Science of Computer Programming 97 (2015) 163–172

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

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, CA 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

HIGHLIGHTS

- Change calculus allows inferring "modifies" clause automatically.
- The change calculus is based on alias calculus and covers most of a modern OO language.
- A large part of the calculus has been proved sound, mechanically, using Coq.
- Applied to an existing formally specified library, the analysis uncovered missing "modifies" clauses.

ARTICLE INFO

Article history:
Received 25 September 2013
Accepted 5 November 2013
Available online 12 November 2013

Keywords:
Verification
Alias analysis
Frame inference
Object-oriented
Static analysis

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.

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.

* Corresponding author.
E-mail addresses: alexk@eiffel.com (A. Kogtenkov), Bertrand.Meyer@inf.ethz.ch (B. Meyer), velder@rain.ifmo.ru (S. Velder).

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.11.006

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 [11,12] 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 [6,8,19], 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 [12]:

- 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 $x.y.z.\ldots$ 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: $\{e\} \cup \{x \in E \mid [x, e] \in r\}$. 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: if c then $x := y$ else $x := z$ 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** $\underline{A, B, C}, \ldots$ where each element is a set of expressions $e, f, \ldots$, none of them a subset of another; such a set is written $\underline{e, f, \ldots}$. 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