Alias calculus, change calculus and frame inference

Alexander Kogtenkov[a,c], Bertrand Meyer[a,b,c], Sergey Velder[c]

[a]*Eiffel Software, 5949 Hollister Avenue, Goleta, California 93117 USA*
[b]*ETH Zurich, Chair of Software Engineering, Clausiusstrasse 59, RZ Building, 8092 Zurich, Switzerland*
[c]*NRU ITMO, Software Engineering Laboratory, Kronverkskiy pr., 49, Saint Petersburg, Russia*

Abstract

Alias analysis, which determines whether two expressions in a program may reference to the same object, has many potential applications in program construction and verification. We have developed a theory for alias analysis, the "alias calculus", implemented its application to an object-oriented language, and integrated the result into a modern IDE. The calculus has a higher level of precision than many existing alias analysis techniques.

One of the principal applications is to allow automatic *change analysis*, which leads to inferring "modifies clauses", providing a significant advance towards addressing the Frame Problem. Experiments were able to infer the "modifies" clauses of an existing formally specified library. Other applications, in particular to concurrent programming, also appear possible.

The article presents the calculus, the application to frame inference including experimental results, and other projected applications. The ongoing work includes building more efficient model capturing aliasing properties and soundness proof for its essential elements.

Keywords: Verification, Alias analysis, Alias calculus, Change calculus, Frame inference, Object-oriented, Static analysis

Email addresses: `alexk@eiffel.com` (Alexander Kogtenkov), `Bertrand.Meyer@inf.ethz.ch` (Bertrand Meyer), `velder@rain.ifmo.ru` (Sergey Velder)

1. Overview

A largely open problem in program analysis is to obtain a practical mechanism to detect whether the runtime values of two expressions can become *aliased*: point to the same object. "Practical" means that the analysis should be:

- *Sound*: if two expressions can become aliased in some execution, it will report it.
- *Precise* enough: since aliasing is undecidable, we cannot expect completeness; we may expect false positives, telling us that expressions may be aliased even though that will not happen in practice; but there should be as few as possible.
- *Realistic*: the mechanism should cover a full modern language.
- *Efficient*: reasonable in its time and space costs.
- *Integrated*: usable as part of an integrated development environment (IDE), with an API (abstract program interface) making it accessible to any tool (compiler, prover. . .) that can take advantage of alias analysis.

The present discussion considers "may-alias" analysis, which reports a result whenever expressions *may* become aliased in *some* executions. The "must-alias" variant follows a dual set of laws, not considered further in the present paper.

The papers [13, 14] introduced the **alias calculus**, a theory for reasoning about aliasing through the notion of "alias relation" and rules determining the effect of every kind of instruction on the current alias relation. We have refined, corrected and extended the theory and produced a new implementation fully integrated in the EVE (Eiffel Verification Environment) open-source IDE [1] and available for download at the given URL. In the classification of [6, 8, 21], the analysis is untyped, flow-sensitive, path-insensitive, field-sensitive, interprocedural, and context-sensitive.

The present paper describes the current state of alias analysis as implemented. It includes major advances over [14]:

- The calculus and implementation cover most of a modern OO language.

- The implementation is integrated with the IDE and available to other tools.
- The performance has been considerably improved.
- A part of the calculus has been proved sound, mechanically, using Coq.
- An error affecting assignment handling in an OO context that has been corrected (see Section 3).
- New applications have been developed, in particular to *frame inference.*

Frame inference relies on a complement to the alias calculus: the *change calculus*, also implemented, which makes it possible to infer the "modifies clause" of a routine (the list of expressions it may modify) automatically. Applied to an existing formally specified library including "modifies" clauses, the automatic analysis yielded all the clauses specified, and uncovered a number of clauses that had been missed, even though the library, intended to validate new specification techniques (theory-based specification), had been very carefully specified.

Section 2 presents the general assumptions and section 3 the calculus. Section 4 introduces the change calculus and automatic inference of frame conditions. Section 5 describes the implementation and the results it yielded in inferring frame conditions for a formally specified library. Section 6 discusses related work. Section 7 presents the ongoing work concerning other applications, such as deadlock detection, and a new theoretical basis. Section 8 is a conclusion and review of open problems.

2. The mathematical basis: alias relations

E denotes the set of possible expressions. An expression is a path of the form $x.y.z....$ where x is a local variable or attribute of one of the classes of the program, or $Current$, and $y, z, \ldots$, if present, are attributes. Variables and attributes are also called "tags". $Current$ represents the current object in OO computation (also known as "this" or "self").

An alias relation is a binary relation on E (that is, a member of $\mathcal{P}(E \times E)$) that is symmetric and irreflexive. If r is an alias relation and e an expression, r/e denotes the set consisting of all elements aliased to e, plus e itself: $\{e\} \bigcup \{x \in E \mid [x, e] \in r\}$. An alias relation may be infinite; for example the instruction $a.set\_u(a)$, where $a.set\_u$ assigns the u field, causes a to become aliased to $a.u.u. \ldots$, with any number of occurrences of the tag u; in this case the set r/x is also infinite.

Alias relations are in general not transitive, since expressions can receive different aliases on different branches of a program: $\textsf{if } c \textsf{ then } x := y \textsf{ else } x := z \textsf{ end}$ yields an alias relation that contains the pairs $[x, y]$ and $[x, z]$ but not necessarily $[y, z]$.

To define the meaning of alias relations, we note that the calculus cannot be complete, since aliasing is undecidable for a realistic language. It must of course be sound; so the semantics (section 7.2) is that if an alias relation r holds in a computation state, then any pair of expressions $[e, f]$ *not* in r is *not* aliased (i.e. $e \neq f$) in that state. Incompleteness means that some pairs of expressions might appear in r even though they cannot actually become aliased.

A convenient way to write an alias relation is the **canonical form** $A, B, C, \ldots$ where each element is a set of expressions $e, f, \ldots$, none of them a subset of another; such a set is written $\overline{e, f, \ldots}$. For example the above conditional instruction, starting from an empty alias relation, yields $\overline{x, y}, \overline{x, z}$. More generally $\overline{A}$, for a list or set of expressions A, denotes $A \times A - Id_A$, i.e. the "de-reflexived" (by removing any pair $[x, x]$) set of all pairs of elements in A.

3. The alias calculus

The alias calculus is a set of rules defining the effect of executing an instruction on the aliasings that may exist between expressions. Each of these rules gives, for an instruction p of a given kind and an alias relation r that holds in the initial state, the value of $r \gg p$, the alias relation that holds after the execution of p.

By itself the alias calculus is automatic: it does not require programmer annotations. Since it only addresses a specific aspect of program correctness, it may have to be used together with another technique of program verification, in particular Hoare-style semantics, which uses annotations. The relation goes both ways:

- If a routine's postcondition expresses a non-aliasing property $x \neq y$, the calculus can prove it (using lighter techniques than the usual axiomatic proof mechanisms).

- Conversely, the alias calculus may need to rely on properties established separately. In particular, it ignores conditional expressions; so in computing $r \gg$ (if $x \neq y$ then $z := x$ end) where r contains $[x, y]$, it will yield a relation containing $[x, z]$ even though x and z cannot actually become aliased. In many cases the resulting imprecision is harmless, but its removing requires help from other techniques. The solution takes the form of an instruction cut, such that $r \gg$ (cut x, y) is obtained from r by removing the pair $[x, y]$ and the pairs $[x.e, y.e]$ for any expression e.

To support this complementarity with other verification techniques, the alias calculus uses the following conventions:

- It ignores the conditions in conditionals, writing them just then p else q end, and loops, written loop p end.
- It includes an instruction cut x, y, expressing that $x \neq y$. The cut instruction is not intended for use by programmers; rather, it is an annotation that can be inserted by another verification tool, such as a Hoare prover, whenever more precision is required and a condition is easy to establish. The most obvious example is a conditional instruction if $x \neq y$ then p end, which would normally be understood as just then p end for the alias calculus; to make the analysis more precise, a verifier (even a very simple-minded one) can turn it into then cut x, y; p end for the benefit of the alias analyzer. Note that the cut instruction is a safety valve designed for future use; in practice we have not encountered the need for it so far.

Here now is the calculus. The rules for control structures are:

$r \gg (p; q) = (r \gg p) \gg q$

$r \gg$ (then p else q end) $= (r \gg p) \bigcup (r \gg q)$

$r \gg$ (loop p end) $= t_N$, for the first N such that $t_N = t_{N+1}$, where $t_0 = r$ and $t_{n+1} = t_n \bigcup (t_n \gg p)$ (see below about finiteness)

For a creation instruction ($x :=$ new $(\ldots)$ in Java style) and a "forget" ($x :=$ null):

$r \gg$ (create x) $= r - x$

$$r \gg (\mathsf{forget}\ x) = r - x$$

where "–" is set difference generalized to elements (x stands for $\{x\}$), relations and paths: $r - x$ is obtained from r by removing all pairs of which one element is $x.e$ (or $e_0.x.e$ where e_0 is aliased to $Current$ in r). For "cut" we have:

$$r \gg (\mathsf{cut}\ x, y) = r - \overline{x, y}$$

The rule for unqualified routine call, with l as the list of actual arguments, is:

$$r \gg (\mathsf{call}\ f(l)) = r[f^{\bullet} : l] \gg |f|$$

where $f^{\bullet}$ is the formal argument list of f, $r[u : v]$ the relation r with every element of the list v substituted for its counterpart in u, and $|f|$ the body of f.

The rule for qualified calls relies on a notion of "negative variable" [14, 15] to transpose the context of the call to the context of the caller:

$$r \gg (x.\mathsf{call}\ f(l)) = x\bullet((x'\bullet r) \gg \mathsf{call}\ f(x'\bullet l))$$

where x' is the "negation" of x, with $x'.x = Current$ and "$\bullet$" is generalized distributively to lists ($x\bullet\langle a, b, \ldots\rangle = \langle x.a, x.b, \ldots\rangle$), sets and relations.

The main instruction that creates aliasings, removing previous ones, is reference assignment: $t := s$. The assignment rule given in [14] was unsound (in the cases when any expression of r/s starts from expression of the form $e.t$ where e is aliased to $Current$ in r). The new rule has been proved sound in the semantics discussed in section 7.2. It can be expressed in several ways, of which the easiest to understand uses a fresh variable ot (for "$\mathsf{old}\ t$"):

$$r \gg (t := s) = \mathsf{given}\ r_1 = r[ot = r/t]\ \mathsf{then}\ (r_1 - t)[t = r_1/s - t] - ot\ \mathsf{end}$$

with $r[x = u]$ denoting the relation r augmented with pairs $[x, y]$ where y is an element of u, and made *dot-complete* [14], that is to say extended with the following pairs: $[u.a, v]$ for any t, u, v and a where $[t, u]$ and $[t.a, v]$ are alias pairs; and $[t.a, u.a]$ for any t, u, a where $[t, u]$ is an alias pair and a is in the domain of t.

In words, the assignment rule works as follows. Consider an instruction $t := s$ being applied to an alias relation r. First, assign variable ot to t and compute the resulting alias relation r_1. It is obtained from r by augmenting it with pairs $[ot, y]$ for all $y \in r/t$ (remember that r/t contains all aliases of

t in r, including t) and making it dot-complete. Then remove from r_1 all "redundant" aliases of expressions starting from t and similar. After that, assign t to s, that is to say add to the resulting alias relation all pairs $[t, y]$ where y is a "non-redundant" alias of s in r_1. Make the resulting alias relation dot-complete, and, finally, remove "redundant" aliases of ot.

As an example, we compute $r \gg (t := t.u.v)$ for $r = \{[a.e, t.u.e] \mid e \in E\}$. First, $r/t = \{t\}$, so $r_1 = \{[ot.e, t.e], [a.e, t.u.e], [a.e, ot.u.e], [t.u.e, ot.u.e] \mid e \in E\}$, and $r_1 - t = \{[a.e, ot.u.e] \mid e \in E\}$. Next, $r_1/t.u.v = \{t.u.v, ot.u.v, a.v\}$, $r_1/t.u.v - t = \{ot.u.v, a.v\}$, and $(r_1 - t)[t = r_1/t.u.v - t] = \{[a.e, ot.u.e], [a.v.e, t.e], [a.v.e, ot.u.v.e], [t.e, ot.u.v.e] \mid e \in E\}$. Therefore, $r \gg (t := t.u.v) = \{[a.v.e, t.e] \mid e \in E\}$.

Another example demonstrates how the alias calculus works for programs with compound constructions. Consider a program in Eiffel: if x /= y then $x.set\_a(b)$ else $y :=$ Void end. This program has one-argument routine $set\_a(b)$ performing $a := b$ with respect to local object. Applying transformations described above to this program starting from the alias relation $R = \overline{x, y}$ yields the following alias relations at every step:

```
                         $R = \overline{x, y}$
then                     $R = \overline{x, y}$
    cut $x, y$           $R = \varnothing$
    $x$.call $set\_a(b)$ $R = \{[x.a.e, b.e] \mid e \in E\}$
else                     $R = \overline{x, y}$
    forget $y$           $R = \varnothing$
end                      $R = \{[x.a.e, b.e] \mid e \in E\}$
```

The most intriguing line in this example is the instruction x.call $set\_a(b)$. The alias calculus rule for this instruction starting from $R = \varnothing$ works as follows: since $x'.R = \varnothing$, we compute $x.(\varnothing \gg \text{call } set\_a(x'.b)) = x.(\varnothing \gg a := x'.b) = x.\{[a.e, x'.b.e] \mid e \in E\} = \{[x.a.e, x.x'.b.e] \mid e \in E\} = \{[x.a.e, b.e] \mid e \in E\}$.

Since alias analysis cannot be complete, the calculus introduces possible imprecisions (over-approximations); it is important to understand where they actually lie. In fact, the above rules are precise. Over-approximations come from ignoring conditions in conditionals and loops, such as c in if c then a else b end. It is possible to remove some imprecision of this kind by introducing cut instructions (normally, as noted, not manually but as annotations generated by a verifier).

The implementation of the calculus introduces another source of possible imprecision. In an OO language with unbounded runtime object structures, the alias relation may be infinite. To stick to finite structures the implementation must cut off the graph. The first idea [14] is to limit ourselves to M, the maximum length of a path appearing in an expression of the program (including contracts, especially postconditions). This is, however, not sufficient; in a case such as:

$a := \mathit{first};\ \ a := a.\mathit{right};\ a := a.\mathit{right};\ \ldots$ — n times

$b := \mathit{first};\ \ b := b.\mathit{right};\ b := b.\mathit{right};\ \ldots$ — n times

where $n > M > 1$, the expressions a and b, both of length $< M$, become aliased to each other through being both aliased to an expression of length greater than M that does not appear in the program: *first.right.right....* (n "*right*" tags). A similar problem arises for code containing loops:

$a := \mathit{first};$ loop $a := a.\mathit{right}$ end;

$b := \mathit{first};$ loop $b := b.\mathit{right}$ end;

The implementation and the formal model use a maximum path length $L \geq M$ and treat any expressions longer than L as aliased to all expressions. This technique introduces imprecision but retains soundness. In the future it may be improved using type information (in a statically typed language e and f can only be aliased if their types are compatible; also in polymorphic version of the qualified call rule we replace the resulting alias relation by the union of similar alias relations for all features corresponding to inherited classes). Unlike some of the approximations found in the alias analysis literature, where the equivalent of L is very small, our L can run into large values.

4. The change calculus and frame condition inference

One of the key problems of software verification, still largely open for OO programs, is frame analysis: determining what an operation does *not* change. Current solutions, following in part from tools such as ESC/Java [2] and its successors, assume that the programmer writes a "modifies clause" listing the expressions whose value may change. (As a matter of syntactic taste we prefer the keyword "only" to "modifies", since the goal is not to list expressions that *will* change, but to specify that any expression not listed will not change.) Writing such clauses is, however, tedious. It is hard enough to convince programmers to state what their program does; forcing them in addition to specify all that it does not do may be a tough sell. We find it desirable, as much as possible to infer the "modifies" clauses.

The alias calculus opens the way to such an approach by enabling a *change calculus* (as an abbreviation for *may-change calculus*) which, for any instruction p, yields $\underline{p}$, the set of expressions whose value may change as a result of executing p. Like the alias calculus, the change calculus is an over-approximation: for soundness $\underline{p}$ must include anything that changes, but conversely an expression might appear in $\underline{p}$ and not change in some executions of p, or even (as a sign of our incompetence, inevitable because of undecidability) in none of them. The basic rules of the calculus are (r is the alias relation in the initial state, r/x is the set of aliases of x plus x itself, and "." distributes over sets):

$$\begin{array}{lll}
\underline{t := s} & = (r/Current).t & \\
\underline{p; q} & = \underline{p} \bigcup \underline{q} & \\
\underline{\textsf{then } p \textsf{ else } q \textsf{ end}} & = \underline{p} \bigcup \underline{q} & \text{— same as for ";"} \\
\underline{\textsf{loop } p \textsf{ end}} & = \underline{p} \bigcup \underline{p^2} \bigcup \underline{p^3} \bigcup \ldots & \text{— limited to } L \text{ elements as discussed} \\
\underline{\textsf{call } f(l)} & = \underline{|f|}[l : f^\bullet] &
\end{array}$$

The most important rule, requiring alias analysis, is for qualified calls:

$$\underline{\textsf{call } x.f(l)} \quad = (r/x) \; \boldsymbol{.} \; \underline{\textsf{call } f(x'.l)}$$

where, as before, "**.**" distributes over sets and $y.x' = Current$ if x and y are aliased in r. The rule states that for any u that f may change, call $x.f(l)$ may change not only $x.u$ but also $y.u$ for y aliased to x.

The change calculus, implemented on top of the alias calculus thanks to this rule, enables us to infer frame conditions. This inference is a possible over-approximation. It makes it possible to verify programmer-supplied "modifies" clause in the following way. Let p_c be the set of expressions that can change as a result of the execution of an instruction p, typically a routine call. Let p_m be the list of expressions in the "modifies" clause. The clause is sound if and only if

$$p_c \subseteq p_m \tag{1}$$

For theoretical reasons (undecidability) and practical ones (tool limitations), the verification cannot compute p_c exactly; instead it computes $\underline{p}$. Assuming soundness of the change calculus (and hence of the alias calculus), we have the guarantee that

$$p_c \subseteq p \tag{2}$$

In other words, $\underline{p}$ is a possible over-approximation of the actual change set. Then if a tool such as our implementation is able to compute $\underline{p}$, a compiler can examine the program and its annotations to ascertain the property

$$\underline{p} \subseteq p_m \tag{3}$$

which guarantees (1) and hence the correctness of the "modifies" clause.

In our work towards an integrated development and verification environment as discussed in next section, we intend, for the reasons mentioned above, not to include syntactic support for a "modifies" (or only) clause. Instead we simply consider that any expression not listed in the postcondition (ensure) of a routine must remain unchanged. An informal survey of specifications in JML libraries validated this approach by indicating that in the practice of specification every expression e listed in a "modifies" clause *also* appears in the postcondition. For any exceptions to this observation it is always possible to include a special predicate *involved* (e).

This convention has not yet been applied on a large scale. Until it is, we are validating the calculus on code with explicit "modifies" clause, as discussed in section 5.

5. Implementation, and results of frame inference

The alias and change calculi described in previous sections have been fully implemented. Earlier papers [13, 14] described a prototype stand-alone implementation. The present implementation is integrated in EVE [1], the research version of EiffelStudio, a modern integrated IDE covering the full Eiffel language. On a standard laptop computer, the time to analyze a class from a kernel library ranges from less than a second for simple classes to 7 minutes for a two-way linked tree class (about 4.5 seconds per feature) with a naïve implementation that recomputes the alias relation from scratch for every analyzed feature without any optimization to avoid repetitive analysis. We are working to improve the performance so as to allow immediate user feedback even for large classes.

To assess the approach we performed change analysis on a formally specified library, EiffelBase+ [19]. The library has the attraction of providing "*full contracts*" that specify all properties; for example the postcondition of a "push" operation for stacks states not only that the number of elements has been incremented by one and that the new top is the routine's argument, but also that the previous elements remain. EiffelBase+ also has the characteristic of having been written very carefully, since it is intended to support full verification.

EiffelBase+ currently includes "modifies" clauses. Since the specification style relies on mathematical "model queries" (theory-based specification, also known as specification variables [7]), these clauses list such queries, not directly the program attributes (fields). An example model query, for class *STACK*, is *sequence*, which gives the associated sequence of elements. Running the analysis required mapping attributes to model queries. In most cases the correspondence is straightforward: many model queries map directly to attributes. In a few cases, the model query has no direct attribute counterparts; for example, the model query *sequence* of *LINKED_LIST* is computed by traversing all elements of the list.

We ran the frame inference on 36 classes with 278 "modifies" clauses, detecting a number of missing or different "modifies" specifications; for example, the analysis reports that routines *disjoint* and *is_subset* of a class *ARRAYED_SET* can modify the attribute *index*, not listed in the "modifies" clause. The full results with detailed analysis of found differences are available at http://sel.ifmo.ru/results/alias/EiffelBase+/.

For 614 analyzed features, 592 (96%) "modifies" clauses could be mapped from model to source code. For that code the analysis yielded 100% of the needed "modifies" clauses. The rest (4%) relied on an Eiffel-specific mechanism, which the analysis does not yet support: redeclaring a function as an attribute in a descendant. The summary of the analysis is given in Table 1.

The analysis reported more changed values than specified in the "modifies" clauses. We manually checked that 7 of the inferred clauses indeed reveal unique errors showing a discrepancy between specification and implementation. This result is all the more significant that EiffelBase+, as noted, is carefully written and designed for formal verification; the library has been extensively tested as reported in [19]. (A testing effort using the AutoTest tool for Eiffel [11], posterior to the release and independent from the present work, found 5 of the errors, but missed the other two.)

The analysis also detected 7 unnecessary "modifies" specifications: values listed in the specification but not actually changed by the implementation. Four of these were simply superfluous and could be removed. The remaining 3 were inherited "modifies" specifications; further investigation revealed that they reflected inconsistencies caused by underspecified ancestor contracts.

Table 1: Frame inference experimental results

614	Total number of features			
	22	Not mapped due to implementation constraints		
	592	Mapped		
		514	Code and "modifies" clauses matches	
		7	Missing "modifies" clauses (code/contract discrepancy)	
		7	Unnecessary "modifies" clauses	
			4	Redundant ("modifies" clauses can be removed)
			3	Redundant in descendants
		64	False positives	
			46	Variable backup-restore (dangerous with exceptions)
			15	Simplistic array representation
			3	Unreachable code

There were 64 (11%) false positives (clauses inferred but not needed). Of these, 3 were found to reflect actual changes but in unreachable code due to defensive programming in the library. The majority, 46, correspond to the case of a value that the code actually changes, after backing it up, but then restores from the backup. Here the change calculus correctly returns that the value has been changed, twice or more in fact, and other mechanisms are required to find out that the changes cancel each other out. However manual inspection shows that such temporary changes are dangerous in presence of exceptions [17]. If a value is not reverted back at the time of an exception, the object may remain in an unexpected state.

The remaining 15 (2.5% of the total) are the genuine false positives; they are due to the implementation's model of arrays, which does not distinguish between changes to array items and to array size, and which we hope to improve.

The experiments yield the following lessons.

1. Ignoring the temporary problem of functions redeclared into attributes, the change calculus reports 100% of expected "modifies" (frame) properties.
2. It succeeded in pointing out missing "modifies" specifications.
3. It also detected unnecessary "modifies" specifications.
4. The number of false positives is limited, and most of them correspond to values actually changed then restored. Better array handling should

entirely limit false positives to this category, plus changes in dead code (which merit attention anyway).

5. If "modifies" specifications rely on model queries (an approach that is not currently dominant, but which we find the most appropriate), the problem remains of mapping attributes to model queries. For the current experiments we performed the mapping manually, but an automatic approach appears possible.

We find these results promising, opening the possibility that automatic alias and change analysis will become a standard component of program verification.

6. Related work

There is a considerable literature on alias analysis, in particular for compiler optimization. We only consider work that is directly comparable to the present approach.

6.1. Alias analysis rules

There are different approaches to compute alias information for programs. All of them, including classic iteration-based variants converging to a fixed point and equation-based techniques as in [16], define a set of rules that help compute alias information. The rules are associated with program elements, expressions and instructions (statements), and specify how they affect the model elements used to compute alias information. In C-like languages this usually includes [5, 9, 16]:

Address-of /Alloc (y = &x)

Load (y = *x)

Copy (x = y)

Store (*x = y)

Here we only mention the differences for the assignment instructions, but depending on the level of the language there could be some more instructions and associated rules. For example, [21] uses an intermediate language, RTL, to perform the analysis.

Many of the earlier approaches address C or languages of that level; the present work has been applied to a full-fledged object-oriented language. In an OO context some of the instructions may become unnecessary. In particular, there is no notion of plain pointers. They are replaced by class fields [27]:

New (x := new O)

Load (y := x.f)

Assign (x := y)

Store (x.f := y)

The rules can be simplified even further when an OO language, such as Eiffel, in line with the information hiding principle, disallows remote modification of an object field (x.u := a must be written x.set_u (a) using a setter set_u); then there is no need for *Store*. The formalism and implementation of the present work rely on that assumption but can be generalized to languages accepting direct setting.

Many earlier approaches are flow-insensitive: in a := b; a := c they will find that a can be aliased to both b and c. Such imprecision is unacceptable for the applications examined in the present work, such as change analysis and frame inference. An example of flow-sensitive analysis is [5], but it too introduces imprecision, in particular in its handling of assignment. As compared to such work the high precision of our approach is obtained at the expense of performance, although we hope to improve it.

The analyses of which we are aware compute the alias information using only instructions that may change the object state or a variable value. It is also useful to introduce constructs that do not change any state, but do change the alias information; they include, as described in section 3, instructions asserting equality such as cut asserting $x \neq y$. (Our work also uses bind, which asserts $x = y$.) We do not know of other alias work using such instructions. They make it possible to take advantage of Hoare-style assertions for alias analysis, rather than simply ignoring them, and may provide a way to combine may-alias and must-alias analysis as suggested in [3].

As in some other inter-procedural analyses [3, 4], the information computed for every routine is recorded for later use to avoid unnecessary recomputation.

6.2. Soundness proof

This paper mentions a partial proof of soundness in section 7.2. A soundness proof for alias analysis appears in [21], using Coq as in the present work. The proof in [21], however, applies to C, through an intermediate language. The proof mentioned here, and the underlying theory (alias calculus), apply directly to the programming language. In addition, that programming language is not C but an OO language.

6.3. Frame condition inference

Automated support for code verification is a well-known problem that has been tackled in the past two decades with increasing success. The approaches range from static analysis [24] to dynamic contract inference [18]. Instead of the contracts in general here we focus on frame conditions. This is similar to the work described in [20], but it is done in the context of a safe object-oriented language. The analysis is performed on a whole program. It might be possible to use frame conditions specified in the source code, e.g. dynamic frames [7] to achieve modularity, but we left it for future research.

According to [19], the completeness of the contracts is an important condition for realistic program verification. The contracts should include not only pre- and postconditions but also "modifies" clauses that list all the data affected by the particular method of a class. It turns out [20] that 90% of such information can be obtained automatically. Our experiments confirm this estimation. However in our setting the most part of inaccuracy was caused by backup-restore operations. This is somewhat close to the caching issues mentioned in [23] where authors traded soundness for usability. Our implementation preserves soundness for the cost of several false positives.

The frame conditions could also be proved with must-alias analysis, or even by applying may-alias and must-alias analyses together as described in [3]. For every attribute x of a class the following property could be checked at routine exit: $x = \mathsf{old}\ x$, where $\mathsf{old}\ x$ stands for the value of x on routine entry. Indeed, must-alias analysis would tell whether this expression is always true. But then the problem is to apply must-alias analysis to all the (possibly nested) attributes of reachable objects and that does not seem practical.

In [17] it is demonstrated that in the presence of exceptions postconditions should be specified in two parts: one for a normal case and one for an exceptional. Our change analysis computes the union for both cases and highlights that backup-restore operations may not be ideal when combined with exceptions. Additional research is required to find most convenient way to express the two cases in specifications.

7. Future work

Alias analysis can have many applications beyond frame inference. Section 7.1 sketches one of them, deadlock detection. Section 7.2 presents a theoretical improvement leading to better performance and precision of alias analysis. Both of these reflect work in progress.

7.1. Deadlock analysis

The SCOOP concurrency model makes no firm difference between computational mechanisms and resources, all captured by the notion of "processor". For example, in the SCOOP solution of "Dining philosophers", both philosophers and forks are objects residing on their own processors. A processor can access objects handled by another processor by explicitly reserving that object's processor.

The SCOOP reservation mechanism reduces the risk of deadlock by reserving any number of objects atomically, through the syntactical device of argument passing: $r(a, b, \ldots)$ reserves all the processors of the objects associated with $a, b, \ldots$. For example a philosopher will execute *eat* (*left_fork*, *right_fork*). It remains possible, however, to create "Coffman deadlocks" whereby a set of processors reserve each other circularly. The difficulty of detecting them is that processors are known from object references, which may be aliased. Alias analysis may help find possible cycles by considering, in every class, any variable which is declared as `separate` (meaning that the object may have a different processor) as aliased to its processor, and looking for cycles in the reservation graph. We are currently implementing this technique.

7.2. Towards a better mathematical basis: alias diagrams

The mathematical basis for the present article is the notion of alias relation. A new model under development, alias diagrams, is intended to improve the rigor of mathematical description and find more effective representation of pointer aliasing properties. Depending on the context, the alias relation or alias diagram may be the more convenient view.

The formalization only includes elements relevant to aliasing; in particular, an object contains only reference fields, since value fields such as integers ("expanded" in Eiffel) can be ignored.

A state in the alias diagram model is a directed rooted graph. Vertices represent objects of the program, and edges represent class fields. Every edge is labeled by a tag corresponding to the name of some class attribute (field). Every vertex has, for any tag, only one outgoing edge labeled by it. The presence of a root reflects the OO context of this work (see also [15]): we treat any expressions and aliases as always relative to a "current" object. The current object is always part of the state.

An alias diagram is a multigraph (labeled directed graph, but with the possibility of more than one edge labeled by same tags going from a given vertex to another) where: each vertex represents an object (abstracted from execution objects); there is a distinguished vertex called the "root" , representing the current object; and every edge is labeled by a tag x indicating (in the style of shape analysis [25, 22]) that from any of the objects represented by the source node, the reference x can point to one of the objects represented by the target node.

Every rooted path in an alias diagram corresponds to some expression, and we can put the terminal vertex of this path to correspondence with this expression as well.

An alias diagram represents an alias relation, with the convention that e and f are aliased if and only if one of the following holds: for some node V, there are paths labeled e and f both leading to V; or e is $e_1.t$ and f is $f_1.t$, where t is a tag and e_1 and e_2 are expressions (recursively) aliased. As a special case of the first variant, $Current$ is aliased to a path e if and only if e leads to the root.

Of most interest are alias diagrams in "canonical form" (closely connected to the canonical form of alias relations seen above), where all vertices are *reachable* and *necessary.* A vertex is **reachable** if there is a path from the root to it (the path may be empty — $Current$ — so that the root is always reachable); it is **necessary** if it is the root or has at least two incoming edges or at least one outgoing edge. For the associated alias relation, unreachable and unnecessary vertices are irrelevant; conversely, if a vertex V is reachable and necessary, either V or one of its successors, direct or indirect, has two or more paths leading to it, and hence is relevant for the alias relation. The **canonical form** of a diagram is its maximal canonical sub-diagram.

For any state S, if we choose its root object as "current", there is an

associated alias diagram D_S: the canonical form of S treated as alias diagram.

To define the semantics of alias diagrams, we say that a diagram D **holds** in a state S for an object — written $holds(S, D)$ — if there is an injective morphism from D_S to D preserving the root and the transitions.

The definition of soundness for the alias calculus reflects the conservative (over-approximation) nature of the calculus: it states that if different expressions e and e', defined relative to a state S, have the same value (point to the same object), then the pair $[e, e']$ must be in the alias relation for the state for which the object is "current". There is no reverse implication, which would correspond to a "must-alias" analysis.

The alias relations associated with an alias diagram and its canonical form are the same. Also, $holds(S, D_S)$ is always satisfied.

We can treat instructions as functions mapping states to states. The alias calculus is a set of rules that for any instruction I and alias diagram D yield another alias diagram $D \gg I$. In this framework, soundness is defined as follows:

$$\forall S, D : holds(S, D) \implies holds(I(S), D \gg I)$$

The soundness proof for the alias calculus must establish this property for each kind of instruction I and the corresponding rule in alias calculus.

The rules of the alias calculus in the alias diagram model are just graph transformations. We will give just one example, affecting the core operation: assignment; it should be compared with the alias relation version of the rule in section 3.

Given an instruction $t := e$ where $e = t_1.t_2 \ldots t_n$, the rule can be expressed in terms of alias diagrams as follows. For a diagram D add a new path corresponding to expression e, and for every vertex corresponding to any prefix $t_1.t_2 \ldots t_m$ of e add an edge from it to the vertex of this new path corresponding to the next prefix $t_1.t_2 \ldots t_{m+1}$ of e. Then remove all the edges labeled t from the root and add edges labeled t from the root to all vertices corresponding to expression e in D and to the last vertex of the new path.

The soundness of this rule follows from the monotonicity of operation "$\gg I$" with respect to alias diagrams (for detailed discussion of alias monotonicity see [14]). We checked this proof of it (assuming monotonicity) mechanically using the Coq proof assistant. The proof is available at http://sel.ifmo.ru/results/alias/semantics/.

8. Conclusion

The alias calculus and change calculus, as described here, are implemented as part of the EVE development environment; the reader can try them out by downloading EVE at [1]. A number of challenges remain open:

- Building alias calculus rules for composite constructions in the model of alias diagrams. The rules must be sound and allow efficient implementation.
- Close integration with other verification tools, in particular (in EVE) the Boogie-based AutoProof proof system (taking advantage of the interplay, discussed in section 3, between the automatic alias calculus and annotation-based Hoare-style proofs), and the AutoTest automatic testing mechanism.
- New applications, including deadlock detection, as sketched in section 7.1.
- Better integration of modularity concerns; although the calculus supports modularity, the current implementation has not focused on this aspect.
- Performance improvement; 7 minutes for a large class is acceptable for an initial version, especially if the tools run in the background, but turning change and alias calculus into routine tools of the environment, with immediate feedback, requires a significant performance improvement.
- Human engineering, in particular the development of suitable mechanisms to display the results of alias and change analysis in a form directly meaningful for programmers, and as a tool for suggesting missing contracts.

Among the main benefits of the approach as developed so far, we find the following: it is entirely automatic (with the provision of `cut` and `bind` annotations produced by other tools); it is of much higher precision than many of the existing approaches (the only sources of imprecision being the neglect of conditionals and the approximation of infinite diagrams by finite but large ones); it is based on a simple and (we hope) convincing calculus; its soundness has been partly established; it applies to a full-fledged, practical, modern OO language; and it is implemented as part of a modern IDE. We believe the approach provides a significant practical advance towards the automatic computation of frame properties and other fundamental program properties resulting from the unpleasant but inevitable presence of aliasing in modern programming frameworks.

Acknowledgements. On the occasion of Paul Klint's 65^{th} birthday we are delighted to acknowledge his seminal contributions to programming theory and practice. This work was carried out in the ITMO Software Engineering and Verification Laboratory, as part of a "megagrant" funded by the Mail.ru group. We are grateful to the anonymous referees for excellent comments which led to a considerable improvement of the article.

References

[1] EVE (Eiffel Verification Environment), https://svn.eiffel.com/eiffelstudio/branches/eth/eve

[2] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., Stata, R. "Extended static checking for Java". In PLDI 2002, pp. 234–245 (2002)

[3] Godefroid, P., Nori, A. V., Rajamani, S. K., Tetali, S. D.: Compositional may-must program analysis: unleashing the power of alternation In: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL 2010, pp. 43–56. ACM, New York, NY, USA (2010)

[4] Gorbovitski, M., Liu, Y. A., Stoller, S. D., Rothamel, T., Tekle, T. K.: Alias analysis for optimization of dynamic languages. Proceedings of the 6th symposium on Dynamic languages. DLS 2010, pp. 27–42. ACM, Reno/Tahoe, Nevada, USA (2010)

[5] Hardekopf, B., Lin, C.: Flow-sensitive pointer analysis for millions of lines of code. In: 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization. CGO 2011, pp. 289–298. IEEE (2011)

[6] Hind, M.: Pointer analysis: haven't we solved this problem yet? In: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pp. 54–61. ACM, Snowbird, Utah, USA (2001)

[7] Kassios, I.: Dynamic Frames: Support for Framing, Dependencies and Sharing without Restrictions. In: J. Misra, T. Nipkow, E. Sekerinski (eds.) Formal Methods 2006. LNCS, vol. 4085, pp. 268–283. Springer-Verlag (2006)

[8] Lenherr, T.: Taxonomy and applications of alias analysis. Master thesis. ETH, Department of Computer Science, Institut für Computersysteme (2008)

[9] Lhoták, O., Chung, K.-C. A.: Points-to analysis with efficient strong updates. In: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL 2011, pp. 3–16. ACM, Austin, Texas, USA (2011)

[10] Loginov, A., Reps, T., Sagiv, M.: Automated Verification of the Deutsch-Schorr-Waite Tree-Traversal Algorithm. In: Yi, K. (ed.) Static Analysis 2006. LNCS, vol. 4134, pp. 261–279. Springer Berlin Heidelberg (2006)

[11] Meyer, B., Ciupa, I., Leitner, A., Fiva, A., Wei, Y., Stapf, E. Programs that Test Themselves, IEEE Computer, vol. 42, no. 9, pages 46–55 (2009)

[12] Meyer, B., Kogtenkov, A., Stapf, E.: Avoid a Void: The Eradication of Null Dereferencing. In: Reflections on the Work of C. A. R. Hoare, eds. C. B. Jones, A. W. Roscoe and K. R. Wood, pp. 189–211. Springer-Verlag (2010)

[13] Meyer, B.: Towards a Theory and Calculus of Aliasing, in J. of Object Technology, vol. 9, no. 2, March–April 2010, pp. 37–74 (first version of [14] and superseded by it) (2010)

[14] Meyer, B.: Steps Towards a Theory and Calculus of Aliasing. In: International Journal of Software and Informatics, special issue (Festschrift in honor of Manfred Broy), 2011, pp. 77–116. Chinese Academy of Sciences (2011)

[15] Meyer, B., Kogtenkov, A.: Negative Variables and the Essence of Object-Oriented Programming. Unpublished. http://se.ethz.ch/˜meyer/publications/proofs/negative.pdf (2012)

[16] Nasre, R., Govindarajan, R.: Points-to Analysis as a System of Linear Equations. In: Cousot, R., Martel, M. (eds.) Static Analysis 2011. LNCS, vol. 6337, pp. 422–438. Springer Berlin Heidelberg (2011)

[17] Nordio, M., Calgagno, C., Müller, P., Meyer, B.: A Sound and Complete Program Logic for Eiffel. In: Proceedings of the 47th International Conference, TOOLS EUROPE 2009, pp.195–214. Springer Berlin Heidelberg (2009)

[18] Polikarpova, N., Ciupa, I., Meyer, B.: A comparative study of programmer-written and automatically inferred contracts. In: Proc. 18th Int. symposium on Software testing and analysis, July 2009, pp. 93–104. ACM, Chicago, Illinois, USA (2009)

[19] Polikarpova, N., Furia, C. A., Pei, Y., Wei, Y., Meyer, B.: What Good Are Strong Specifications? In: International Conference on Software Engineering 2013. To appear (2013)

[20] Rakamaric, Z., Hu, A. J.: Automatic Inference of Frame Axioms Using Static Analysis. In: 23rd IEEE/ACM Int. Conf. on Automated Software Engineering, 2008, pp. 89–98.

[21] Robert, V., Leroy, X.: A Formally-Verified Alias Analysis. In: Hawblitzel, C., Miller, D. (eds.) Certified Programs and Proofs 2012. LNCS, vol. 7679, Springer, pp. 11–26.

[22] Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst., 2002, vol. 24, pp. 217–298. ACM (2002)

[23] Sălcianu, A., Rinard, M.: Purity and Side Effect Analysis for Java Programs. In: 6th International Conference, VMCAI 2005. LNCS, vol. 3385, pp. 199–215. Springer Berlin Heidelberg (2005)

[24] Taghdiri, M., Seater, R., Jackson, D.: Lightweight extraction of syntactic specifications. In: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, 2006, pp. 276–286. ACM, Portland, Oregon (2006)

[25] Wies, T., Kuncak, V., Zee, K., Podelski, A., Rinard, M. C.: On Verifying Complex Properties using Symbolic Shape Analysis. In: Workshop on Heap Abstraction and Verification (collocated with ETAPS). CoRR (2006)

[26] Woo, J., Gaudiot, J.-L., Wendelborn, A. L.: Alias Analysis in Java with Reference-Set Representation for High-Performance Computing. Int. Journal of Parallel Programming, 2004, vol. 32, issue 1, pp. 39–76. Kluwer Academic Publishers-Plenum Publishers (2004)

[27] Yan, D., Xu, G., Rountev, A. Demand-driven context-sensitive alias analysis for Java. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. ISSTA 2011, pp. 155–165. ACM, Toronto, ON, Canada (2011)