Bull. Nov. Comp. Center, Comp. Science, 37 (2014), 131–147
© 2014 NCC Publisher

Alias calculus for a simple imperative language with decidable pointer arithmetic[*]

N.V. Shilov, A. Satekbayeva, A.P. Vorontsov

Abstract. Alias calculus was proposed by Bertrand Meyer in 2011 for a toy programming language with a single data type for abstract pointers. This original calculus is a set-based formalism insensitive to the control flow; it is a set of syntax-driven rules how to compute an upper approximation $aft(S, P)$ for aliasing after the execution of a program P for a given initial aliasing S. The primary purpose of our paper is to present a variant of alias calculus for a more realistic programming language with automatic and dynamic memory, regular data and a decidable pointer arithmetic. Our variant is insensitive to the control flow (like the original calculus by B. Meyer), but (in contrast to the original calculus) this calculus is equation-based.

Keywords: aliasing problem, alias calculus, logic of partial correctness.

1. Introduction

1.1. Aliasing problem

In this paper, we present a variant of alias calculus [7], i.e. a syntax-driven procedure for computing the aliasing $aft(S, P)$ after execution of a program P for a given initial aliasing S in such a way that a triple $\{S\}P\{aft(S, P)\}$ be valid (in the sense of Hoare logic of partial correctness).

In general, the aliasing problem is to predict, detect, and/or trace pointers to the same addresses in the dynamic memory. The problem is very critical because of exceptions that may happen in run-time due to improper alias handling. Below are two simple examples of alias-related exceptions[1]:

- `x = new(int); x = new(int)` (memory leak)
- `y = x; free x; free y` (invalid access).

The first example shows the loss of a link to a piece of memory allocated first (which can result in run out of memory if iterated); the second example shows an attempt to free a deleted piece of memory (which can result in an

[*]Supported by Nazarbayev University Seed Grant KF-14/16 *Research of Formal Models for analysis of programs with Dynamic Memory.*

[1]We do not assume any particular programming language and the dynamic memory management but use a pseudo-code and intuitive concept of memory allocation and deallocation.

abnormal program termination immediately). We refer to these two examples as *exceptions* because they may be program errors in an application or a part of functionality of a virus.

If the above fragments cause errors, then it is easy to find them and fix (due to close location of allocation/deallocation operators). But if allocation/dealocation operators are separated by large pieces of code with a complicated modular structure, then detection of errors of these types becomes a complicated problem. In other words, the development of compilers capable of detecting similar exceptions is an important problem from the industrial point of view, as well as from the educational and research perspective (e.g. for the verifying compiler research [4]).

The purpose of aliasing analysis is to control statically the address expressions in a program which can/may point to the same memory location in run-time. Such analysis is intended to find and eliminate the errors in the program that are due to single (like memory leak) or multiple links to pieces of memory (like invalid memory access). In the general settings, the problem is undecidable for a programming language with an expressive pointer (address) arithmetic; however many approximate algorithms have been published that provide a trade-off between the efficiency, accuracy and soundness of the aliasing analysis [9].

There are several attributes that characterize the alias analysis [3], some of them are listed and explained below:

- flow-sensitivity,
- context-sensitivity,
- heap modeling,
- alias representation.

While the flow-sensitive analysis usually computes aliases for all control points in a program, the flow-insensitive analysis computes aliasing for a program as a whole. Context-sensitivity is about function/procedure calls and it means whether the context of a call is taken into consideration or not. Analysis may be founded on different models of the dynamic memory (the *heap*): it may be a data structure consisting of cells with abstract addresses capable to save arbitrary data, or a collection of cells indexed by integers to store primitive data values only, etc. Aliasing may be presented by equalities, sets of synonyms, or somehow else.

How can we measure the precision of the alias analysis? The most straightforward way is the so-called direct metric, the average number of memory locations that may be aliased to each address expression in the program. But this metrics has some drawbacks. For example, the number of locations aliased to an expression greatly depends on the heap granularity:

if the heap is represented by a single indivisible object, then all expressions point to one object (the heap).

After decades of research and development, there are still challenges in the alias analysis [3, 9]:

- scalability vs. precision;
- flow- and context sensitivity;
- object-oriented languages;
- libraries and low-level functions,
- multithreaded programs.

Due to the above and other reasons, a new research on the alias analysis emerges (e.g. [2]). In particular, alias calculi proposed by Bertrand Meyer [7] are a new approach to aliasing research. Three variants of alias calculus for toy imperative languages with a single data type for abstract pointers are presented in [7]; these calculi are set-based formalisms without address arithmetic insensitive to the control flow and context.

The primary purpose of our paper is to present an alias calculus for a more realistic programming language with the automatic and dynamic memory, regular data, and addresses (with a decidable address arithmetic). The calculus is a revision of a preliminary variant that has been published recently [10]: these two variants differ in representation of an allocated/accessible memory and are provided by different aft-transformers. Both variants of calculus (in this paper as well as in [10]) are (currently) insensitive to the control flow (like the original calculus by B. Meyer), but our calculi are equation-based (in contrast to the original calculus).

The rest of the paper is organized as follows. The next subsection sketches the alias calculus for a toy programming language **E0**, developed by B. Meyer in [7]. Then in Section 2 we introduced the programming language MoRe, its formal syntax and structural operational semantics (SOS); this language is more realistic than **E0** and may be considered as a dialect of the programming language used in [8] for semantics of Separation Logic. A variant of alias calculus for this language is presented in Section 3. In the concluding Section 4, we discuss how to use this calculus to detect memory leaks and invalid access.

1.2. The alias calculus for E0

Let V be an arbitrary finite (fixed) alphabet whose elements are called (pointer) variables. An alias relation on V is any symmetric and irreflexive binary relation on V. Any alias relation S on V can be interpreted as information (knowledge) about which of these variables may point to the same storage (memory) location.

For any binary relation S on V, let[2] $\overline{S}$ be the symmetric and irreflexive closure of S:

$$\overline{S} = \{(x,y),(y,x) \in V^2 \ : \ (x,y) \in S \text{ and} y \not\equiv x\}.$$

(We reserve the symbol $\equiv$ for the syntactic identity and $\not\equiv$ of the syntactic difference for alphabet symbols and strings.)

For any alias relation S, and any variable x, let

- $(S \setminus -x) = \overline{\{(y,z) \in S \ : \ y \not\equiv x \text{ and } z \not\equiv x\}}$,
- $(S/x) = \{y \in V \ : \ y \equiv x \text{ or } (y,x) \in S\}$.

For any alias relation S, let $cnd(S)$ be the system (i.e. conjunction) of *differences* (i.e. the inequalities) $x \neq y$ for all $x, y \in V$, $x \not\equiv y$ and $(x,y) \notin S$, i.e.

$$cnd(S) = \wedge_{x,y \in V,\ x \not\equiv y,\ (x,y) \notin S}\, x \neq y;$$

it is easy to see that the constructor *cnd* possesses the monotonicity property: for any alias relations S_1 and S_2, if $S_1 \subseteq S_2$, then $cnd(S_1) \to cnd(S_2)$.

The programming language **E0** has a single data type for pointers only. The syntax of the language is defined as follows:

$$P ::= \ skip \mid forget(V) \mid create(V) \mid V := V \mid \\ (P;P) \mid P^N \mid (then\ P\ else\ P) \mid (loop\ P),$$

where

- V is a metavariable for the set of variables (that was fixed above),
- N is a metavariable for natural numbers in any fixed notation.

As we already stated in Introduction, the alias calculus is a set of syntax rules which work with formulas of the type $aft(S,P)$, where P is a program, S is an alias relation on the set V of address variables, and aft (abbr. from *after*) denotes the transformer of alias relations. The alias calculus allows us to determine (by forward reasoning) the upper (over-) approximation $aft(S,P)$ for the alias relation after execution of a program P for a given initial aliasing S; it is possible to say in terms of Hoare logic that the calculus should guarantee correctness for the triple

[2]In this section, we use a slightly modified notation adopted from the original paper on the alias calculus [7].

$$\{cnd(S)\}\ P\ \{cnd(aft(S,P))\}.$$

The alias calculus for **E0** and its informal operational semantics follow below.

- $aft(S, skip) = S$ because *skip* is the empty operator.
- $aft(S, forget(x)) = aft(S, create(x)) = S\backslash - x$, i.e. memory deallocation and allocation operators have the same effect on an alias relation because after these operations the variable x is not an alias to any other variable.
- $aft(S, x := y) = \overline{(S\backslash - x) \cup \{x\} \times ((S\backslash - x)/y)}$, i.e. in the result of the assignment $x := y$ the address variable x forgets all its former aliases and becomes an alias to all aliases of the variable y.
- $aft(S,\ (\alpha;\beta)) = aft(aft(S,\alpha),\ \beta)$, i.e. the sequential composition of programs means the sequential application of programs.
- $aft(S,\alpha^0) = S$ and $aft(S,\alpha^{n+1}) = aft(aft(S,\alpha^n),\ \alpha)$ for every $n \geq 0$, i.e. the n-fold iteration (repetition) α^n is the n-fold sequential composition.
- $aft(S,\ then\ \alpha\ else\ \beta\ end) = aft(S,\alpha) \cup aft(S,\beta)$, i.e. $then - else$ is a nondeterministic choice of either of the two branches.
- $aft(S, loop\ \alpha) = \bigcup_{n\geq 0} aft(S,\alpha^n)$, i.e. a loop is a nondeterministic iteration.

2. Programming language MoRe

In this section, we present a programming language MoRe that may be considered as a dialect of the programming language used to define Separation Logic in [8]; the acronym MoRe stays for *More Realistic.*

The language has two data types that are called *addresses* and *integers* with an implicit type casting from integers to addresses. The integer data type is *explicit* while the address data type is *implicit*: only the integer values are depictable and all variables in MoRe are integer by default, while the address values result from integer values after the implicit type casting, and integer expressions are interpreted as addresses only in a special syntactic context.

The address data type in MoRe is any (finite or infinite) set of values ADR with constants that are called *zero* and *one* (conventionally denoted by 0 and 1 in the meta-theory of MoRe), operations that are called *addition* and *subtraction* (conventionally denoted by $+$ and $-$ in the meta-theory of MoRe) such that $(ADR, 0, 1, +, -, =)$ is a commutative additive semigroup with a decidable first-order theory T_{ADR}. Examples include $\mathbb{Z}_m$, the ring of residuals modulo any fixed positive m, Presburger arithmetic, etc.

Let us remark that T_{ADR} is a complete theory of a particular algebraic system $(ADR, 0, 1, +, -)$; it implies that for any sentence ϕ (in the language $(0, 1, +, -, =)$) the following holds: $T_{ADR} \vdash \phi$ iff $(ADR, 0, 1, +, -) \models \phi$.

The integer data type in MoRe is any (finite or infinite) (sub)set INT of (mathematical) integers $\mathbb{Z}$ with the standard constants *zero* (0) and *one* (1) that is closed under operations called *addition* $+$, *subtraction* $-$, *multiplication* $\times$ (denoted by $*$) and *division* $/$ with an implicit computable surjective type-casting function $in2ad : INT \to ADR$; we assume that $in2ad$ is a homomorphism of $(INT, 0, 1, +, -, =)$ onto $(ADR, 0, 1, +, -, =)$ and (due to this assumption) we can consider multiplication- and division-free integer expressions as address expressions.

Let V be an infinite[3] alphabet of variables[4] (for legal integers and/or addresses), C be a language for representation of integer constants (i.e. integer values as well as addresses via implicit type casting), T be a language of arithmetic expressions (terms) with constants from C and variables from V, and F be a language of the admissible logical formulas constructed with equalities ($=$) and inequalities ($\neq$ at least and, optionally, $<$, $>$, etc.) between expressions from T using Boolean operations.

The syntax of MoRe is defined as follows:

$$\begin{aligned} P ::= \ & skip \mid var\ V = C \mid V := T \mid \\ & V := cons(C^*) \mid [V] := V \mid V := [V] \mid dispose(V) \mid \\ & (P; P) \mid (if\ F\ then\ P\ else\ P) \mid (while\ F\ do\ P). \end{aligned}$$

The structural operational semantics (SOS) of this model language uses a (memory) model consisting of two disjoint parts:

- a static memory (conventionally) called *stack* and
- a dynamic memory (conventionally) called *heap*.

A state is an arbitrary pair (couple) of mappings $s = (s.st,\ s.hp)$ (or, for short, $s = (st, hp)$, or (st, hp) when s is implicit), where:

- st is a state of the stack, i.e. a partial mapping (with a finite domain) from variables V to integers INT (understood as their values), i.e. $st : V \xrightarrow{fin} INT$,
- hp is a state of the heap, i.e. a partial mapping (with a finite domain) from addresses ADR to integers INT (understood as referenced values), i.e. $hp : ADR \xrightarrow{fin} INT$.

The semantics of expressions (terms) T and logical formulas F is defined as follows.

[3]We need an infinite alphabet since we need *fresh* variables in our calculus.

[4]The alphabet may overlap or be disjoint with the alphabet of pointer variables in Subsection 1.2.

Expressions: Since the expressions T are constructed from the constants C and variables V, every expression $t \in T$ in every stack state st : $V \xrightarrow{fin} INT$ has a definite or an indefinite value $st(t) \in INT \cup \{\omega\}$; the exceptional indefinite value may result from division by 0, the use of an undeclared variable, or the use of a variable with an indefinite value.

Formulas: Since the logical formulas F are constructed using the Boolean connectives from equalities and inequalities of arithmetic expressions, every formula $\phi \in F$ in any stack state st: $V \xrightarrow{fin} INT$ can be either true (valid) $st \models \phi$, or false (invalid) $s \not\models \phi$, or indeterminate $st \models ?\phi$ according to the following rules:

- if both expressions of an equality/inequality have the definite values in st, the truth value of this equality/inequality corresponds to the values of the expressions;
- if one or both expressions of an equality/inequality have the indefinite values in st, the value of this equality/inequality in st is indeterminate;
- if all subformulas of a Boolean formula are true or/and false in st, then the truth value of the formula is defined in the standard Boolean manner;
- if a subformula of a Boolean formula is indeterminate in st, then the formula is also indeterminate.

So, to define the expression and formula values, we use a Pascal-like approach rather than C-like: an expression or a formula has a definite value if all its subexpressions/subformulas do.

SOS is an inference system for deduction of triples of the form

$$s\langle\alpha\rangle s',$$

where s is a state, s' is a state or an exception *abort* (an exceptional state or situation), and α is a program; the intuition behind this triple is as follows: the program α converts the input state s into the output "state" s' (that may be an exception). The inference rules are syntax-driven and have the following form:

$$\frac{s_1\langle\alpha_1\rangle s'_1 \ \dots \ s_n\langle\alpha_n\rangle s'_n}{s\langle\alpha\rangle s'} \quad condition,$$

where $n \geq 0$ is the number of premises of the rule, and *condition* is an applicability condition; the inference rules without premises (i.e. when $n = 0$) are axioms. Below is a list of axioms and inference rules with comments.

Variable declaration axioms. If a variable has not been declared yet, it can be declared and initialized by a constant value, but an attempt to re-declare the variable results in an exception:

- $\frac{}{(st,hp)\langle var\ \ x{=}c\rangle(st\cap(x\mapsto c),\ hp)}$ if $x \notin dom(st)$;
- $\frac{}{(st,hp)\langle var x{=}c\rangle abort}$ otherwise.

Here $(x \mapsto c) : V \to INT$ is a singleton function with the graph (x, c).

Empty operator axiom: $\frac{}{s\langle skip\rangle s}$.

Direct assignment axioms. If a variable has been declared and a term has a definite value, the assignment updates the value of the variable by the value of the term; otherwise the assignment results in an exception:

- $\frac{}{(st,hp)\langle x{:=}t\rangle(upd(st,x,st(t)),\ hp)}$ if $x \in dom(st)$ and $st(t) \in INT$;
- $\frac{}{(st,hp)\langle x{:=}t\rangle abort}$ otherwise.

Here $upd(st, x, st(t)) : V \to INT$ is an updated function st, such that for every variable $y \in V$

$$upd(st, x, st(t))(y) = \begin{cases} st(t), \text{ if } y \equiv x, \\ st(y), \text{ if } y \not\equiv x. \end{cases}$$

Memory allocation axioms. The command *cons* allocates (if possible) a fresh heap "segment", initializes the cells within the segment by constant values, and saves the first address of the segment in a specified declared variable; otherwise the allocation results in an exception:

- $\frac{}{(st,hp)\langle x{:=}cons(c_0,\ldots c_k)\rangle(upd(st,x,l),\ hp\cup(in2ad(l)\mapsto c_0)\cup\ldots(in2ad(l+k)\mapsto c_k))}$
  if $x \in dom(st)$,
  addresses $in2ad(l), \ldots in2ad(l + k)$ are disjoint,
  and $\{in2ad(l), \ldots in2ad(l + k)\} \cap dom?(hp) = \varnothing$;
- $\frac{}{(st,hp)\langle x{:=}cons(c_0,\ldots c_k)\rangle abort}$ otherwise.

Indirect assignment axioms. If the variables x and y have been declared, the cell pointed by x has been allocated, the indirect assignment updates the value of this cell in the heap by the value of y; otherwise the attempt of the indirect assignment results in an exception:

- $\frac{}{(st,hp)\langle [x]{:=}y\rangle(st,\ upd(hp,\ in2ad(st(x)),\ st(y)))}$ if $x, y \in dom(st)$ and $in2ad(st(x)) \in dom(hp)$;
- $\frac{}{(st,hp)\langle [x]{:=}y\rangle abort}$ otherwise.

Dereferencing axioms. If the variables x and y have been declared and the cell pointed by y has been allocated, then the dereferencing updates the value of the variable x; otherwise the attempt results in an exception:

- $\frac{}{(st,hp)\langle x{:=}[y]\rangle(upd(st,\ x,\ hp(st(y))),\ hp)}$ if $x, y \in dom(st)$ and $in2ad(st(y)) \in dom(hp)$;

- $\overline{(st,hp)\langle x{:=}[y]\rangle abort}$ otherwise.

Memory deallocation axioms. If a variable has been declared and the cell pointed by the variable has been allocated, then this cell is deallocated; otherwise the attempt results in an exception:

- $\overline{(st,hp)\langle dispose(x)\rangle(st,\ hp\upharpoonright(dom(hp)\setminus in2ad(st(x))))}$ if $x \in dom(st)$ and $in2ad(st(x)) \in dom(hp)$;
- $\overline{(st,hp)\langle dispose(x)\rangle abort}$ otherwise.

Here $hp \upharpoonright (dom(hp) \setminus in2ad(st(x)))$ is the restriction of the function $hp : ADR \to INT$ onto the domain $dom(hp) \setminus in2ad(st(x))$.

Sequential composition inference rules. If the first subprogram aborts, then the composition aborts; otherwise the second subprogram should be applied to the result of the first one:

$$\frac{s\langle\alpha\rangle abort}{s\langle\alpha;\beta\rangle abort} \qquad \frac{s\langle\alpha\rangle s' \quad s'\langle\beta\rangle s''}{s\langle\alpha;\beta\rangle s''} .$$

Choice axiom and inference rules. If the choice condition is true, then select then-branch; if the condition is false, then select else-branch; otherwise the choice results in an exception:

$$\frac{s\langle\alpha\rangle s'}{s\langle if\ \phi\ then\ \alpha\ else\ \beta\rangle s'} \text{ if } s.st \models \phi \qquad \frac{s\langle\beta\rangle s'}{s\langle if\ \phi\ then\ \alpha\ else\ \beta\rangle s'} \text{ if } s.st \not\models \phi$$

$$\overline{s\langle if\ \phi\ then\ \alpha\ else\ \beta\rangle abort} \text{ if } s.st \models ?\phi.$$

Loop axioms and rule. If the loop condition is true, one iteration is executed and the loop should be attempted again; if the condition is false, the loop halts; if the condition is indeterminate, the loop results in an exception:

$$\frac{s\langle\alpha\rangle s' \quad s'\langle while\ \phi\ do\ \alpha\rangle s''}{s\langle while\ \phi\ do\ \alpha\rangle s''} \text{ if } s.st \models \phi \qquad \overline{s\langle while\ \phi\ do\ \alpha\rangle s} \text{ if } s.st \not\models \phi$$

$$\overline{s\langle while\ \phi\ do\ \alpha\rangle abort} \text{ if } s.st \models ?\phi.$$

3. The alias calculus for MoRe

Let us fix a MoRe-program for simplicity of notation. All variables, expressions and programs within this section are variables, expressions and sub-programs of this fixed program.

3.1. Preliminaries

The sets of address variables AV and address expressions AE (of the program) are defined by joint induction as follows:

- an address variable is any variable x that occurs (within the program) in
  - the left-hand side of any memory allocation $x := cons(\dots)$,
  - the left-hand side of any indirect assignment $[x] := \dots$,
  - the right-hand side of any dereferencing $\dots := [x]$,
  - any memory deallocation operator $dispose(x)$,
  - any address expression;
- address expressions (within the program) are
  - all address variables,
  - all subexpressions of any address expression,
  - all expressions t, constructed from C and V using addition and subtraction which occur in the right-hand side of any assignment to any address variable $x := t$,
  - all expressions $x + 1, \dots\ x + k$ such that the program has the memory allocation $x := cons(c_0, \dots, c_k)$.

For any set of address expressions AS and any address variable x in AV, let

- $AS \sim x$ be the set of the expressions in AS that use x (i.e. have instances of x),
- $AS \backslash x$ be the set of the expressions in AS that do not use x (i.e. have no instances of x).

It is obvious that $AS = (AS\backslash x) \cup (AS \sim x)$.

For any set of address expressions AS and any set of address variables $D \subseteq AV$, let $AS(D)$ be the set of all address expressions in AS that do not use variables other than in D (i.e. $AS(D) = \bigcap_{x \in D} AS\backslash x$).

A pair of *aliases* (*synonyms*) is an equality of two address expressions. A pair of *anti-aliases* (*antonyms*) is a difference (i.e. the inequality $\neq$) of two address expressions.

Recall that all address expressions in AE are linear expressions with integer coefficients. Hence the pairs of synonyms or antonyms over AE look like Diophantine equations and differences over integers. Nevertheless we consider all these pairs as equations and differences over $(ADR, 0, 1, +, -)$ assuming implicit type casting.

A *configuration* is a triple $Cnf = (I, A, S)$ consisting of

- a set $I \subseteq AV$ of address variables,
- a set of address expressions $A \subseteq AE(I)$,
- a *consistent* set S of pairs of synonyms and antonyms (with variables in I).

Here consistency means that S has a solution as a system of equalities and differences in $(ADR, 0, 1, +, -)$. Informally speaking, the set I represents the initialized address variables, the set A — address expressions that point onto the allocated memory, and the set S is a system of equations and differences specifying which expressions can be aliases and which cannot.

For any configuration $Cnf = (I, N, S)$, let

- $\&Cnf$ be the conjunction of all pairs of synonyms and antonyms in S;
- $cls(Cnf) = \{e' = e'' \ : \ e', e'' \in AE(I),\ T_{ADR} \vdash \&Cnf \to (e' = e'')\} \cup$
  $$\cup\ \{e' \neq e'' \ : \ e', e'' \in AE(I),\ T_{ADR} \vdash \&Cnf \to (e' \neq e'')\};$$
- $ncl(Cnf) = cls(Cnf) \cup \{e' \neq e'' \ : \ e', e'' \in AE(I),\ (e' = e'' \notin cls(cnf))\}$.

(Here *cls* stays for *closure* and *ncl* stays for *negative closure*.)

Let $Cnf = (I, N, S)$ be a configuration and $s = (st, hp)$ be a state; we write $s \models Cnf$ and say that s satisfies the configuration Cnf, when

- I is the set of all address variables that are declared in st (i.e. $I = dom(st)$);
- $st(A) = \{st(e) \ : \ e \in A\}$ is the set of the allocated heap elements in hp (i.e. $st(A) = dom(hp)$);
- all synonyms and antonyms in $ncl(Cnf)$ are true (valid) in s, i.e.:
  - $in2ad(st(e')) = in2ad(st(e''))$ for every pair of synonyms $e' = e''$ in S,
  - $in2ad(st(e')) \neq in2ad(st(e''))$ for every pair of antonyms $e' \neq e''$ in S.

For any configurations $Cnf' = (I', A', S')$ and $Cnf'' = (I'', A'', S'')$, let us say that they are equivalent if $I' = I''$, for every $e' \in A'$ there exists $e'' \in A''$ such that $T_{ADR} \vdash \&Cnf' \to (e' = e'')$ (and vice versa), and $ncl(Cnf') = ncl(Cnf'')$.

A *distribution* (or alias distribution) is an arbitrary finite set of configurations in which every two configurations are not equivalent. If D is an arbitrary set of configurations (a distribution in particular), then its refinement is a distribution $rfn(D)$ obtained from D by leaving a single configuration in each equivalence class in D.

Let D be an arbitrary alias distribution and s be an arbitrary state; we write $s \models D$ and say that s satisfies the distribution D, when $s \models Cnf$ for some configuration $Cnf \in D$.

3.2. The calculus

Let D be any alias distribution. We define the distribution converter

$$\lambda\alpha : \text{MoRe. } \lambda D : \text{distribution. } aft(D, \alpha)$$

by induction on program structure: the induction base defines the converter for individual operators; the induction step defines the converter for compound programs. The definition in its nature is executable (i.e. the definition is an algorithm) and an execution of this algorithm may cast (i.e. make) some warnings in run-time.

3.2.1. Individual operators

For any MoRe (syntax) expressions[5] e, e' and e'', let $e_{e'/e''}$ be the result of substitution of e' instead of all instances of e'' into e.

For operators that do not change the address variables, we have:

- $aft(D, skip) = D$;
- $aft(D, var x = i) = D$, if x is not an address variable;
- $aft(D, x := t) = D$, if x is not an address variable;
- $aft(D, x := [y]) = D$, if x is not an address variable;
- $aft(D, [x] := y) = D$, if t is not an address expression.

If x is any address variable, the distribution $aft(D, var x = i)$ is obtained as follows. Let $Cnf = (I, A, S)$ be an arbitrary configuration in D. If $x \in I$ then the algorithm makes *re-initialization* warning. Let $Cnf_{var\ x=c} = (I_{var\ x=c}, A_{var\ x=c}, S_{var\ x=c})$, where

- $I_{var\ x=c} = I \cup \{x\}$,
- $A_{var\ x=c} = \{e' \in AE(I_{var\ x=c})$: $T_{ADR} \vdash \&Cnf \to (e'_{c/x} = e'')$ for some $e'' \in A\}$,
- $S_{var\ x=c)} = ncl($ $\{e' = e''$: $e', e'' \in AE(I_{var\ x=c})$ and $T_{ADR} \vdash \&Cnf \to (e'_{c/x} = e''_{c/x})\} \cup$ $\{e' \neq e''$: $e', e'' \in AE(I_{var\ x=c})$ and $T_{ADR} \vdash \&Cnf \to (e'_{c/x} \neq e''_{c/x})\})$.

[5] i.e. variables, terms, formulas, programs, etc.

Then let $aft(D,\ var\ x = c)$ be $rfn\{Cnf_{var\ x=c} : Cnf \in D\}$.

If x is any address variable, the distribution $aft(D, x := t)$ is obtained as follows. Let $Cnf = (I, N, S)$ be an arbitrary configuration in D. If $x \notin I$ or t has an uninitialized variable (i.e. not in I), the algorithm gives an *un-initialization* warning. Let $Cnf_{x:=t} = (I_{x:=t}, A_{x:=t}, S_{x:=t})$, where

- $I_{x:=t} = I$,
- $A_{x:=t} = \{e' \in AE(I_{x:=t})\ :$ $T_{ADR} \vdash \&Cnf \to (e'_{t/x} = e'')$ for some $e'' \in A\}$,
- $S_{x:=t} = ncl($
  $\{e' = e''\ : e', e'' \in AE(I_{x:=t})$ and $T_{ADR} \vdash \&Cnf \to (e'_{t/x} = e''_{t/x})\} \cup$
  $\{e' \neq e''\ : e', e'' \in AE(I_{x:=t})$ and $T_{ADR} \vdash \&Cnf \to (e'_{t/x} \neq e''_{t/x})\})$.

If there exists $e'' \in A$ such that $T_{ADR} \not\vdash \&Cnf \to (e'_{t/x} = e'')$ for every $e' \in A_{x:=t}$, then the algorithm gives a *memory-leak* warning. Then let $aft(D,\ x := t)$ be $rfn\{Cnf_{x:=t} : Cnf \in D\}$.

The distribution $aft(D,\ x := cons(c_0, \ldots c_k))$ is obtained as follows. Let $Cnf = (I, A, S)$ be an arbitrary configuration in D. If $x \notin I$, the algorithm gives an *un-initialization* warning. Let y be a new (fresh) variable and let $\&Cnf_{New(y,k)}$ be the conjunction of $\&Cnf$ with all pairs of antonyms that have the form $e \neq y+i$ or $y+i \neq y+j$, where $e \in AE(I)$ and $0 \leq i < j \leq k$. Let $Cnf_{x:=cons_k}$ be $(I_{x:=cons_k},\ A_{x:=cons_k},\ S_{x:=cons_k})$, where

- $I_{x:=cons_k} = I$,
- $A_{x:=cons_k} = A \cup \{x, (x+1), \ldots (x+k)\}$,
- $S_{x:=cons_k} = ncl($
  $\{e' = e'' : e', e'' \in AE(I_{x:=cons_k}),\ T_{ADR} \vdash \&Cnf_{New(y,k)} \to e'_{y/x} = e''_{y/x}\} \cup$
  $\{e' \neq e'' : e', e'' \in AE(I_{x:=cons_k}),\ T_{ADR} \vdash \&Cnf_{New(y,k)} \to e'_{y/x} \neq e''_{y/x}\})$.

If there exists $e'' \in A$ such that $T_{ADR} \not\vdash \&Cnf \to (e'_{t/x} = e'')$ for every $e' \in A_{x:=cons(c_0,\ldots c_k)}$, then the algorithm makes *memory-leak* warning. Then let $aft(D,\ x := cons(c_0, \ldots c_k))$ be $rfn\{Cnf_{x:=cons_k}\ :\ Cnf \in D\}$.

If x is any address variable, the distribution $aft(D, x := [y])$ is obtained as follows. Let $Cnf = (I, A, S)$ be an arbitrary configuration in D. If $x \notin I$ or $y \notin I$, then the algorithm makes *un-initialization* warning. If $T_{ADR} \not\vdash \&Cnf \to (y = e)$ for every $e \in A$, then the algorithm makes *un-allocation* warning. Let $Cnf_{x:=[y]}$ be the set of all configurations $(I_{x:=[y]}, A_c, S_c)$, where c be an integer constant, and

- $I_{x:=[y]} = I$,

- $A_c = \{e' \in AE(I_{x:=[y]}) \ : \ T_{ADR} \vdash \&Cnf \to (e'_(c/x) = e'')$ for some $e'' \in A\}$,
- $S_c = ncl($
  $\{e' = e'' : e', e'' \in AE(I_{x:=[y]}),\ T_{ADR} \vdash \&Cnf \to (e'_{c/x} = e''_{c/x})\} \cup$
  $\{e' \neq e'' : e', e'' \in AE(I_{x:=[y]}),\ T_{ADR} \vdash \&Cnf \to (e'_{c/x} \neq e''_{c/x})\})$.

If there exists an integer constant c and an expression $e'' \in A$ such that $T_{ADR} \not\vdash \&Cnf \to (e'_{c/x} = e'')$ for every $e' \in A_{x:=[y]}$, the algorithm gives a *memory-leak* warning. Then let $aft(D, x := [y])$ be $rfn(\bigcup_{Cnf \in D} Cnf_{x:=[y]})$.

The distribution $aft(D,\ dispose(x))$ is obtained as follows. Let $Cnf = (I, A, S)$ be an arbitrary configuration in D. If $x \notin I$, the algorithm gives an *un-initialization* warning. If $T_{ADR} \not\vdash \&Cnf \to (x = e)$ for every $e \in A$, the algorithm gives a *un-allocation* warning. Let $Cnf_{dispose(x)} = (I_{dispose(x)}, A_{dispose(x)}, S_{dispose(x)})$, where

- $I_{dispose(x)} = I$,
- $A_{dispose(x)} = A \setminus \{e \in AE(I_{dispose(x)}) \ : \ T_{ADR} \vdash \&Cnf \to (e = x)\}$,
- $S_{dispose(x)} = S$.

Then let $aft(D,\ dispose(x))$ be $rfn\{Cnf_{dispose(x)} \ : \ Cnf \in D\}$.

3.2.2. Compound programs

- $aft(D,\ (\alpha; \beta)) = aft(aft(D, \alpha),\ \beta)$;
- $aft(D,\ if\ \phi\ then\ \alpha\ else\ \beta) = rfn(aft(D, \alpha) \cup aft(D, \beta))$;
- $aft(D,\ while\ \phi\ do\ \alpha) = rfn(\bigcup_{i \geq 0} aft(D,\ \alpha^i))$,
  where $\alpha^0 \equiv skip$, and $\alpha^{i+1} \equiv (\alpha^i; \alpha)$ for any $i \geq 0$.

4. Conclusion

4.1. Results

The presented here version of the alias calculus for the programming language MoRe is *safe* (by construction) in the following sense.

Proposition 1. *Let D be any alias distribution, α be any MoRe-program and s be any state such that $s \models D$;*

- *if s' is a state such that $s\langle\alpha\rangle s'$ then $s' \models aft(D, \alpha)$;*
- *if α started in s aborts due to*
  - *re-initialization of some address variable, then the re-initialization warning will be casted in $aft(D, \alpha)$;*

  - *the use of an un-initialized address variable, then the un-initialization warning will be cast in $aft(D, \alpha)$;*
- *if α started in s has*
  - *a memory leak, then the memory-leak warning will be cast in $aft(D, \alpha)$;*
  - *an invalid memory access, then the un-allocation warning will be cast in $aft(D, \alpha)$.*

Thus the presented variant of the alias calculus answers the question formulated in [10] for a preliminary variant of the calculus: How do run-time memory leaks relate to memory-leak warnings? How do run-time invalid accesses relate to invalid-access warnings?

In particular, the examples of exceptions mentioned in the Introduction can be represented in MoRe as follows:

- $\alpha \equiv\ x := cons(1); x := cons(2)$,
- $\beta \equiv\ y := x; dispose(x); dispose(y)$.

If we execute $aft((\{x, y\}, \varnothing, \varnothing),\ \alpha)$, then the memory-leak warning will be cast; if we compute $aft((\{x, y\}, \varnothing, \varnothing),\ \beta)$, then un-allocation warning will be cast. In both cases we start with the configuration $Cnf = (I, A, S) = (\{x, y\}, \varnothing, \varnothing)$, where the variables x and y are declared and initialized (i.e. I is $\{x, y\}$), but no dynamic memory is allocated (i.e. A is the first $\varnothing$) and there are no alias relations between x and y (i.e. S is the second $\varnothing$).

4.2. Further research topics

In Section 2, we assume that the address data type in MoRe is any (finite or infinite) set of values ADR such that $(ADR, 0, 1, +, -, =)$ is a commutative additive semi-group with decidable first-order theory T_{ADR}. Examples include the ring of residuals modulo any fixed positive integer, versions of Presburger arithmetic, etc. The decidability condition for T_{ADR} has been used in Section 3 to solve the properties of the form

$$T_{ADR} \vdash \&S \to e' = e'' \text{ and } T_{ADR} \vdash \&S \to e' \neq e'',$$

where S is a system of linear Diophantine equations and differences ($\neq$, i.e. negation of equality). It implies that we do not need the "full" decidability of T_{ADR} but just an efficient algorithm to check consistency for systems of this type. Unfortunately, we could not find any efficient algorithm to solve the consistency of systems with differences but algorithms to solve systems with inequalities $<$ and $>$. We plan to extend the algorithms from [5, 6] to handle the differences as well[6].

[6]Now we know how to do this but in an inefficient manner.

Let us recall that the primary purpose of this paper was to present an alias calculus for a programming language which is more realistic than the original one described in [7]. The presented calculus is insensitive to the control flow and it uses only stack (automatic) variables for analysis and a very rough method to define the address variables and expressions. So, the most evident topics for further research are the design and development of a calculus sensitive to the control flow that takes into account some information about the heap. Also, we have to add other data types different from INT and their addresses ADR. Only after that it will be reasonable to add to the research agenda the prototyping of a calculus-based alias analysis tool. Nevertheless, it makes sense to examine the relations between the alias calculus and the so-called Andersen-styled [1] and the Steensgaard-styled [11] analyses in the near future.

References

[1] Andersen L.O. Program Analysis and Specialization for the C Programming Language. – Ph.D. Thes. / DIKU, University of Copenhagen, Denmark, 1994.

[2] Haberland R., Ivanovskiy S. Dynamically allocated memory verification in object-oriented programs using Prolog // Prelim. Proc. of the 8th Spring-Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2014), May 29-31, 2014, Saint Petersburg, Russia. – Institute for System Programming of the Russian Academy of Sciences (ISPRAS), 2014. – P. 46–50.

[3] Hind M. Pointer analysis: Haven't we solved this problem yet? // Proc. of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01). – P. 54–61.

[4] Hoare C.A.R. The verifying compiler: A grand challenge for computing research // Perspectives of Systems Informatics (PSI'2003). – Lect. Notes Comput. Sc. – Springer, 2003. – Vol. 2890. – P. 1–12.

[5] Kryvyi S.L. Algorithms for solving systems of linear diophantine equations in integer domains // Cybernetics and Systems Analysis. – 2006. – Vol. 42(2). – P. 163–175.

[6] Kryvyi S. L. Algorithms for solving systems of linear Diophantine equations in residue // Cybernetics and Systems Analysis. – 2007. – Vol. 43(6). – P. 787–798.

[7] Meyer B. Steps towards a theory and calculus of aliasing // Internat. J. of Software and Informatics. – 2011. – Special Issue (Festschrift in honor of Manfred Broy). – P. 77–115.

[8] Reynolds J.C. Separation logic: A logic for shared mutable data structures // Proc. of 17th IEEE Symp. on Logic in Computer Science (LICS 2002). – IEEE Computer Press, 2002. – P. 55–74.

[9] Sridharan M., Chandra S., Dolby J., Fink S.J., Yahav E. Alias analysis for object-oriented programs // Aliasing in Object-Oriented Programming: Types, Analysis, and Verification / Eds. D. Clarke, T. Wrigstad, J. Noble. – Lect. Notes Comput. Sci. – Springer, 2013. – Vol. 7850. – P. 196–232.

[10] Shilov N.V., Vorontsov A.P., Satykbayeva A. Alias calculus for a simple imperative language with decidable pointer arithmetic // Tools and Methods of Program Analysis. Proc. of Internat. Science and Practical Conf. Kostroma (14–15 November, 2014). – Kostroma State Technological University, 2014. – P. 29–35.

[11] Steensgaard B. Points-to analysis in almost linear time // POPL'96: Proc. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. – ACM, 1996. – P. 32–41.