Experimenting with Alias Calculi for a Simple Imperative Language with Decidable Pointer Arithmetic

(A talk for ruSTEP —

`https://persons.iis.nsk.su/en/ruSTEP`.)

Niolay V. Shilov and Leonid I. Lygin

Innopolis University

October 7, 2021

Outline

Introduction

Programming Language *MoRe*

Memory Leak Calculus for *MoRe*

Implementation and Testing

Conclusion

References

The first Calculus

Alias calculus was proposed by Bertrand Meyer in 2011 in

- Meyer B. *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.

for a model 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.

Further Progress with the Calculus

The main focus of the original Alias Calculus was object-oriented programs, with further developments not on the calculus itself, but rather on additional tooling to provide a whole software verification framework

- Kogtenkov A., Meyer B., and Velder S.Alias calculus, change calculus and frameinference.Science of Computer Programming, vol.97, 2015, pp. 163-172.

The most recent Progress with the Calculus

The most recent development here is the *AutoAlias* tool, presented by

- Rivera V, and Meyer B.AutoAlias: Automatic Variable-Precision Alias Analy-sis for Object-Oriented Programs. SN Computer Science, vol. 1, n.12, 2020

which introduces *variable-precision* aliasing analysis, meaning you can adjust your precision level depending on your needs (whether you need "fast and dirty" or "slow and accurate").

MoRe **Calculus**

In 2014 by N. Shilov, A. Satekbayeva, and A. Vorontsov suggested in

- Shilov N., Satekbayeva A., Vorontsov A. *Alias calculus for a simple imperative languagewith decidable pointer arithmetic.* Bulletin of the Novosibirsk Computing Center, series: Computer Science. 2014. n. 37, pp. 131-147.

an alias calculus for a more realistic (but still a model) procedural programming language *MoRe* that has addressable memory and pointer arithmetic.

Calculi story in brief

MoRe **Calculi**

In the talk we report an implementation of an aliasing analysis prototype tool for *MoRe*. The tool is based on a new *light* version of the alias calculus designed for memory leak analysis. It had been tested using a number of short snippets of *MoRe* code that contain various memory leak bugs and has demonstrated its correctness (by finding these bugs).

Programming Language *MoRe*

(Abstract) Syntax

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

Programming Language *MoRe*

Types

- 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.
- Address values result from integer values after the implicit type casting, and integer expressions are interpreted as addresses only in the special syntactic context.

Memory Model

A memory consisting of two disjoint parts:

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

A state is a 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 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. $hp : ADR \xrightarrow{fin} INT$.

Semantics of Expressions (terms)

- 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, use of an undeclared variable, or use of a variable with an indefinite value.

Programming Language *MoRe*

Semantics of Formulas

Since the logical formulas F are constructed using the Boolean connectives from equalities and inequalities, 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 $st \models ? \phi$.

Semantics of Formulas (cont.)

- if both expressions of an equality/inequality have the definite values in *st*, the truth value of this equality/inequality is according 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.

Structural Operational Semantics (SOS)

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).

Structural Operational Semantics (cont.)

The inference rules are syntax-driven and have the following form:

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

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.

Sample axioms: Variable Declaration

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:

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

Here and after $(a \mapsto b)$ denotes a singleton function with the graph $\{(a, b)\}$.

Sample axioms: Direct Assignment

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:

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

Here and after $upd(f, a, b)$ denotes an update for the function f, i.e. such a function that for every argument value c

$$upd(f, a, b)(c) = \begin{cases} b, \text{ if } a \equiv c, \\ f(c), \text{ if } c \not\equiv a. \end{cases}$$

Sample axioms: Indirect Assignment

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:

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

Sample axioms: Memory Allocation

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:

- $\overline{(st,hp)\langle x{:=}cons(c_0,\dots c_k)\rangle(upd(st,x,l),\ hp\ \cup\ hp')}$
  if $x \in dom(st)$,
  addresses $in2ad(l+0), \dots in2ad(l+k)$ are disjoint,
  $\{in2ad(l+0), \dots in2ad(l+k)\} \cap dom(hp) = \varnothing$,
  and $hp' = ((in2ad(l+0) \mapsto c_0),\ \dots\ (in2ad(l+k) \mapsto c_k))$;

- $\overline{(st,hp)\langle x{:=}cons(c_0,\dots c_k)\rangle abort}$ otherwise.

Sequential Composition Inference Rule

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

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

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:

- $\dfrac{s\langle \alpha \rangle s'}{s\langle if\ \ \phi\ \ then\ \ \alpha\ \ else\ \ \beta \rangle s'}$ if $s.st \models \phi$
- $\dfrac{s\langle \beta \rangle s'}{s\langle if\ \ \phi\ \ then\ \ \alpha\ \ else\ \ \beta \rangle s'}$ if $s.st \not\models \phi$
- $\dfrac{}{s\langle if\ \ \phi\ \ then\ \ \alpha\ \ else\ \ \beta \rangle abort}$ 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:

- $\dfrac{s\langle\alpha\rangle s' \quad s'\langle while\ \ \phi\ \ do\ \ \alpha\rangle s''}{s\langle while\ \ \phi\ \ do\ \ \alpha\rangle s''}$ if $s.st \models \phi$
- $\dfrac{}{s\langle while\ \ \phi\ \ do\ \ \alpha\rangle s}$ if $s.st \not\models \phi$
- $\dfrac{}{s\langle while\ \ \phi\ \ do\ \ \alpha\rangle abort}$ if $s.st \models ?\phi$

Memory Leak Calculus for *MoRe*

Preliminaries

Let us fix a MoRe-program and refer the program as a (program) *context*. All variables, expressions and programs within this section are variables, expressions and sub-programs of this fixed context.
An *address variable* is any variable x that occurs (in the context) in

- the left-hand side of any memory allocation $x := \mathit{cons}(\dots)$,
- a variable in the left-hand side of any indirect assignment $[x] := \dots$,
- a variable in the right-hand side of any dereferencing $\dots := [x]$,
- a variable in any memory deallocation operator $\mathit{dispose}(x)$,
- any address expression;

Preliminaries (cont.)

Address expressions (in the context) 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 $x + 1, \ldots\ x + k$ such that the program has the memory allocation $x := cons(c_0, \ldots, c_k)$.

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.

Memory Leak Calculus for *MoRe*

Expression Aliasing

- A pair of *aliases* (*synonyms*) is an equality of two address expressions.
- Recall that all address expressions in AE are linear expressions with integer coefficients.

Hence the pairs of synonyms over AE look like Diophantine equations over integers. Nevertheless we consider all these pairs as equations over $(ADR, 0, 1, +, -)$ assuming implicit type casting.

Memory Leak Calculus for *MoRe*

Configurations

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

- a set $I \subseteq AV$ of address variables,
- a set of address expressions $A \subseteq AE(I)$.

Comment: The set I represents currently available initialized address variables, the set A — currently available address expressions that point onto the allocated memory, and the set S is a system of equalities specifying which expressions currently definitely are aliases.

(Alias) Configurations

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

- $\&Cnf$ be the conjunction of all pairs of synonyms in S;
- the *closure* $cls(Cnf)$ be the set of synonyms $\{e' = e'' \ : \ e', e'' \in AE(I),\ T_{ADR} \vdash \&Cnf \to (e' = e'')\}$.

A state $s = (st, hp)$ satisfies the configuration Cnf $(s \models Cnf)$, 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 in $cls(Cnf)$ are valid in s, i.e. $in2ad(st(e')) = in2ad(st(e''))$ for every pair of synonyms $e' = e''$ in S.

(Alias) Distributions

- For any two 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 $T_{ADR} \vdash \& Cnf' \rightarrow (e' = e'')$ (and vice versa).
- 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.

The Calculus

We define the distribution converter

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

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 is executable in nature (i.e. the definition is an algorithm) and an exercise of this algorithm may cast (i.e. make) some warnings in run-time.

Individual Operators

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

- $aft(D, skip) = D$;
- $aft(D, var\ x = c) = D$, 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 y is not an address variable.

Individual Operators (cont.)

If x is any address variable, the distribution $aft(D, \underline{var\ x = c})$ 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 $e'' \in A\}$,
- $S_{var\ x=c} = cls($ $\{e' = e''\ : e', e'' \in AE(I_{var\ x=c})$ and $T_{ADR} \vdash \& Cnf \to (e'_{c/x} = e''_{c/x})\})$.

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

Individual Operators (cont.)

If x is any address variable, the distribution $aft(D, \underline{x := t})$ 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) then the algorithm makes *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 $e'' \in A\}$,
- $S_{x:=t} = cls($ $\{e' = e'' \ : e', e'' \in AE(I_{x:=t})$ and $T_{ADR} \vdash \& Cnf \to (e'_{t/x} = e''_{t/x})\})$.

Memory Leak Calculus for *MoRe*

(cont. from the previous slide)

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

Individual Operators (cont.)

The distribution $aft(D,\ \underline{x := cons(c_0, \ldots c_k)})$ is obtained as follows. Let $Cnf = (I, A, S)$ be an arbitrary configuration in D. If $x \notin I$ then the algorithm makes *un-initialization* warning. Let z be a new (fresh) variable and let $Cnf_{x:=cons_k}$ be $(I_{x:=cons_k},\ A_{x:=cons_k},\ S_{x:=cons_k})$, where

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

Memory Leak Calculus for *MoRe*

(cont. from the previous slide)

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

Individual Operators (cont.)

If x is any address variable, the distribution $aft(D, \underline{x := [y]})$ 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 $T_{ADR} \not\vdash \& Cnf \rightarrow (y = e)$ for every $e \in A$, then the algorithm makes *un-allocation* warning. Let z be a new (fresh) variable and let $Cnf_{x:=(new\ z)}$ be $(I_{x:=(new\ z)},\ A_{x:=(new\ z)},\ S_{x:=(new\ z)})$, where

- $I_{x:=(new\ z)} = I$,
- $A_{x:=(new\ z)} = A$,
- $S_{x:=(new\ z)} = cls(\{e' = e''\ :\ e', e'' \in AE(I_{x:=(new\ z)}),\ T_{ADR} \vdash \& Cnf \rightarrow e'_{z/x} = e''_{z/x}\})$.

Memory Leak Calculus for *MoRe*

(cont. from the previous slide)

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

Individual Operators (cont.)

The distribution $aft(D,\ \underline{dispose(x)})$ is obtained as follows. Let $Cnf = (I, A, S)$ be an arbitrary configuration in D. If $x \notin I$, then the algorithm makes *un-initialization* warning. If $T_{ADR} \nvdash \& Cnf \to (x = e)$ for every $e \in A$, then the algorithm makes *un-allocation* warning. Let $Cnf_{dispose(x)} = (I_{dsp(x)}, A_{dsp(x)}, S_{dsp(x)})$, where

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

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

Memory Leak Calculus for *MoRe*

Compound Programs

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

Overall Design

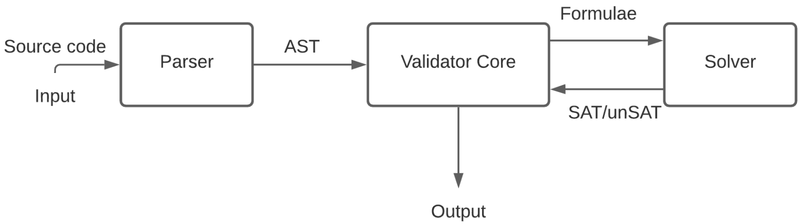

Figure: Linter structure

Outline of the Data Flow

- Source code comes in through the parser (and syntax analyzer) that transforms it into an AST (abstract syntax tree).
- The AST is then passed into the validator core, where it is first pre-analyzed for address expressions and other information, and then processed statement-by-statement using the *aft* transformer that produces warnings in the process.
- Some operations can lead to inconsistent systems. The solver is invoked to resolve the situation.

Parser

To parse the MoRe source code, we used `tspeg` — a tool that generates a parser (and a syntax analyzer) in TypeScript using a PEG (parsing expression grammar) — a definition language that is similar to context-free grammar rules but provides a different interpretation for the choice operator.

Processing Distributions

- Advancing the distribution requires checking whether a system of equations with address variables implies another equality of inequality.
- Solving such a problem in general is very hard, but our language only allows linear expressions with address variables - expressions of form $\mathbf{ax} + \mathbf{b}$, where $\mathbf{a}$ and $\mathbf{b}$ are all integer vectors and $\mathbf{x}$ is a vector of address variables.
- Checking satisfiability for a system is then an ILP (integer linear programming) problem, which can already be solved efficiently by existing methods.

Solving Logical Inference

To check if a boolean formula S implies another linear constraint c_{new}, one can just check if introducing the inverse of the new constraint, $\neg c_{new}$, is making the formula unsatisfiable:

$$S \rightarrow c_{new} \Leftrightarrow \neg sat(S) \vee (sat(S) \wedge \ \neg sat(S \wedge \neg c_{new}))$$

The important part is $sat(S) \wedge \neg sat(S \wedge \neg c_{new})$, which means that if the implementation holds onto a satisfiable formula, checking implication of another constraint can be reduced to checking satisfiability of a slightly tweaked formula.

MIP vs. SMT

- Firstly we attempted to use existing MIP (Mixed-Integer-Programming) solvers. Unfortunatly, all tried MIP solvers (`python-mip`, and `google-ortools`) do not have any method to to convert a complicate propositional combination of integer linear constraints to DNF (Disjunctive Normal Form, disjunction-of-conjunctions).
- So we have replaced this whole module with `z3` — an SMT solver, instead of a MIP one. The reason for this is that the previous MIP solver could only check satisfiability of a system of equations in the usual meaning (i.e., a conjunction of equations), `z3`, being an SMT solver, can check satisfiability for any boolean formula of equations (in our case — any “tree” of conjunctions, dis junctions, and equations themselves as propositions).

Evaluation

The evaluation is done on *fabricated* simple examples, because all considered real-world applications are not applicable for one (or both) of the following reasons:

- The memory-related programming error is related to object-oriented variable scope or lifetime. A lot of these examples were in C++, where there are smart pointers, move constructors, and the memory model is more complex than the one in *MoRe*, thus the implemented validator is unable to "understand" what is going on.
- The memory-related programming error is related to a missing `free()` call. As *MoRe* does not support procedure calls, it has no concept of variable scope, and the validator cannot tell that exiting a function without freeing the array should be an error.

Benchmarks and Metrics

- The “run-time” chart displays (in seconds) how much time did the linter take to process the corresponding program.
- The “solver calls” chart displays how many times did the linter call the solver core. This requires a separate process, and is generally slow, so this is a useful metric to minimize.
- The “equations” chart displays the total amount of equations at the end of processing the program. This metric is useful because solver’s complexity involves it, so minimizing amount of equations maximizes performance.

Benchmarks and Metrics (cont.)

In general (with exceptions being “disposeUnallocated” and “whileUnallocated”) the run-time of the linter decreases in the following order: “non-light without cache” (NL-NC), “non-light with cache” (NL-C), “light without cache” (L-NC), “light with cache” (L-C).

Contribution Summary

- We define a new variant of alias calculus (designed for memory leaks analysis) with support for pointer arithmetic;
- a validation tool based on the calculi was prototyped (with aid of the z3 solver) and evaluated its performance on a set of program examples.

Limitations and Future Work

- Testing has proven perspectives of the Calculi for static analysis, while implementation scalability and utility for industrial code analysis need further research.
- One other possible improvement is scalability of the theory — the current calculi are *intraprocedural*, meaning that the underlying language, *MoRe*, does not support procedure calls. That is a significant drawback, because the calculus does not have any way of determining one major class of memory leaks regarding “hanging” pointers that are not freed when a procedure (function) exits.

Limitations and Future Work (cont.)

The tool was only evaluated without comparison with other tools on a limited set of example programs, which presents two major further research avenues:

- Compare the implemented tool with other similar tools. This might showcase strengths and weaknesses, as well as provide useful insights into how to further improve the tool.
- Test the implementation using larger examples. All presented examples contain no more than 20 lines of code. Testing using larger examples might show major performance bottlenecks.

Main References

Andersen L.O. *Program Analysis and Specialization for the C Programming Language.* Ph.D. Thesis, DIKU, University of Copenhagen, Denmark, 1994.

Kogtenkov A., Meyer B., and Velder S. *Alias calculus, change calculus and frame inference.* Science of Computer Programming, vol.97, 2015, pp. 163-172.

Meyer B. *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.

Reynolds J.C. 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.

Rivera V, and Meyer B. *AutoAlias: Automatic Variable-Precision Alias Analysis for Object-Oriented Programs*. SN Computer Science, vol. 1, n.12, 2020, 15 p. `https://doi.org/10.1007/s42979-019-0012-1`

Shilov N., Satekbayeva A., Vorontsov A. *Alias calculus for a simple imperative languagewith decidable pointer arithmetic.* Bulletin of the Novosibirsk Computing Center, series: Computer Science. 2014. n. 37, pp. 131-147.

Steensgaard B. *Points-to Analysis in Almost Linear Time.* POPL'96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1996, pp. 32-41.