computes required alias relations.

Any alias relation that may hold at the program point just before/after (the occurrence of) the procedure call c is called the *input*/*output* alias relation for (the occurrence of) the procedure call c. (In this definition and in what follows we assume that the aliasing semantics of (possibly mutually recursive) procedures is intuitively clear, however the semantics is to be precisely defined in our future work.)

Let $c_1, \ldots, c_n$ be all the occurrences of procedure calls in the program. Let c_0 be an (implicit) occurrence of the *Main* procedure call, which is performed when the program starts its execution.

For each $j = 0, \ldots, n$ we need to compute

(1) the maximal (w. r. t. inclusion) input alias relation $a^{c_j}_{max}$ for the procedure call c_j (obviously, $a^{c_0}_{max} = \emptyset$) and
(2) the maximal output alias relation $A^{c_j}_{max}$ for the procedure call c_j.

To obtain these alias relations, Algorithm 1 takes into account all possible sequences of procedure calls by iteratively computing an input alias relation a^{c_j} and output alias relation A^{c_j} for the procedure call c_j ($j = 0, \ldots, n$) so that at the termination of the algorithm

$$a^{c_j} = a^{c_j}_{max} \;\text{ and }\; A^{c_j} = A^{c_j}_{max} \;\text{ for each } j = 0, \ldots, n.$$

For an occurrence c of a procedure call in the program we denote by r^c the procedure and by l^c the list of actual arguments of the call.

Algorithm 1 Computes the maximal input and maximal output alias relation for each procedure call.

---

(1) for each $j = 0, \ldots, n$
  (1.1) $a^{c_j} := \emptyset$;
  (1.2) $A^{c_j} := \emptyset$;
(2) while all A^{c_j} ($j = 0, \ldots, n$) are not stabilized
  (2.1) for each $j = 0, \ldots, n$
    (2.1.1) $A^{c_j} := (a^{c_j}[(r^{c_j})^\bullet : l^{c_j}] \gg \underline{r^{c_j}})$,
      where $a^{c_j}[(r^{c_j})^\bullet : l^{c_j}] \gg \underline{r^{c_j}}$ is computed using calculus rules and whenever a subtask of computing $\bar{a}^{c_k} \gg \textbf{call } r^{c_k}(l^{c_k})$ is encountered at a program point just before c_k (for some k):
      (2.1.1.1) $\bar{a}^{c_k} \gg \textbf{call } r^{c_k}(l^{c_k})$ is treated as A^{c_k};
      (2.1.1.2) $a^{c_k} := a^{c_k} \cup \bar{a}^{c_k}$.

Algorithm 1 terminates since the set of expressions (i. e., variables in case of the non-object-oriented alias calculus) that

may be in the computed alias relations is finite. Note also that if all A^{c_j} $(j = 0, \ldots, n)$ are stabilized, then obviously all a^{c_j} $(j = 0, \ldots, n)$ are stabilized too. The algorithm is a variant of the Chaotic Iteration algorithm (see [10]) and it is subject to some optimization, e. g., maintaining a worklist of what really needs to be recomputed.

A. An example.

Let us illustrate how Algorithm 1 works on the model language program given on Fig. 1.

```
procedure Main   -- c_0
    then
        x := y
    else
        x := a;
        call q   -- c_1
    end
end
procedure q
    x := b
    then
        call Main   -- c_2
    else
        a := c
    end
end
```

Fig. 1. An example program [2, Example 13].

1. For $j = 0, 1, 2$: $a^{c_j} := \emptyset$; $A^{c_j} := \emptyset$.
2. $A^{c_0} := (a^{c_0} \gg \underline{r^{c_0}}) = (\emptyset \gg \underline{Main}) = (\overline{x, y},\ (\overline{x, a} \gg \textbf{call}\ q)) = \overline{x, y}$
   as currently $(\overline{x, a} \gg \textbf{call}\ q) = A^{c_1} = \emptyset$;
   $a^{c_1} := \overline{x, a}$.
3. $A^{c_1} := (a^{c_1} \gg \underline{r^{c_1}}) = (\overline{\underline{x, a}} \gg \underline{q}) = ((\overline{x, b} \gg \textbf{call}\ Main),\ \overline{x, b} \gg (a := c)) = \overline{x, y}, \overline{x, b}, \overline{a, c}$
   as currently $(\overline{x, b} \gg \textbf{call}\ Main) = A^{c_2} = \emptyset$;
   $a^{c_2} := \overline{x, b}$.
4. $A^{c_2} := (a^{c_2} \gg \underline{r^{c_2}}) = (\overline{x, b} \gg \underline{Main}) = (\overline{x, y},\ (\overline{x, a} \gg \textbf{call}\ q)) = \overline{x, y}, \overline{x, b}, \overline{a, c}$
   as currently $(\overline{x, a} \gg \textbf{call}\ q) = A^{c_1} = \overline{x, y}, \overline{x, b}, \overline{a, c}$;
   $a^{c_1} := \overline{x, a}$.
5. $A^{c_0} := (a^{c_0} \gg \underline{r^{c_0}}) = (\emptyset \gg \underline{Main}) = (\overline{x, y},\ (\overline{x, a} \gg \textbf{call}\ q)) = \overline{x, y}, \overline{x, b}, \overline{a, c}$
   as currently $(\overline{x, a} \gg \textbf{call}\ q) = A^{c_1} = \overline{x, y}, \overline{x, b}, \overline{a, c}$;
   $a^{c_1} := \overline{x, a}$.
6. $A^{c_1} := (a^{c_1} \gg \underline{r^{c_1}}) = (\overline{\underline{x, a}} \gg \underline{q}) = ((\overline{x, b} \gg \textbf{call}\ Main),\ \overline{x, b} \gg (a := c)) = \overline{x, y}, \overline{x, b}, \overline{a, c}$
   as currently $(\overline{x, b} \gg \textbf{call}\ Main) = A^{c_2} = \overline{x, y}, \overline{x, b}, \overline{a, c}$;
   $a^{c_2} := \overline{x, b}$.
7. $A^{c_2} := (a^{c_2} \gg \underline{r^{c_2}}) = (\overline{x, b} \gg \underline{Main}) = (\overline{x, y},\ (\overline{x, a} \gg \textbf{call}\ q)) = \overline{x, y}, \overline{\overline{x, b}}, \overline{a, c}$
   as currently $(\overline{x, a} \gg \textbf{call}\ q) = A^{c_1} = \overline{x, y}, \overline{x, b}, \overline{a, c}$;
   $a^{c_1} := \overline{x, a}$.

Items 8–10 (not shown) are the same as items 5–7 respectively, so $A^{c_0}, A^{c_1}, A^{c_2}$ given in items 5–7 are stabilized and equal to the maximal output alias relations sought for; the maximal input alias relations are $a^{c_0}, a^{c_1}, a^{c_2}$ given in items 1, 7, 6 respectively.

IV. Conclusion and future work

We are elaborating on the alias calculus and in this research-in-progress report presented how aliases might be used in Hoare-style reasoning, derived at what program points we were to compute aliases and proposed an algorithm that computed the required aliases.

Future work includes:

- elaborating on the overall process of Hoare-style reasoning using the alias calculus;
- precisely defining the aliasing semantics of (possibly mutually recursive) procedures; optimizing Algorithm 1, which computes alias relations for calls to (possibly mutually recursive) procedures; proving its correctness; and adopting the algorithm for qualified calls;
- coping with the infinity of some alias relations in the object-oriented alias calculus (e. g., after assigning $cur := first$ and then iterating $cur := cur.next$, the variable cur may be aliased to any element of the infinite set described by the regular expression $first(.next)^*$).

Acknowledgment

The author would like to thank his scientific supervisor Professor Bertrand Meyer, the head of the Software Engineering Laboratory at Saint Petersburg National Research University of Information Technologies, Mechanics and Optics, and Alexander Kogtenkov for helpful discussions. The Software Engineering Laboratory is supported by a grant from Mail.Ru Group.

The author is grateful to the anonymous referees for their useful comments on the original version of this paper.

References

[1] B. Meyer, "Towards a theory and calculus of aliasing," *Journal of Object Technology*, vol. 9, no. 2, pp. 37–74, 2010.

[2] ——, "Steps towards a theory and calculus of aliasing," *Int. J. Software and Informatics*, vol. 5, no. 1-2, pp. 77–115, 2011.

[3] ——, "Towards a calculus of object programs," *CoRR*, vol. abs/1107.1999, 2011.

[4] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *LICS*. IEEE Computer Society, 2002, pp. 55–74.

[5] S. Sagiv, T. W. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 3, pp. 217–298, 2002.

[6] D. G. Clarke, J. M. Potter, and J. Noble, "Ownership types for flexible alias protection," *SIGPLAN Not.*, vol. 33, no. 10, pp. 48–64, Oct. 1998. [Online]. Available: http://doi.acm.org/10.1145/286942.286947

[7] I. T. Kassios, "Dynamic frames: Support for framing, dependencies and sharing without restrictions," in *FM*, ser. Lecture Notes in Computer Science, J. Misra, T. Nipkow, and E. Sekerinski, Eds., vol. 4085. Springer, 2006, pp. 268–283.

[8] Eiffel verification environment. [Online]. Available: http://eve.origo.ethz.ch

[9] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 583, 1969.

[10] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis (2. corr. print)*. Springer, 2005.