Alias Calculus for a Simple Imperative Language with Decidable Pointer Arithmetic

Aizhan Satekbayeva
L.N. Gumilyov Eurasian National University
Astana, Kazakhstan
satekbayeva@gmail.com

Nikolay V. Shilov
Nazarbayev University
Astana, Kazakhstan
nikolay.shilov@nu.edu.kz

Aleksandr P. Vorontsov
Novosibirsk State University
Novosibirsk, Russia
corvin7@gmail.com

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

Keywords*— aliasing problem; alias calculus; logic of partial correctness

I. INTRODUCTION

A. Aliasing Problem

In this position paper we present a variant of *alias calculus*, i.e. a syntax-driven procedure to compute 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)}* to be valid (in Hoare logic of partial correctness).

In general *aliasing problem* is to predict, detect and/or trace pointers to the same addresses in dynamic memory. Importance of the problem is due to mistakes and errors that may happen in program run-time due to improper alias handling. Two simple examples of errors of this type follows below:

-
  ```
  x = malloc(sizeof(int));
  x = malloc(sizeof(int));
  //memory leak;
  ```

-
  ```
  y = x; free x;
  free y;
  // invalid memory access.
  ```

The first example shows a 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 abnormal program termination immediately).

Although errors given in the examples seem obvious and easy to fix, similar problems often happen in real programs with thousands of lines, with complicated modular structure. Therefore, the development of methods to detect and eliminate of similar errors is an important problem from industrial point of view as well as from educational and research perspective (e.g. for *verifying compiler* research [3]).

The purpose of aliasing analysis is to determine statically address expressions in a program which can/may point to the same memory location in run-time. This analysis is intended to find and eliminate the errors in the program that are due to single (as memory leak) or multiple links to pieces of memory (as invalid memory access). In general settings the problem is undecidable for a programming language with expressive pointer (address) arithmetic; however a large collection of approximate algorithms have been published that provide a trade-off between the efficiency, accuracy and soundness of the aliasing analysis [6].

There are several attributes to characterize alias analyses [2], some of them are listed and explained below:

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

While a flow-sensitive analysis usually computes aliases for every control point in a program, flow-insensitive analyses computes aliasing for the program as a whole. Context-sensitivity is about function/procedure calls and means whether a context of a call is taken into consideration or not. Analysis may be founded on different models of the heap (i.e.

This research is supported by Nazarbayev University Seed Grant KF-14/16 “Research of Formal Models for analysis of programs with Dynamic Memory”.

dynamic memory): heap may be a data structure consisting of cells with abstract addresses capable to save arbitrary data, or with integer addresses to store primitive data only, etc. Aliasing may be presented by equalities, by sets of synonyms, or somehow else.

In spite of decades of research, development and use there are still challenges in alias analysis [2, 6]:

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

New research on alias analysis emerge (e.g. [1]) due to these and other reasons. In particular, alias calculus proposed by Bertrand Meyer [4] is new approach to aliasing research. Three variants of calculus for toy imperative language with single data type for abstract pointers are presented in [4]; these variants are set-based formalisms insensitive to control flow and context, and without address arithmetic.

B. Paper and its Structure

The primary purpose of our position paper is to present an alias calculus for more realistic programming language with static and dynamic memory, a language with data values and decidable address arithmetic. The presented variant is insensitive to control flow (as the original calculus by B. Meyer), but is equation-based (in contrast to the original calculus).

The rest of the paper is organized as follows. The next subsection sketches alias calculus for a toy programming language E0 that was developed by B. Meyer in [4]. Then in section II we introduced programming language MoRe, its formal syntax and structural operational semantics; this language is more realistic than E0 and may be considered as a dialect of programming language used in [5] for separation logic. Stack-based alias calculus for this language is presented in section III. A preliminary discussion of perspectives of the calculus for detecting memory leaks and invalid memory access is presented in the last section IV; some topics for further research are also discussed in the concluding section.

C. Alias Calculus for Programming Language E0

Let V be an arbitrary finite fixed set which elements are called (pointer or address) 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 about which of these variables may point to the same storage (memory) location. For any binary relation S on V let *sic(S)* be symmetric irreflexive closer[1] of S i.e.

- $sic(S)=\{(x,y), (y,x) : (x,y)\in S\}$.

For any alias relation S and any variable x let[2]

- $(S\text{-}x)=\{(y,z)\in S :$ neither $y\equiv x$ nor $z\equiv x\}$,
- $(S/x)=\{y : y\equiv x$ or $(x,y)\in S\}$.

For any alias relation S let *cnd(S)* be the following first-order quantifier-free formula[3]

- $\vee_{x,y\in V,(x,y)\notin S}(x\neq y)$;

it is easy to see that *cnd*-constructor possesses is monotone: for any alias relations S_1 and S_2, if $S_1\subseteq S_2$ then $cnd(S1)\rightarrow cnd(S2)$.

In [4] the alias calculi were defined for two programming languages E0 and E1: E1 is a superset of E0 with procedures. Both languages have single data type for addresses only. Syntax of the language E0 is defined as follows:

P::= *skip* | *forget(V)* | *create(V)* | *V:=V* |
(P;P) | P^N | *then P else P* | *loop P*

where

- V is metavariable for the set of address variables (that was fixed above),
- N is metavariable for natural numbers in any fixed notation (e.g. *N*::= *0* | *1* | *2* | …)

As we already stated in the above, an 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* denotes the transformer of alias relations[4]. In terms of Hoare's logic it is possible to say that the calculus guarantees the correctness for the following triple *{cnd(S)}P{cnd(aft(S,P))}*.

Alias calculus for E0 and its informal operational semantics follow below.

- $aft(S, skip) = S$ because *skip* is the empty operator.
- $aft(S, forget(x)) = aft(S, create(x)) = S\text{-}x$, i.e. memory deallocation and allocation operators have the same effect on an alias relation because after these operations the variable x isn't alias to any other variable.
- $aft(S, x:=y) = sic((S\text{-}x)\cup (\{x\}\times((S\text{-}x)/y)))$, i.e. in a 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 sequential application of programs.
- $Aft(S, \alpha^0) = S$ and $aft(S, \alpha^{n+1}) = aft(aft(S, \alpha^n), \alpha)$ for any $n\geq 0$, i.e. n-fold iteration (repetition) α^n is the n-fold sequential composition.
- $Aft(S, then\ \alpha\ else\ \beta) = aft(S, \alpha) \cup aft(S, \beta)$, i.e. then-else is nondeterministic choice of any branch in two.
- $aft(S, loop\ \alpha) = \cup_{n\geq 0}\ aft(S, \alpha^n)$, i.e. loop is nondeterministic iteration.

[1] Acronym *sic* stays for Symmetric Irreflexive Closure.
[2] Hereafter we use symbol $\equiv$ to denote syntactic identity.
[3] Acronym *cnd* stays for CoNDition.
[4] Acronym *aft* stays for AFTer.

II. MoRe Programming Language

In this section we present a programming language MoRe that is a dialect of the programming language used for definition of Separation Logic in [5]; the acronym MoRe stays for *More Realistic*.

The language has two data types that are called *addresses* and *integers* with implicit type casting from integers to addresses.

Address data type in MoRe is any (finite or infinite) set of values *ADR* with constants that are conventionally denoted *0* and *1*, operations that are conventionally called *addition* and *subtraction* (denoted + and –) such that *(ADR, 0, 1, +, –)* is a commutative additive semi-group with decidable first-order theory $\boldsymbol{T_{ADR}}$. Examples include $\boldsymbol{Z}_m$ the ring of residuals modulo any particular fixed positive *m*, Presburger arithmetic, etc. Let us remark that $\boldsymbol{T_{ADR}}$ is a complete theory of a particular algebraic system *(ADR, 0, 1, +, –)*; it implies that for any sentence φ the following holds: $(ADR, 0, 1, +, -)\models\varphi$ iff $\boldsymbol{T_{ADR}}\vdash\varphi$.

Integer data type in MoRe is any (finite or infinite) set of (mathematical) integers *INT* with "standard" constants *0* and *1*, "standard" operations addition, subtraction, multiplication and division within the range of *INT* (denoted +, –, * and /) and with *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 (subject to the type-casting).

Let *V* be an alphabet variables (for legal integers and/or addresses), *C* be a language for representation of integer constant (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* is language of the admissible logical formulas constructed of equalities (=) and inequalities (≠) between expressions from *T* using of Boolean operations. Syntax of MoRe programming language is defined as follows:

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

Structural operational semantics of this model language uses memory model consisting of two disjoint parts: a static memory (conventionally) called *stack* and dynamic memory (conventionally) called *heap*. State is an arbitrary pair 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 finite domain) from variables *V* to integers *INT* (understood as their values), i.e. $st:V\to INT$,
- *hp* is a state of the heap, i.e. a partial mapping with finite domain from addresses *ADR* to integers *INT* (understood as referenced values), i.e. $hp:ADR\to INT$.

The semantics of expressions (terms) *T* and logical formulas *F* is defined as follows. Since expressions *T* are constructed from constants *C* and variables *V*, every expression $\tau\in T$ in every stack state $st:V\to INT$ has a definite or indefinite value $st(\tau)\in INT\cup\{\omega\}$. Since logical formulas *F* are constructed (using Boolean connectives) of equalities and inequalities of arithmetic expressions, every formula $\varphi\in F$ in any stack state $st:V\to INT$ can be either true (valid) $st\models\varphi$, false (invalid) $st\sim\varphi$, or indeterminate $st?\varphi$ according to the following rules:

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

Structural operational semantics (SOS) of programming language MoRe is axiomatic system for 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; intuition behind this triple follows: program α converts input state s' into output state s'' (that may be exception). SOS inference rules are syntax-driven and have the following form:

$$\frac{s_1\langle\alpha_1\rangle s_1',\ \dots\ s_n\langle\alpha_n\rangle s_n'}{s\langle\alpha\rangle s'}\qquad \text{(application condition)}$$

where *n* is the number of premises of the rule, and *condition* is an applicability condition; inference rules without premises (i.e. when *n*=0) are axioms. Commented list of axioms and inference rules follows below.

Variable declaration axioms. If a variable *x* hasn't been declared yet, it can be declared and initialized by a value *i*, but an attempt to re-declare the variable results in exception:

$$\frac{}{(st,hp)\langle var\ x=i\rangle(st\cup\{(x,i)\},\ hp)}\qquad \text{if } x\notin Dom(st));$$

$$\frac{}{(st,hp)\langle var\ x=i\rangle abort}\qquad \text{otherwise.}$$

Empty operator axiom:

$$\frac{}{s<skip>s}\quad .$$

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

$$\frac{}{(st,hp)<x:=t>(upd(st,\ x,\ t),\ hp)}\qquad \text{if } x\in Dom(st)) \text{ and } st(t)\in INT;$$

$$\frac{}{(st,hp)<x:=t>abort}\qquad \text{otherwise.}$$

Memory allocation axioms. The command *cons* allocates (if possible) a fresh segment of *(k+1)* heap elements, and initializes cells in this segment by provided initial values; otherwise the allocation results in exception:

$$\frac{}{(st,hp)<x:=cons(i_0,\dots i_k)>(upd(st,\ x,\ l),hp')}\qquad \text{if } x\in Dom(st),\ l\in INT,$$

$in2ad(l)\notin Dom(hp)$, … $in2ad(l+k)\notin Dom(hp)$ are disjoint addresses, $hp' = hp\cup\{(in2ad(l),\ i_0),\ \dots\ (in2ad(l+k),\ i_k)\}$;

$$\frac{}{(st,hp)<x:= cons(i_0,\dots i_k)>abort}\qquad \text{otherwise.}$$

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

$$\frac{}{(st,hp)<[x]:=y>(st,\ hp')}\qquad \text{if } x,y\in Dom(st),\ in2ad(st(x))\in Dom(hp),$$

$h' = upd(hp,\ in2ad(st(x)),st(t))$;

$$\frac{}{(st,hp)<x:=y>abort}\qquad \text{otherwise.}$$

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

$$\frac{}{(st,hp)<x:=[y]>(st',hp)}\qquad \text{if } x,y\in Dom(st),\ in2ad(y)\in Dom(hp),$$

$st'=upd(st,\ x,\ hp(in2ad(y)))$;

$$\frac{}{(st,hp)<x:=[y]>abort}\qquad \text{otherwise.}$$

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

$$\frac{}{(st,hp)<dispose(x)>(st,\ hp')}\qquad \text{if } x\in Dom(st),\ in2ad(st(x))\in Dom(hp),$$

$hp' = hp\setminus\{(in2ad(st(x)),\ hp(in2ad(st(x))))\}$;

$$\frac{}{(st,hp)<dispose(x)>abort}\qquad \text{otherwise.}$$

Composition rules:

$$\frac{s<\alpha>abort}{s<\alpha;\beta>abort}\quad ;$$

$$\frac{s<\alpha>s'\ ,\ s'<\beta>s''}{s<\alpha;\beta>s''}\qquad \text{if } s' \text{ isn't } abort.$$

Choice rules and axiom. If the condition is true in a state, the choice selects then-branch; if the condition is false, else-branch is selected; if the condition is indeterminate, the choice results in exception:

$$\frac{s<\alpha>s'}{s<if\ \varphi\ then\ \alpha\ else\ \beta>s'}\qquad \text{if } s\models\varphi\,;$$

$$\frac{s<\beta>s'}{s<if\ \varphi\ then\ \alpha\ else\ \beta>s'}\qquad \text{if } s\sim\varphi\,;$$

$$\frac{}{s<if\ \varphi\ then\ \alpha\ else\ \beta>abort}\qquad \text{if } s?\varphi\,.$$

Loop rule and axioms. If the condition is true in a state, then one iteration have to be executed and then the loop have to be attempted again; if the condition is false, the loop halts; if the condition is indeterminate, the loop results in exception:

$$\frac{s<\alpha>s' \,,\; s'<while\ \varphi\ do\ \alpha>s''}{s<while\ \varphi\ do\ \alpha>s''} \qquad \text{if } s \models \varphi\,;$$

$$\frac{}{s<while\ \varphi\ do\ \alpha>s} \qquad \text{if } s\sim\varphi\,;$$

$$\frac{}{s<while\ \varphi\ do\ \alpha>abort} \qquad \text{if } s?\varphi\,.$$

III. STACK-BASED ALIAS CALCULUS FOR MORE

Let us fix a program. The set of address variables *AV* and the set of address expressions *AE* (of the program) are defined by mutual induction as follows.

1) Address variables is any variable *x* that occurs (within the program) in

a) the left-hand side of any allocation *x:=cons…*;

b) the left-hand side of any indirect assignment *[x]:=…*;

c) the right-hand side of any dereferencing *…:= [x]*;

d) any memory deallocation operator *dispose(x)*;

e) any address expression.

2) Address expressions (within the program) are

a) all address variables;

b) all subexpression of any address expression;

c) 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*;

d) all expressions $x+1$, … $x+k$ such that the program has memory allocation $x:=cons\ i_0 \ldots i_k$.

For any set of address expressions *AS* and any set of address variables *D* let *AS(D)* be the set of all address expressions in *AS* that don't use variables other than in *D*. In particular, any set of address variables *D* the set *AE(D)* is the set of all address expressions in the program that don't use variables other than in *D*.

For any expression *e*, any expression *t* and any variable *x* let $e_{t\to x}$ be result of substitution *t* instead of *x* in *e*.

A *pair of aliases* (or *synonyms*) is an equality of any two address expressions. A *pair of anti-aliases* (or *antonyms*) is an inequality (≠) of any two address expressions.

Recall that all address expressions in *AE* are linear expressions with integer coefficients. It implies that pairs of synonyms or antonyms over *AE* look like Diophantine equations and inequalities over integers. But we think all these pairs as equations and inequalities over *(ADR, 0, 1, +, –)* assuming implicit type casting (applied to all used integer constants).

A configuration is a triple *Cnf=(I, N, S)* consisting of two sets $N\subseteq I\subseteq AV$ of address variables and a finite set *S* of pairs of synonyms and antonyms (with variables in I) that has a solution as a system of equalities and inequalities in *(ADR, 0, 1, +, –)*, i.e. that is consistent with theory $\boldsymbol{T_{ADR}}$; informally speaking the set *I* is for Initialized address variables, the set *N* is for Non-allocated initialized address variables, and the set *S* is a System of equations and inequalities to specify what expressions may be aliases and what can't be.

For any configuration *Cnf =(I, N, S)* let[5]

- *&Cnf* be conjunction of all pairs of synonyms and antonyms in *S* (assuming implicit type casting);
- *cls(Cnf) =*

$$= \{e'=e'' : e',e''\in AE(I),\ \boldsymbol{T_{ADR}} \vdash \&Cnf\to(e'=e'')\} \cup$$

$$\cup \{e'\neq e'' : e',e''\in AE(I),\ \boldsymbol{T_{ADR}} \vdash \&Cnf\to(e'\neq e'')\};$$

- *ncl(Cnf) = cls(Cnf)* ∪

$$\cup \{e'\neq e'' : e',e''\in AE(I),\ (e'=e'')\notin cls(Cnf)\}.$$

Let *st* be a state of a stack; we write $st \models Cnf$ and say that *st* satisfies configuration *Cnf*, when all variables in *I* are declared in *st* (i.e. $I\subseteq Dom(st)$) and all formulas from *ncl(Cnf)* are true (valid) in *st* (i.e. $st \models \&ncl(Cnf)$).

Any two configurations $Cnf'=(I',N',S')$ and $Cnf''=(I'',N'',S'')$ are said to be equivalent if $I'=I''$, $N'=N''$ and $ncl(Cnf')= ncl(Cnf'')$. Distribution (or alias distribution) is an arbitrary finite set of configurations in which neither two are equivalent. If *D* is an arbitrary set of configurations, 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, *st* be an arbitrary state of a stack; we write $st \models D$ and say that *st* satisfies distribution *D*, when $st \models Cnf$ for a configuration *Cnf* in *D*.

Let us define the distribution converter *aft* for MoRe programs by structural induction – for individual operators and for compound programs.

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

a. *aft(D, skip)=D*;

b. *aft(D, var x=i)=D*, if *x* isn't address variable;

c. *aft(D, x=t)=D*, if *x* isn't address variable;

d. *aft(D, x=[y])=D*, if *x* isn't address variable;

e. *aft(D, [x]=y)=D*, if *y* isn't address variable.

[5] Acronym *cls* stays for <u>cl</u>o<u>s</u>*ure*, acronym *ncl* – for <u>n</u>*egative* <u>cl</u>*osure*.

2) If x is some address variable, distribution $aft(D, var\ x{=}i)$ is obtained as follows. Let $Cnf=(I,N,S)$ be an arbitrary configuration in D. Make *re-initialization warning* if $x\in I$. Let $Cnf_{var\ x=i}$ be $(I_{var\ x=i}, N_{var\ x=i}, S_{var\ x=i})$ where

a. $I_{var\ x=i}=I\cup\{x\}$,

b. $N_{var\ x=i}=N\cup\{x\}$, and

c. $S_{var\ x=i}= ncl(\{e'=e'' : e',e''\in AE(I_{var\ x=i}),$

$$\boldsymbol{T_{ADR}}\vdash \&Cnf\rightarrow(e'_{i\rightarrow x}=e''_{i\rightarrow x})\}\cup$$

$$\cup\{e'\neq e'' : e',e''\in AE(I_{vae\ x=i}),$$

$$\boldsymbol{T_{ADR}}\vdash \&Cnf\rightarrow(e'_{i\rightarrow x}\neq e''_{i\rightarrow x})\}).$$

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

3) If x is some address variable, distribution $aft(D, x{:=}t)$ is obtained as follows. Let $Cnf=(I,N,S)$ be an arbitrary configuration in D. Make *un-initialization warning* if $x\notin I$ or t has uninitialized variable (i.e. not in I). Make *memory-leak warning* if $x\notin N$ and $\boldsymbol{T_{ADR}}\vdash \&Cnf\rightarrow(e_{t\rightarrow x}\neq x)$ for every address expression $e\in AE(I)$. Let $Cnf_{x:=t}$ be $(I_{x:=t}, N_{x:=t}, S_{x:=t})$ where

a. $I_{x:=t}=I$,

b. $N_{x:=t}=N$, and

c. $S_{x:=t}= ncl(\{e'=e'' : e',e''\in AE(I),$

$$\boldsymbol{T_{ADR}}\vdash \&Cnf\rightarrow(e'_{t\rightarrow x}=e''_{t\rightarrow x})\}\cup$$

$$\cup\{e'\neq e'' : e',e''\in AE(I),$$

$$\boldsymbol{T_{ADR}}\vdash \&Cnf\rightarrow(e'_{t\rightarrow x}\neq e''_{t\rightarrow x})\}).$$

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

4) Distribution $aft(D, x{:=}cons(i_0,\ldots i_k)$ is obtained as follows. Let $Cnf=(I,N,S)$ be an arbitrary configuration in D. Make *un-initialization warning* if $x\notin I$. Let y be a new (fresh) variable and let $New_{Cnf}(y,k)$ be the set of all pairs of antonyms that have the form $e\neq y+i$ and $y+i\neq y+j$ where $e\in AE(I)$ and $0\leq i<j\leq k$. Make *memory-leak warning* if $x\notin N$ and $\boldsymbol{T_{ADR}}\vdash (\&Cnf\ \&\ \&New_{Cnf}(y,k))\rightarrow(e_{y\rightarrow x}\neq x)$ for every address expression $e\in AE(I)$. Let $Cnf_{x:=cons(i0,..ik)}$ be $(I_{x:=cons(i0,..ik)}, N_{x:=cons(i0,..ik)}, S_{x:=cons(i0,..ik)})$ where

a. $I_{x:=cons(i0,..ik)}=I$,

b. $N_{x:=cons(i0,..ik)}=N\backslash\{x\}$, and

c. $S_{x:=cons(i0,..ik)}= ncl(\{e'=e'' : e',e''\in AE(I),$

$$\boldsymbol{T_{ADR}}\vdash (\&Cnf\ \&\ \&New_{Cnf}(y,k))\rightarrow(e'_{y\rightarrow x}=e''_{y\rightarrow x})\}\cup$$

$$\cup\{e'\neq e'' : e',e''\in AE(I),$$

$$\boldsymbol{T_{ADR}}\vdash (\&Cnf\ \&\ \&New_{Cnf}(y,k))\rightarrow(e'_{y\rightarrow x}\neq e''_{y\rightarrow x})\}).$$

Then let $aft(D, x{:=}cons(i_0,\ldots i_k))$ be $rfn\{Cnf_{x:=cons(i0,..ik)} : Cnf\in D\}$.

5) If x is some address variable, distribution $aft(D, x{:=}[y])$ is obtained as follows. Let $Cnf=(I,N,S)$ be an arbitrary configuration in D. Make *un-initialization warning* if $x\notin I$ or $y\notin I$. Make *invalid-access warning* if $\boldsymbol{T_{ADR}}\vdash \&Cnf\rightarrow(y=z)$ for some variable $z\in N$. Let $Cnf_{x:=[y]}$ be the set of configurations $(I_{x:=[y]}, N_{x:=[y]}, S')$ where

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

b. $N_{x:=[y]}=N$, and

c. S' is consistent with

$$ncl(\{e'=e'' : e',e''\in AE(I\backslash\{x\}),$$

$$\boldsymbol{T_{ADR}}\vdash \&Cnf\rightarrow(e'=e'')\}\cup$$

$$\cup\{e'\neq e'' : e',e''\in AE(I\backslash\{x\}),$$

$$\boldsymbol{T_{ADR}}\vdash \&Cnf\rightarrow(e'\neq e'')\}).$$

Then let $aft(D, x{:=}[y])$ be $rfn(\cup_{Cnf\in D}Cnf_{x:=[y]})$.

6) If x is some address variable, distribution $aft(D, dispose(x))$ is obtained as follows. Let $Cnf=(I,N,S)$ be an arbitrary configuration in D. Make *un-initialization warning* if $x\notin I$. Make *invalid-access warning* if $\boldsymbol{T_{ADR}}\vdash \&Cnf\rightarrow(x=y)$ for some variable $y\in N$. Let $Cnf_{dispose(x)}$ be $(I_{dispose(x)}, N_{dispose(x)}, S_{dispose(x)})$ where

a. $I_{dispose(x)}=I$,

b. $N_{dispose(x)}=N\cup\{x\}$, and

c. $S_{dispose(x)}=S$.

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

7) Compound Programs:

a. $aft(D, (\alpha\ ;\beta)) = aft(aft(D,\alpha), \beta)$,

b. $aft(D, if\ \varphi\ then\ \alpha\ else\ \beta) = rfn(aft(D,\alpha)\cup aft(D,\beta))$,

c. $aft(D, while\ \varphi\ do\ \alpha) = rfn(\cup_{n\geq 0}aft(D,\alpha^n))$ where $\alpha^0=skip$, and $\alpha^{k+1}=(\alpha^k\ ;\alpha)$.

IV. Results and Conclusion

Stack-based alias calculus for programming language MoRe is safe in the following sense.

Theorem: *Let D be any alias distribution, α be any MoRe-program and $s=(st, hp)$ be any state such that $st\models D$; if $s'=(st',hp')$ is a state such that $s<\alpha>s'$ then $st'\models aft(D,\alpha)$.*

This theorem can be proven by routine induction on program structure. Its formal proof will be published in the full (journal) version of the paper soon.

More interesting questions are relations between run-time exceptions and warnings that may be casted in exercise *aft*-transformation. In particular, if a program α started in a state s aborts due to re-initialization of some address variable then re-initialization warning will be casted in $aft(D,\alpha)$; if α uses un-initialized address variable then un-initialization warning will be casted in $aft(D,\alpha)$.

But how run-time memory leaks do relate to memory-leak warnings casted in exercise of $aft(D,\alpha)$? How run-time invalid memory accesses do relate to invalid-access warnings casted in $aft(D,\alpha)$? If to answer these questions then these warnings can be used for errors prediction.

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

- $x{:=}\ cons(1)\ ;\ x{:=}\ cons(2)$,
- $y{:=}\ x\ ;\ dispose(x);\ dispose(y)$.

if to compute *aft({({x, y}, ∅, ∅)}, (x:= cons(1) ; x:= cons(2))* then memory-leak warning will be casted; if to compute *aft({({x, y}, ∅, ∅)}, (y:= x ; dispose(x); dispose(y))* then invalid-access warning will be casted.

Recall that the primary purpose of our position paper is to design and present an alias calculus for more realistic programming language than the original one in [4]. Our calculus is control flow insensitive and use only stack variables for analysis. So the most evident topics for further research are design and development of a sensitive to control flow calculus that takes into account some information about the heap. Only after that it will make sense to put into research agenda prototyping on base of the calculus of some alias analysis tool and try it for some benchmarks.

References

[1] R. Haberland and S. Ivanovskiy, Dynamically Allocated Memory Verification in Object-Oriented Programs using Prolog. Accepted for publication in Proc. of Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE-2014, May 29-31, 2014, Saint Pitersburg). Institute for System Programming of the Russian Academy of Sciences (ISPRAS). DOI: 10.15514/SYRCOSE-2014-8-7. Availabale at http://syrcose.ispras.ru/2014/files/submissions/07_syrcose2014.pdf.

[2] M. Hind, Pointer Analysis: Haven’t We Solved This Problem Yet? Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '01), pp.54-61.

[3] C. A. R.. Hoare, The Verifying Compiler: A Grand Challenge for Computing Research. Perspectives of Systems Informatics (PSI'2003), SpringerVerlag, Berlin, LNCS., no. 2890, pp. 1-12, 2003.

[4] B. Meyer, Steps Towards a Theory and Calculus of Aliasing. International Journal of Software and Informatics, special issue (Festschrift in honor of Manfred Broy), 2011., pp.77-115.

[5] J.C. Reynolds, Separation Logic: A Logic for Shared Mutable Data Structures. Proceedings of 17th IEEE Symposium on Logic in Computer Science (LICS 2002). IEEE Computer Press., 2002, pp.55-74.

[6] M. Sridharan, S. Chandra, J. Dolby, S.J. Fink, and E. Yahav, Alias analysis for object-oriented programs. In D. Clarke, T. Wrigstad, and J. Noble, editors, Aliasing in Object-Oriented Programming: types, analysis, and verification. Springer, 2013, Lecture Notes in Computer Science, Vol. 7850, p. 196-232.