Автономная некоммерческая организация высшего образования
«Университет Иннополис»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(БАКАЛАВРСКАЯ РАБОТА)
по направлению подготовки
09.03.01 - «Информатика и вычислительная техника»

GRADUATION THESIS
(BACHELOR'S GRADUATION THESIS)
Field of Study
09.03.01 – «Computer Science»

Направленность (профиль) образовательной программы
«Информатика и вычислительная техника»
Area of Specialization / Academic Program Title:
«Computer Science»

Тема / Topic	**Исчисление алиасов в языках семейства C / Alias Calculus in C-like languages**

Работу выполнил / Thesis is executed by	**Лыгин Леонид Ильич / Leonid Lygin**	подпись / signature
Руководитель выпускной квалификационной работы / Supervisor of Graduation Thesis	**Шилов Николай Вячеславович / Nikolay Shilov**	подпись / signature

Иннополис, Innopolis, 2021

Contents

List of Tables

List of Figures

Abstract

We present an implementation of an aliasing analysis tool for *MoRe*, a toy procedural programming language that has addressable memory. A tool is based on a light version of Alias Calculus with decidable address arithmetic and is designed for memory leak analysis. The tool had been tested using a number of short snippets of *MoRe* code (single functions) that contain various memory-related bugs and has proven its correctness (by finding these bugs).

Chapter 1

Introduction

Even though a lot of modern code is written in languages with automatic memory management and garbage collection, a large portion of critical code is still written in procedural C-like languages with direct memory access — especially in "tighter" environments such as microcontrollers or performance-critical datacenters. Unlike "managed" languages, C allows the programmer to access memory directly at almost any address because all pointers to memory are just integers and it is always possible to cast any integer to a pointer. While this enables many efficient applications, it is also dangerous because there are no barriers to prevent a program from non-authorized access (e.g. to non-allocated memory) or memory leaks (e.g. allocating arrays and then "forgetting" about them), some of which would result in a runtime crash.

Automated tests might help in some cases, but testing for memory leaks (situations that occur when a program allocates a piece of memory and then "loses" all references to it, thus being unable to ever dispose of it), is notoriously hard.

Even efficient automated testing for memory leaks does not cover every

single case — coverage metrics usually count only statements and branches, but not all possible combinations. It is much better to verify memory usage statically, without ever executing the target program, because it allows ready-to-use automated tools to be integrated into IDEs (integrated development environments), code editors, and CI (continuous integration) pipelines.

Memory usage validation in presence of pointer arithmetic, a situation where a program can try to access memory at any "computable" memory location with an integer address, can be solved with a variety of methods with varying degrees of precision and efficiency. One specific problem in this area is *aliasing analysis* — determining whether two pointers may (or must, depending on the approach) point to the same location on the heap. Aliasing can occur even without pointer arithmetic — many modern languages have support for *references* (or generally only use references to store values), and having support for references inside variables implies that there might be two variables that "point to" the same location (i.e., contain the same reference).

In this Thesis, we present an implementation of a version of one of aliasing analysis methods — a *light* variant of the *Alias Calculus* introduced by Shilov, Satekbayeva, and Vorontsov [1], originally inspired by Meyer [2]. The original Calculus "calculates", given a program in a toy procedural language with pointer arithmetic, a set of synonyms and antonyms ($x = y$ and $x \neq y$ respectively for some two expressions x and y) that describe the memory distributions.

The main research problem addressed in the present work is design of a light version of the Calculus (based on synonyms exclusively), its implementation, and experimental validation of its utility, correctness, and efficiency in detecting memory leaks in procedural programs.

The rest of this Thesis is structured as follows: Chapter 2 outlines existing

research in the problem domain, Chapter 3 defines the variant of alias calculus that is implemented, Chapter 4 describes the implementation details, Chapter 5 contains an evaluation of the resulting implementation, and Chapter 6 summarizes the process and results.

Chapter 2

Literature Review

Aliasing static analysis is a topic that has been under research for more than 30 years, though are major problems still open in the area. There are several methods on how to approach the problem, we mainly focus on two — Separation Logic (introduced by John C. Reynolds and Peter O'Hearn and later developed by Samin Ishtiaq and Hongseok Yang), and Alias Calculus (introduced by Bertrand Meyer), both described below. Section 2.1 describes the process and criteria for the literature review, section 2.2 addresses alias calculus, section 2.3 addresses separation logic, section 2.4 elaborates on alternative methods, and section 2.6 presents an overview of memory models.

2.1 Criteria & Process

Search for related literature was conducted using three methods — by searching on keywords, by following references from previously found papers, and by using direct recommendations from the supervisor. The search keywords were: "alias calculus", "separation logic", "software verification", "aliasing static analysis". The surveyed literature was then filtered using the following criteria:

- Citations — heavily cited sources are generally more reliable and credible.
- Relevance to the topic — some sources were more applicable for aliasing, and some others were more on the general topic of software verification.
- Recency — newer sources were more likely to contain relevant information.

2.2 Alias Calculus

One promising theory for aliasing analysis is *Alias Calculus*, first published by Meyer [2]. It introduces a notion of an alias relation, which is a binary relation on expressions in the program that answers the question "can certain expressions alias at this point in the program?" (hence the name — *Alias* Calculus) and shows a method to modify such relation given a program statement, which can be used to inductively calculate the alias relation for any point in the program. 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 [3].

The only recent development here is the AutoAlias tool, presented by Rivera and Meyer [4], 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").

Unfortunately, these methods, when used without additional modifications, assume that programs are written in a language with managed memory, where the programmer is not exposed to "raw" pointers on the heap, which is not true in case of C and many other languages from the C family.

2.3 Separation Logic

That assumption can be alleviated by *separation logic*, first introduced by Reynolds [5]. It provides us with a method to describe the integer-addressed heap, as well as with another logic operator — the separating conjunction, which asserts that its operands hold on separate parts of the heap (hence the name — *Separation* logic) — which proves useful when describing operations on such heap. It also provides us with a replacement for the constancy rule — the frame rule — that is valid in presence of aliasing and provides a method to extend a "local" specification to a "global" one.

Separation logic has been successfully used both on its own to provide automated provers and a basis for further theoretical developments. For example, there is an existing (partly incomplete) implementation of logic-based proofs [6], and there is a sound and complete heap theorem prover built on the same theory [7]. As for examples of theoretical implications, there is a framework for automatic extraction of frame axioms [8], and there is an extension to the theory for programs that use concurrent shared memory [9].

More recently, Krebbers, Jung, Bizjak, *et al.* [10] describes an attempt of "unification" of different separation logics, citing that "there is a disturbing trend for each new library or concurrency primitive to require a new separation logic" taken from [11]. That paper introduces a base layer of logic for their proof assistant — Iris.

2.4 Other Formalisms and Methods

There are also other methods of alias analysis. Many of those use the machine language rather than the source code itself, which might lose some

semantic information available. For example, Heintze and Tardieu [12] develop an "ultra-fast aliasing analysis" for C which uses machine code for its internals but unfortunately does not consider address arithmetic at all. Earlier, Debray, Muth, and Weippert [13] provides a method that can understand address arithmetic, but on the other hand does not consider cases of indirect dereferencing (consider a program $b := [a];\ c := [b]$, here the analysis would not know anything about the contents of c).

2.5 Andersen's Pointer Analysis

The most cited practical approach to pointer analysis (and program analysis with respect to memory in general) is *Andersen's Pointer Analysis*, originally introduced in Andersen [14]. Andersen's Analysis is a *points-to* analysis, not an *alias* analysis, and there are some cases where the former cannot be used to implement the latter. The most recent development of this theory is the fine-grained and parallel complexity analysis, published in [15], which even stirred up some activity in the industry [16].

2.6 Memory Models

One specific culprit of C program analysis is a good memory model that can incorporate pointer arithmetic. In this case, "memory model" is not a description of how threads interact, but rather a description of the details of memory addressing. A "trivial" model might be to consider that assignment will bind the name of the variable to the right-hand side of the assignment, but that does not work in presence of aliasing (consider $x := y;\ [x] = 4$, where $[y]$

changes, but is not referenced on the left-hand side anywhere). To include the possibility of using pointers into the model, Zhang [17] proposed using an array with addresses used as indices, but that makes representing complex structures with uneven sizes awkward. To resolve these problems, Xu, Kremenek, and Zhang [18] introduces a hierarchical region-based memory model where each region can have a "parent", and there are 3 basic super-regions — for local variables, global variables, and dynamically-allocated heap memory.

2.7 Conclusion on Literature Review

It is possible to say that research on theory of alias analysis is mainly based on separation logic or alias calculus with dominance of separation logic, and still requires fundamental studies. In part because of this imbalance, we attempted research of alias analysis in a framework of alias calculus but for a language with pointer arithmetic.

Chapter 3

Theoretical Background

This chapter presents a quick recap of the theoretical basis on which this paper is based, presented originally by Shilov, Satekbayeva, and Vorontsov [1], starting with the programming language definition, and then describing the alias calculus for it.

The chapter just summarizes of definitions and models from the original paper and fixes some typos, along with being more easily "digestible" for engineers (as opposed to the original paper, which is more easily "digestible" for mathematicians).

Section 3.1 describes the programming language *MoRe*, Section 3.2 provides an alias calculus for that language, and Section 3.3 introduces a "light" version of the calculus.

3.1 *MoRe* Programming Language

The programming language for which the calculus is developed is *MoRe*, which stands for *More Realistic* (than languages without pointers). *MoRe* is a representation of a program that might contain arbitrary address arithmetic,

$$\begin{aligned} P ::= \ & skip \mid var\ V = C \mid V := T \mid \\ V := cons(C^*) \mid & [V] := V \mid V := [V] \mid dispose(V) \mid \\ & (P;\ P) \mid (if\ F\ then\ P\ else\ P) \mid (while\ F\ do\ P). \end{aligned} \tag{3.1}$$

and thus it can properly represent a C program.

There are only two data types in the language — *addresses* and *integers*, with a special remark —- only integers can be represented in the language source code, and they are implicitly cast to addresses when needed. The address data type in the original paper can be represented by any (finite or infinite) set of values ADR with constants 0 and 1, and operations + and −, such that $(ADR, 0, 1, +, -)$ is a commutative semi-group with a decidable first-order theory T_{ADR}. Such generic definitions can sometimes result in cumbersome notation, so while it is certainly more general not to fixate on some specific theory, we only consider usual integers, with 0, 1, + and − having their usual meaning in $ADR = \mathbb{N}$.

The integer data type is any, finite or infinite, subset INT of $\mathbb{Z}$ (either proper or improper, meaning that $INT = \mathbb{Z}$ is also allowed), with operations and constants defined analogous to the addresses, and again in our case we consider them to be same as $\mathbb{Z}$, except multiplication for integers is also allowed.

Then, let V be an infinite alphabet of variables, C be a representation of integer literals, T be a language or arithmetic expressions with constants from C and variables from V, and F be a language of logical formulas constructed with equalities and inequalities (= and $\neq$) between expressions from T.

Finally, the syntax of the language is defined in Eq. 3.1.

For brevity, the memory model is omitted, and the semantics of the language is provided as a simple textual explanation, instead of strict inference rules.

3.1.1 MoRe Operational Semantics

The following operational semantics is employed in MoRe:

- *skip* — does nothing.
- *var* $V = C$ — declares a variable, and initializes it with an initial value (variables are required to have definite values during their whole lifetime since the declaration). Roughly corresponds to int x = 0; from the C programming language.
- $V := T$ — assigns the current value of T to V. Corresponds to x = y + 1; from the C programming language.
- $V := cons(C^*)$ — allocates a contiguous block of dynamic memory, and stores the pointer to the beginning into V. $x := cons(1, 4, 10)$ corresponds to x = malloc(3 ∗ sizeof(int)); x[0] = 1; x[1] = 4; x[2] = 10; from the C programming language (using some allocator malloc).
- $[V] := V$ — stores the value from the right-hand variable into the memory defined by the address from the left-hand variable. $[x] := y$ corresponds to ∗x = y; from the C programming language.
- $V := [V]$ — loads the value from the memory defined by the address from the right variable into the left variable. $x := [y]$ corresponds to x = ∗y; from the C programming language.
- *dispose*(V) — frees the memory block whose beginning's address is the value of the variable. *dispose*(x) corresponds to free(x); from the C programming language.

- $(P;\ P)$ — sequentially executes both programs, the left one before the right one.
- *if* F *then* P *else* P — if F is true, executes the left-hand program (the "then"-branch), otherwise, executes the right-hand program (the "else"-branch).
- *while* F *do* P — continually executes P as long as F remains true. If F is not true from the very first time the execution reaches this statement, P is not executed at all.

Some constructions from C are unrepresentable — for example, while a simple function call might be inlined using some variable renaming (the variable set is infinite after all), *recursive* function calls are not possible.

Nevertheless, *MoRe* is a reasonable representation that still allows a prototype implementation to detect bugs in real programs.

3.2 Alias Calculus

This section describes the original alias calculus for the programming language MoRe, defined above.

3.2.1 Preliminaries

In order to define the calculus, it is necessary to introduce some additional definitions. All of them assume that there is a single program that the calculus needs to validate.

Let AV - the set of address variables, and AE - the set of address expressions be 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 store statement $[x] := \dots$
  - the right-hand side of any dereferencing $\cdots := [x]$
  - 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, that 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 $x := cons(c_0, \dots, c_k)$

Let (informally) $AE(D)$ for $D \subset AV$ be a set of all address expressions that contain *only* variables from D.

A *synonym* is an equality of two address expressions, $x = y$ for some $x, y \in AE$. An *antonym* is an inequality of two address expressions, $x \neq y$ for some $x, y \in AE$.

Then, a configuration is a triple (I, A, S) consisting of:

- a set $I \subset AV$ of current address variables
- a set of current address expressions $A \subset AE$

- a *consistent* set of synonyms and antonyms S, with variables in I

In this case, a set of synonyms and antonyms being *consistent* means that S has at least one solution as a system of equalities and inequalities. Informally, I is the set of initialized address variables, A is the set of all address expressions that map onto the allocated memory, and S is a system that describes which two expressions can alias (be equal) in the program at a specific point.

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

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

Above, *cls* stands for *closure* and *ncl* stands for *negative closure*.

Informally, $cls(Cnf)$ are pairs of address expressions that have meaning in the configuration (i.e., containing only defined variables) that can either be proven equal (using S) or not equal. Then, $ncl(Cnf)$ is (again, informally) the closure $cls(Cnf)$ extended with inequalities for all pairs that cannot be proven to either be equal or not equal.

Let any two configurations $Cnf_1 = (I_1, A_1, S_1)$ and $Cnf_2 = (I_2, A_2, S_2)$ be *equivalent* if:

- $I_1 = I_2$
- for all $e_1 \in A_1$ there exists $e_2 \in A_2$ such that $\&Cnf_1 \rightarrow e_1 = e_2$ (with $\rightarrow$ meaning the usual "implies"

- $ncl(Cnf_1) = ncl(Cnf_2)$ (which is roughly same as saying that the systems derived from S_1 and S_2 would be equivalent, i.e., have same sets of solutions)

Then, for any syntactic clause e (i.e., an arithmetic expression or a program), let $e_{x/y}$ be same as e, except all instances of x are replaced by y, for example for $e = x + y + 1$, we would get $e_{x/y+3} = (y + 3) + y + 1$.

Let a *memory distribution* (or just *distribution*) be a finite set of mutually not equivalent configurations (i.e., any two configurations from the set are not equivalent). For a set of configurations D, let $rfn(D)$ (*rfn* for refinement) be a distribution obtained from D by leaving a single configuration in each equivalence class.

Finally, for a distribution D and a state s, let $s \models D$ denote that D satisfies s, meaning that $s \models Cnf$ for some $Cnf \in D$.

3.2.2 Transformations

This section describes how the alias calculus works on a program.

For any existing distribution D and a statement p, let $aft(D, p)$ be such a new distribution that a triple $\{s \models D\}p\{s' \models aft(D, p)\}$ is valid, for such s' that $s\langle p\rangle s'$.

For operations that do not change address variables:

- $aft(D, skip) = D$
- $aft(D, var\ x = i) = D$, 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 y is not an address expression

For convenience, let

$$pairs_{x/y}(I', Cnf) = ncl(\{e' = e'' \ : \ e', e'' \in AE(I'), \& Cnf \rightarrow e'_{x/y} = e''_{x/y}\} \cup \{e' \neq e'' \ : \ e', e'' \in AE(I'), \& Cnf \rightarrow e'_{x/y} \neq e''_{x/y}\})$$

With these preliminary definitions out of the way, the following sections describe how *aft* works for each of the individual statements of the language.

Statement $var\ x = y$

If x is an address variable, $aft(D, var\ x = y)$ is a distribution that is obtained by converting all configurations $Cnf \in D$ into $Cnf' = (I', A', S')$ (and then refining the resulting distribution using *rfn*) as follows.

If $x \in I$, an *un-initialization* warning is emitted.

As for the configuration parts:

- $I' = I \cup \{x\}$.
- $A' = \{e' \in AE(I') \ : \ \& Cnf \rightarrow e'_{x/y} = e$ for some $e \in A\}$
- $S' = pairs_{x/y}(I', Cnf)$

Statement $x := y$

If x is any address variable, $aft(D, x := y)$ is a distribution that is obtained by converting all configurations $Cnf \in D$ into $Cnf' = (I', A', S')$ (and then refining the resulting distribution using *rfn*) as follows.

If $x \notin I$ or y contains any variables that are not in I, an "*un-initialized variable*" warning is emitted.

As for the configuration parts:

- $I' = I$.
- $A' = \{e' \in AE(I') \ : \ \&Cnf \rightarrow e'_{x/y} = e \text{ for some } e \in A\}$
- $S' = pairs_{x/y}(I', Cnf)$

If there is an expression $e \in A$ such that there are no $e' \in A'$ with $\&Cnf \rightarrow e'_{x/y}e$, a "*memory leak*" warning is emitted. Informally, there is no way to reach e from the new configuration.

Statement $x := cons(c_0, \ldots, c_k)$

$aft(D, x := cons(c_0, \ldots, c_k))$ is a distribution that is obtained by converting all configurations $Cnf \in D$ into $Cnf' = (I', A', S')$ (and then refining the resulting distribution using rfn) as follows.

If $x \notin I$, then an "*un-initialized variable*" warning is emitted.

Let y be a new temporary variable (the set of variables is infinite, so we can create new ones for our purposes). Then, let $\&Cnf_y$ be a temporary configuration which is a conjunction of the original $\&Cnf$ with the following antonyms:

- For each $e \in AE(I), 0 \leq i \leq k$, the antonym is $e \neq y + i$. Informally, this states that the newly-allocated piece of memory is disjoint with all other allocated pieces of memory.
- For each $0 \leq ij \leq k$, the antonym is $y + i \neq y + j$. Informally, this signifies that the individual cells of the newly-allocated piece of memory do not intersect (there are no two cells that point to the same location).

The above definition is a "helper" configuration that is used to define the new configuration:

- $I' = I$
- $A' = A \cup \{x, x+1, x+2, \ldots, x+k\}$
- $S' = ncl(\{e' = e'' \ : \ e', e'' \in AE(I'), \& Cnf_y \to e'_{x/y} = e''_{x/y}\} \cup \{e' \neq e'' \ : \ e', e'' \in AE(I'), \& Cnf_y \to e'_{x/y} \neq e''_{x/y}\})$.

Note that in the expression for S', the content is almost exactly the same as $pairs_{x/y}(I', Cnf)$, except that we use $\& Cnf_y$ instead of $\& Cnf$.

If there is an expression $e \in A$ such that there are no $e' \in A'$ with $\& Cnf \to e'_{x/y} e$, then a "*memory leak*" warning is emitted. Informally, there is no way to reach e from the new configuration.

Statement $x := [y]$

If x is any address variable, $aft(D, x := [y])$ is a distribution that is obtained by converting all configurations $Cnf \in D$ into $Cnf' = (I', A', S')$ (and then refining the resulting distribution using rfn) as follows.

If $x \notin I$ or $y \notin I$, an "*un-initialized variable*" warning is emitted.

As for the configuration parts:

- $I' = I$.
- $A' = \{e' \in AE(I') \ : \ \& Cnf \to e'_{x/y} = e \text{ for some } e \in A\}$
- $S' = pairs_{x/y}(I', Cnf)$

3.3 “Light” Calculus

One might notice that the only usage of the system (S from the configuration) inside the calculus is to check whether a system implies equality of two expressions. Furthermore, all warnings that are produced as a result of invoking the algorithm only check for equality. Informally, reasoning that an inequality cannot be implied from a combination of equalities, the implementation has an option to employ a “light” version of the calculus — one that does not have any inequalities stored inside the system.

This both reduces the complexity of the underlying theory, and improves performance.

For completeness, this section provides a full explanation of the “light” variation of the Calculus in further subsections.

3.3.1 Preliminaries

Let AV - the set of address variables, and AE - the set of address expressions be 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 store statement $[x] := \dots$
  - the right-hand side of any dereferencing $\cdots := [x]$
  - 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, that 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 $x := cons(c_0, \ldots, c_k)$

Let (informally) $AE(D)$ for $D \subset AV$ be a set of all address expressions that contain *only* variables from D.

A *synonym* is an equality of two address expressions, $x = y$ for some $x, y \in AE$.

Then, a configuration is a triple (I, A, S) consisting of:

- a set $I \subset AV$ of current address variables
- a set of current address expressions $A \subset AE$
- a *consistent* set of synonyms S, with variables in I

In this case, a set of synonyms being *consistent* means that S has at least one solution as a system of equalities. Informally, I is the set of initialized address variables, A is the set of all address expressions that map onto the allocated memory, and S is a system that describes which two expressions can alias (be equal) in the program at a specific point.

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

- $\&Cnf$ be the conjunction of all pairs of synonyms and antonyms in S

- $cls(Cnf) = \{e' = e'' \ : \ e', e'' \in AE(I), \ T_{ADR} \vdash \& Cnf \rightarrow (e' = e'')\} \cup$
  $\cup \{e' \neq e'' \ : \ e', e'' \in AE(I), \ T_{ADR} \vdash \& Cnf \rightarrow (e' \neq e'')\}.$

  Above, *cls* stands for *closure*.

Informally, $cls(Cnf)$ are pairs of address expressions that have meaning in the configuration (i.e., containing only defined variables) that can either be proven equal (using S) or not equal.

Let any two configurations $Cnf_1 = (I_1, A_1, S_1)$ and $Cnf_2 = (I_2, A_2, S_2)$ be *equivalent* if:

- $I_1 = I_2$
- for all $e_1 \in A_1$ there exists $e_2 \in A_2$ such that $\& Cnf_1 \rightarrow e_1 = e_2$ (with $\rightarrow$ meaning the usual "implies"
- $cls(Cnf_1) = cls(Cnf_2)$ (which is roughly same as saying that the systems derived from S_1 and S_2 would be equivalent, i.e., have same sets of solutions)

Then, for any syntactic clause e (i.e., an arithmetic expression or a program), let $e_{x/y}$ be same as e, except all instances of x are replaced by y, for example for $e = x + y + 1$, we would get $e_{x/y+3} = (y + 3) + y + 1$.

Let a *memory distribution* (or just *distribution*) be a finite set of mutually not equivalent configurations (i.e., any two configurations from the set are not equivalent). For a set of configurations D, let $rfn(D)$ (*rfn* for refinement) be a distribution obtained from D by leaving a single configuration in each equivalence class.

Finally, for a distribution D and a state s, let $s \models D$ denote that D satisfies s, meaning that $s \models Cnf$ for some $Cnf \in D$.

3.3.2 Transformations

This section describes how the alias calculus works on a program.

For any existing distribution D and a statement p, let $aft(D, p)$ be such a new distribution that a triple $\{s \models D\}p\{s' \models aft(D, p)\}$ is valid, for such s' that $s\langle p\rangle s'$.

For operations that do not change address variables:

- $aft(D, skip) = D$
- $aft(D, var\ x = i) = D$, 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 y is not an address expression

For convenience, let

$$pairs_{x/y}(I', Cnf) = cls(\{e' = e'' \ : \ e', e'' \in AE(I'), \& Cnf \rightarrow e'_{x/y} = e''_{x/y}\})$$

With these preliminary definitions out of the way, the following sections describe how *aft* works for each of the individual statements of the language.

Statement $var\ x = y$

If x is an address variable, $aft(D, var\ x = y)$ is a distribution that is obtained by converting all configurations $Cnf \in D$ into $Cnf' = (I', A', S')$ (and then refining the resulting distribution using rfn) as follows.

If $x \in I$, an *un-initialization* warning is emitted.

As for the configuration parts:

- $I' = I \cup \{x\}$.
- $A' = \{e' \in AE(I') \ : \ \&Cnf \rightarrow e'_{x/y} = e \text{ for some } e \in A\}$
- $S' = pairs_{x/y}(I', Cnf)$

Statement $x := y$

If x is any address variable, $aft(D, x := y)$ is a distribution that is obtained by converting all configurations $Cnf \in D$ into $Cnf' = (I', A', S')$ (and then refining the resulting distribution using *rfn*) as follows.

If $x \notin I$ or y contains any variables that are not in I, an "*un-initialized variable*" warning is emitted.

As for the configuration parts:

- $I' = I$.
- $A' = \{e' \in AE(I') \ : \ \&Cnf \rightarrow e'_{x/y} = e \text{ for some } e \in A\}$
- $S' = pairs_{x/y}(I', Cnf)$

If there is an expression $e \in A$ such that there are no $e' \in A'$ with $\&Cnf \rightarrow e'_{x/y}e$, a "*memory leak*" warning is emitted. Informally, there is no way to reach e from the new configuration.

Statement $x := cons(c_0, \ldots, c_k)$

$aft(D, x := cons(c_0, \ldots, c_k))$ is a distribution that is obtained by converting all configurations $Cnf \in D$ into $Cnf' = (I', A', S')$ (and then refining the resulting distribution using *rfn*) as follows.

If $x \notin I$, then an "*un-initialized variable*" warning is emitted.

Let y be a new temporary variable (the set of variables is infinite, so we can create new ones for our purposes). Then, let $\&Cnf_y$ be a temporary configuration which is a conjunction of the original $\&Cnf$ with the following antonyms:

- For each $e \in AE(I), 0 \leq i \leq k$, the antonym is $e \neq y + i$. Informally, this states that the newly-allocated piece of memory is disjoint with all other allocated pieces of memory.
- For each $0 \leq ij \leq k$, the antonym is $y + i \neq y + j$. Informally, this signifies that the individual cells of the newly-allocated piece of memory do not intersect (there are no two cells that point to the same location).

The above definition is a “helper” configuration that is used to define the new configuration:

- $I' = I$
- $A' = A \cup \{x, x + 1, x + 2, \ldots, x + k\}$
- $S' = cls(\{e' = e'' \ : \ e', e'' \in AE(I'), \&Cnf_y \rightarrow e'_{x/y} = e''_{x/y}\})$.

Note that in the expression for S', the content is almost exactly the same as $pairs_{x/y}(I', Cnf)$, except that we use $\&Cnf_y$ instead of $\&Cnf$.

If there is an expression $e \in A$ such that there are no $e' \in A'$ with $\&Cnf \rightarrow e'_{x/y}e$, then a “*memory leak*” warning is emitted. Informally, there is no way to reach e from the new configuration.

Statement $x := [y]$

If x is any address variable, $aft(D, x := [y])$ is a distribution that is obtained by converting all configurations $Cnf \in D$ into $Cnf' = (I', A', S')$ (and then refining the resulting distribution using rfn) as follows.

If $x \notin I$ or $y \notin I$, an “*un-initialized variable*” warning is emitted.

As for the configuration parts:

- $I' = I$.
- $A' = \{e' \in AE(I') \ : \ \&Cnf \rightarrow e'_{x/y} = e \text{ for some } e \in A\}$
- $S' = pairs_{x/y}(I', Cnf)$

Chapter 4

Implementation

This chapter describes the implementation of a memory-access linter for *MoRe*. Section 4.1 outlines the overall structure of the implementation, Section 4.2 describes the parser of *MoRe* source code, Section 4.3 describes the validator core itself, Section 4.4 explains the problem that requires an external solver, Section 4.5 details how the implementation invokes the solver, as well as the choice of the solver engine, and Section 4.6 outlines the optimizations performed that deviate from the strict theory but make the implementation run faster.

4.1 Overall structure

The overall structure of the linter implementation can be summed up in Fig. 4.1.

An outline of the data flow:

- Source code comes in through the parser (and syntax analyzer) that transforms it into an AST (abstract syntax tree).

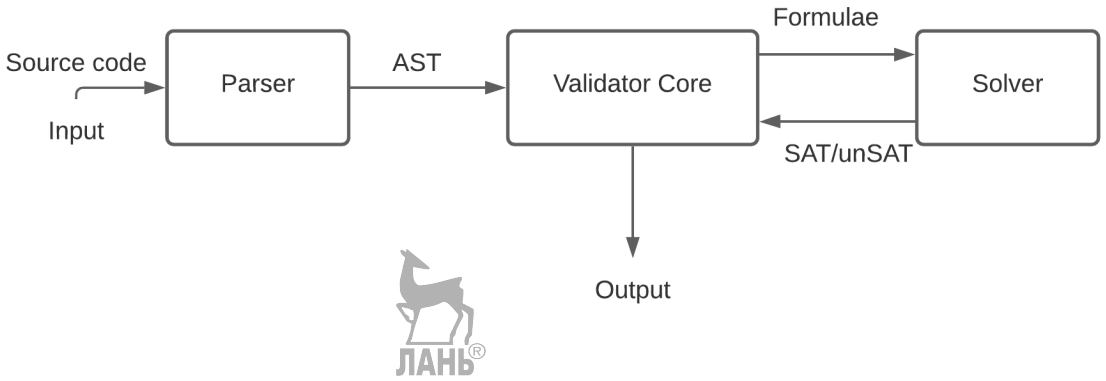

Fig. 4.1. Linter structure

- 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.

The validator core and parser are written in TypeScript, whereas the solver is written in Python, given that there were no restrictions, and these technologies were more familiar than others.

4.2 Parser (Syntax Analyzer)

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.

The PEG source code for the language (and some extensions) is provided

in Appendix B.

The linter uses an extended version of *MoRe* that simplifies the generation of a real-world parser, and allows extending the address expression set from the first pass manually. These extensions are outlined in Subsection 4.2.1 and Subsection 4.2.2.

4.2.1 Program blocks

A *program* is a list of *statements*, but generating a proper parser is simpler when all program blocks are enclosed with a pair of some unique symbols. The parser uses begin and end for that purpose – the example can be seen on Listing 4.1.

Listing 4.1. Program block syntax extension example

```
// this is the root program block
program begin
    var x = 0
    var y = 0
    // this is a program block that
    // contains the "then" branch
    if condition then begin
        x := y
    end else begin
        y := x
    end
end
```

4.2.2 Address expression context

The calculus assumes that the address expressions found in the first pass are all address expressions that are ever used in the program. Consider the program displayed in Listing 4.2. Variable x was pointing to some allocated memory and was then overwritten. That is not an error though, because the reference is "saved" into y first, but as $y + 1$ and $y + 2$ are not used in the program anywhere, the linter has no way of determining that $x + 1$ and $x + 2$ are still reachable addresses after the assignment.

Listing 4.2. Overwriting a variable "leaks" phantom addresses

```
program begin
    var x = 0
    var y = 0

    x := cons(1, 2, 3)
    y := x

    x := 3
end
```

To solve this problem, *MoRe* syntax is extended with an additional "header" block, where one can specify a list of "additional" address expressions in order to tell the linter about the expressions used outside of the current program. The fixed example is displayed in Listing 4.3 and it does not produce any errors, unlike the example in Listing 4.2.

Listing 4.3. Address expression extension example

```
@assume address-expression y + 1, y + 2
```

```
program begin
    var x = 0
    var y = 0

    x := cons(1, 2, 3)
    y := x

    x := 3
end
```

4.3 Linter

The linter is taking the parsed program represented as an AST, and is processing it in two passes:

- "pre"-pass: when it collects all the used address expressions and variables throughout the program, and checks for any obvious syntactic problems.
- "main"-pass: when it performs the actual analysis.

"Pre"-pass is self-descriptive and trivial, so the explanation of implementation is omitted for brevity.

In order to run the "main"-pass, we:

- Create an array variable for the distribution. It is a simple integer-indexed array that would hold all current configurations. Its initial value is a single "empty" configuration $(I, A, S) = (\varnothing, \varnothing, \varnothing)$.

- Iterate through all top-level statements in the program, and for each statement:
  - Iterate through all configurations and replace them according to rules laid out in Chapter 3, emitting warnings into some global array (or printing them out as they appear).
  - After the previous operation, there might have been new configurations in the distribution, or there might have been duplicates as a result of transforming two different configurations, so the distribution is then "refined" (*rfn*) by removing all configurations that have equivalent configurations earlier.

4.4 Integer linear programming

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.

Then, to check if a boolean formula S implies another constraint c_{new} (*constraint* meaning an equality $\mathbf{a}_1\mathbf{x}_1 + \mathbf{b}_1 = \mathbf{a}_2\mathbf{x}_2 + \mathbf{b}_2$ or an inequality $\mathbf{a}_1\mathbf{x}_1 + \mathbf{b}_1 \neq \mathbf{a}_2\mathbf{x}_2 + \mathbf{b}_2$), one can just check if introducing the inverse of the new constraint, $\neg c_{new}$, is making the formula unsatisfiable. For example, consider a predicate *sat* that checks whether a formula is satisfiable (i.e., there exists a set of variable values that are a solution to a system). Then, the equivalence

in Equation 4.1 is true.

$$S \rightarrow c_{new} \Leftrightarrow \neg sat(S) \vee (sat(S) \wedge \neg sat(S \wedge c_{new})) \tag{4.1}$$

The important part is $sat(S) \wedge \neg sat(S \wedge c_{new})$, 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.

4.5 Solver

For some parts of the validator, mainly checking whether an existing synonym set implies another equality or inequality, it is simpler (and more correct) to use an existing solver.

As was stated in Chapter 3, all expressions that are of use for the linter (i.e., all address expressions) are constructed in such a way that all multiplications are between either two integer constants, or between an integer constant and an address. This constraint implies that all address expressions in the program are degree-1 polynomials (with respect to address variables), and thus a system of equalities of these expressions can be formulated as an MIP (mixed-integer-programming) problem.

However, the first attempt at using an existing MIP solver to "power" the linter turned out to be a much bigger hassle than expected, as shown below.

Consider a system with constraints $a_1x_1 = b_1$, $a_2x_2 = b_2$ and a requirement to add a new constraint $a_3x_3 \neq b_3$. The canonical form of an MIP problem does not allow equalities, so the original system must be rewritten to a form displayed in Equation 4.2.

$$\begin{cases} a_1 x_1 \leq b_1 \\ a_1 x_1 \geq b_1 \\ a_2 x_2 \leq b_2 \\ a_2 x_2 \geq b_2 \end{cases} \tag{4.2}$$

Then, adding a new constraint of such form would require introducing a disjunction into the system, displayed in Equation 4.3

$$\begin{cases} a_1 x_1 \leq b_1 \\ a_1 x_1 \geq b_1 \\ a_2 x_2 \leq b_2 \\ a_2 x_2 \geq b_2 \\ \left[\begin{array}{l} a_3 x_3 < b_3 \\ a_3 x_3 > b_3 \end{array}\right. \end{cases} \tag{4.3}$$

All tried MIP solvers (python-mip, and google-ortools) did not have any method to specify such a system in a straightforward way, so we have tried to reorganize these systems to result in a big DNF (disjunctive normal form, disjunction-of-conjunctions). That method turned out to be rather cumbersome, and we found ourselves having a lot of problems unrelated to the research, 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, disjunctions, and equations themselves as propositions). For example, with z3 one can just

directly supply the formula in Equation 4.4 and check its satisfiability.

$$x + y = z \wedge (x + 1 = y \vee x - 1 = y) \tag{4.4}$$

Of course, z3 being more complex than any MIP solver, this module's performance took a hit, but the results were still satisfactory enough for a Proof-of-Concept implementation.

4.6 Optimizations

There are two optimization types that are employed: using the "light" calculus variation and caching subsequent solver calls. These optimizations are described respectively in Subsection 4.6.1 and Subsection 4.6.2.

4.6.1 "Light" calculus optimization

Using the "light" calculus, described in Section 3.3 results in fewer equations in the system, which, in the end, results in better performance. The exact performance gains are presented in Section 5.2.

4.6.2 Solver call cache

Each question of form "does the logical formula S imply $x = y$ for expressions x, y" requires a separate call to the solver, which is not a lightweight operation. Optimizing the performance then can be achieved by reducing the amount of solver calls. One way to reduce the amount of calls is to *memoize* the previous calls — store the result of checking if $S \to x = y$, and if S does not change and x, y do not change, then the answer would not change either.

This fact alone is not that useful — S changes too frequently, but consider two logical formulae S and S'. If there was a method of answering whether $S \wedge S' \to x = y$, given that the answer to whether $S \to x = y$ is already known, that would be a much more useful fact as the equations from the stored formula are rarely removed.

In the "light" version of the calculus, outlined in Subsection 4.6.1, there are no antonyms. Then, both S and S' are constructed using only conjunctions, disjunctions, and equalities of two expressions. Then, $S \to x = y$ already implies $S \wedge S' \to x = y$ — exactly the required implication.

Chapter 5

Evaluation

This chapter describes the evaluation of the resulting implementation. Section 5.1 describes some limitations with respect to the evaluation methods, Section 5.2 displays the results, and Section 5.3 outlines the computational complexity ($\mathcal{O}$).

5.1 Limitations of our approach

The evaluation is done mainly on “fabricated” simple examples, because all considered real-world applications (and many of the complex fabricated examples) 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.

For example, consider Listing 5.1, which follows a popular pattern of allocating memory to do some computation. 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.

Listing 5.1. C program with a memory leak due to absence of free()

```
int fn() {
    int* arr = malloc(16);
    // doing something with arr
    if (everything_is_ok) {
        free(arr);
        return 0;
    } else {
        print_error();
        return -1;
    }
}
```


5.2 Benchmarks

Figures 5.1 through 5.10 display benchmark results for corresponding examples. There are three plots for each example:

- 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.

In general (with exceptions being “disposeUnallocated” and “whileUnallocated”, displayed on Figure 5.3 and Figure 5.9 respectively) 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). The overall order is expected — not introducing any optimizations is slower than introducing only one, which is in turn slower than introducing both at the same time. As for “whileUnallocated” — it still follows same trend, just that “light without cache” is much slower than “non-light with cache”, so just changed the order a bit. The “disposeUnallocated” example, however, is slightly different — L-C turns out to be slower than both NL-C and L-NC! The slowdowns introduced by caching can simply be explained by the additional machinery in place to support the cache taking more time to initialize, but L-C being slower than NL-C is slightly harder to explain.

The “solver call” charts all decrease in the same fashion as the majority of run-time charts — NL-NC, followed by NL-C and L-NC in some order, with L-C being the fastest.

The “equation” charts only have different values for the “light” and “non-light” variants, not changing when introducing a cache. This is an expected result, because caching does not do anything to remove the “add equation to

formula" operations. Otherwise, when moving from "non-light" to "light" the amount of equation drastically decreases in almost all cases — the exception here is the "disposeUnallocated" example, it does not require any additional equations at all.

The values seen in the "solver call" charts seem to follow the same trends as in run-time charts, except for the "conditionOverwrite" example.

5.3 Computational complexity

The overall computational complexity of the linter is a combination of its three parts: parser, described in Subsection 5.3.1, validator core, described in Subsection 5.3.2, and the overall complexity of the linter, described in Subsection 5.3.3.

5.3.1 Parser

Parser's complexity can be summed up as "very efficient for our purposes". Our parser generation tool, tspeg, produces a *recursive descent parser*. These kinds of parsers can take up to $\mathcal{O}(2^T)$ operations to complete (where T is the amount of input tokens), but that could only be achieved with a lot of backtracking and a specially-crafted input — in general simple input strings would result in a more reasonable complexity. Here, each input token is either a non-terminal or a terminal from the PEG presented in Appendix B.

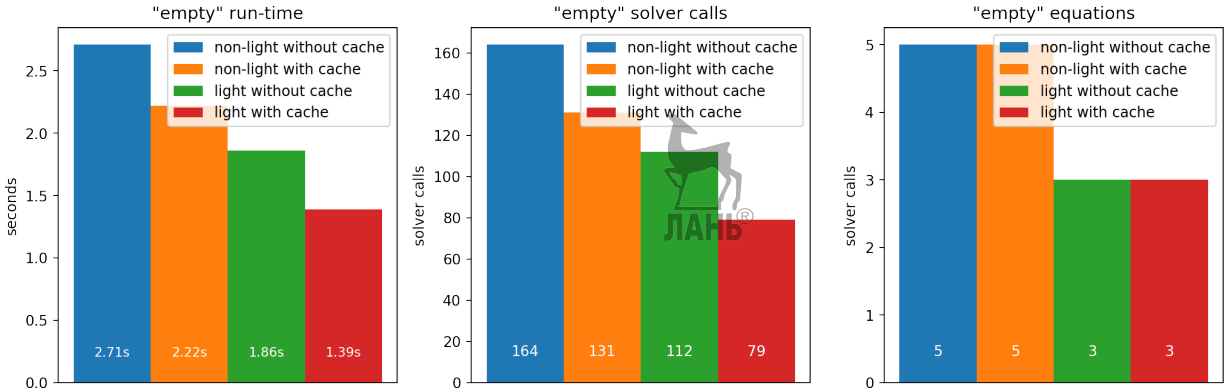

Fig. 5.1. Benchmark of "empty" example

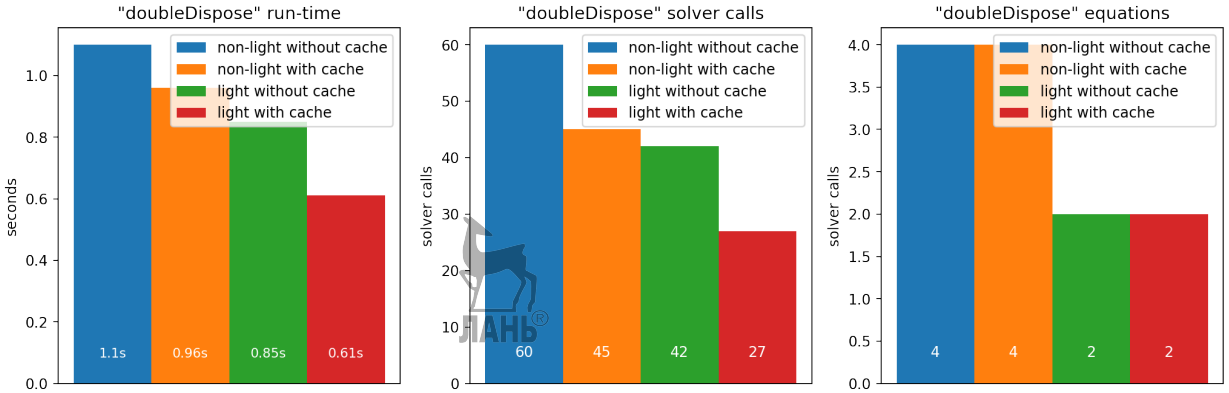

Fig. 5.2. Benchmark of "doubleDispose" example

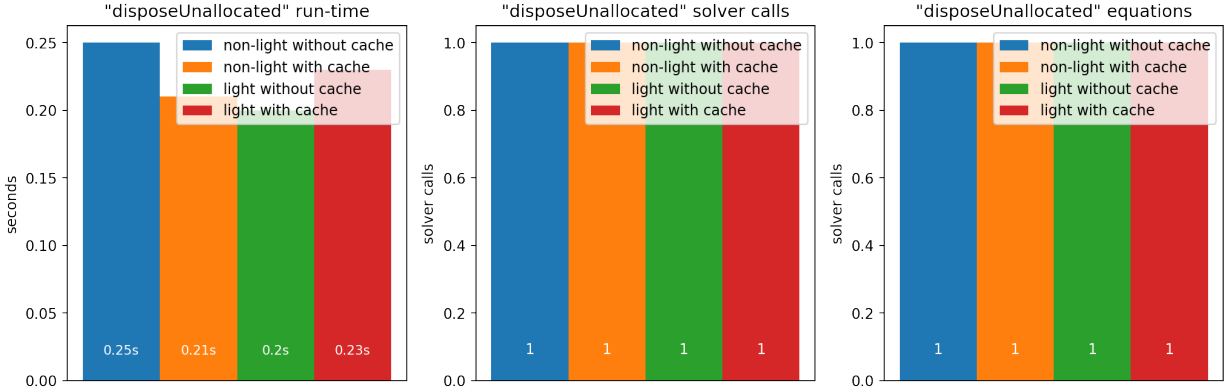

Fig. 5.3. Benchmark of "disposeUnallocated" example

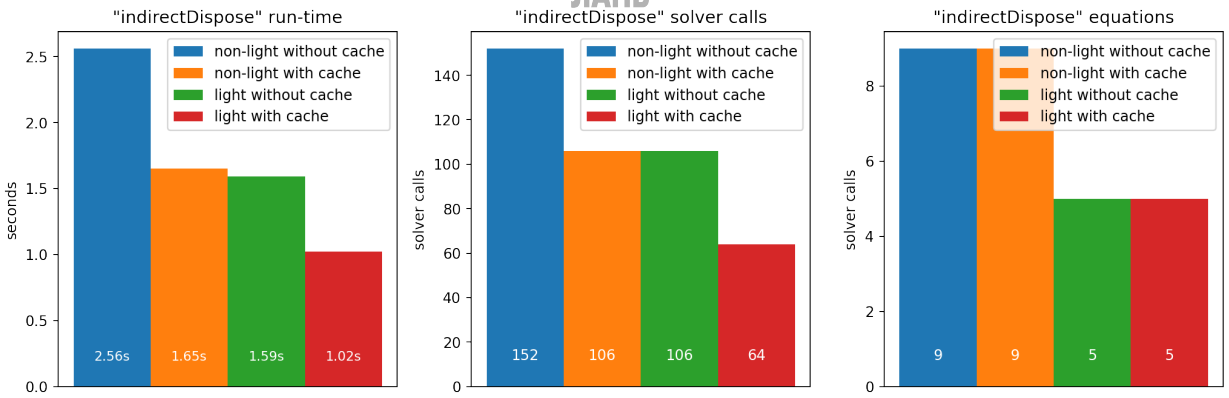

Fig. 5.4. Benchmark of "indirectDispose" example

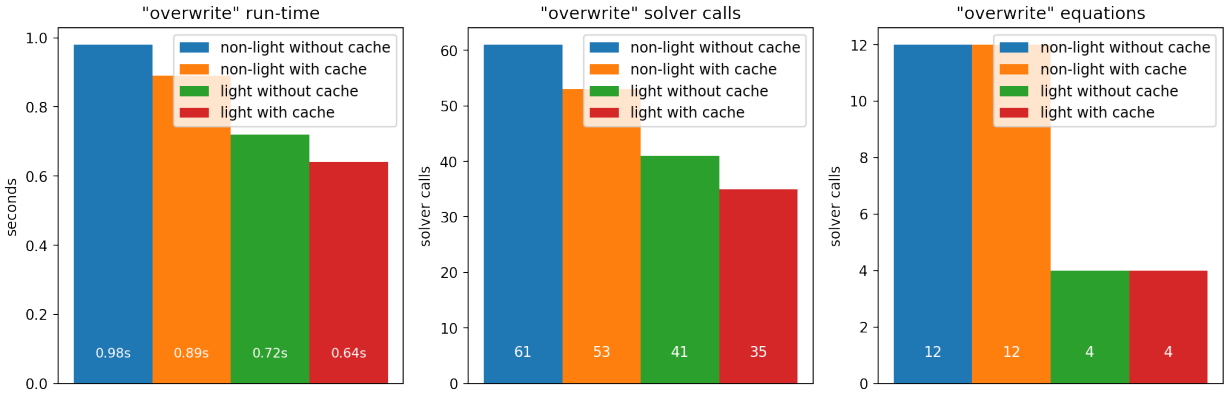

Fig. 5.5. Benchmark of "overwrite" example

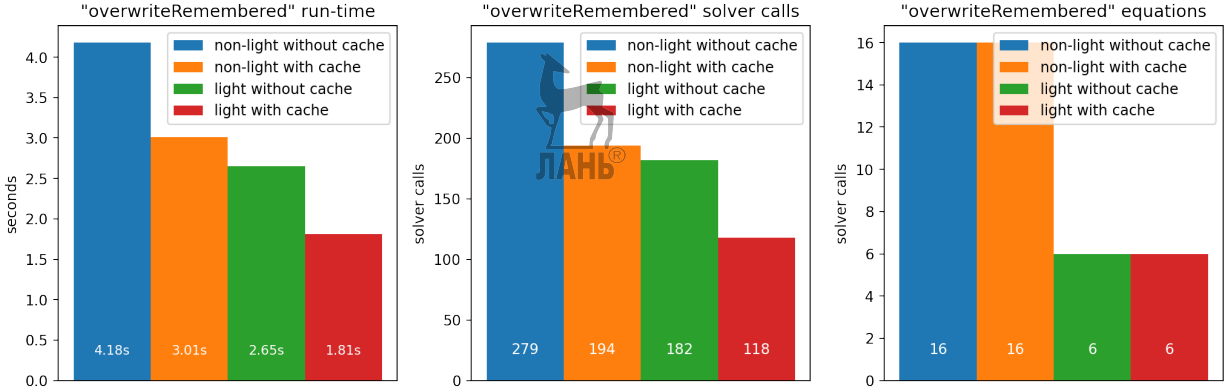

Fig. 5.6. Benchmark of "overwriteRemembered" example

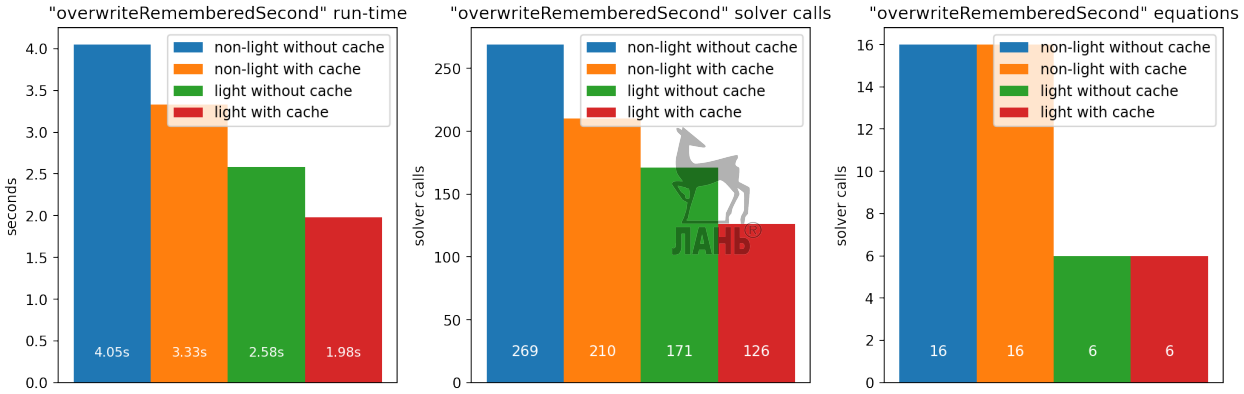

Fig. 5.7. Benchmark of "overwriteRememberedSecond" example

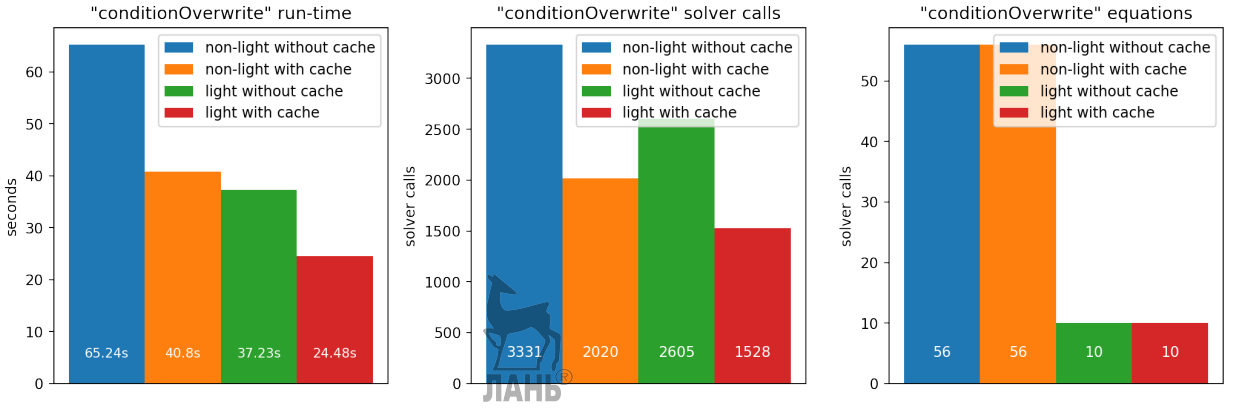

Fig. 5.8. Benchmark of "conditionOverwrite" example

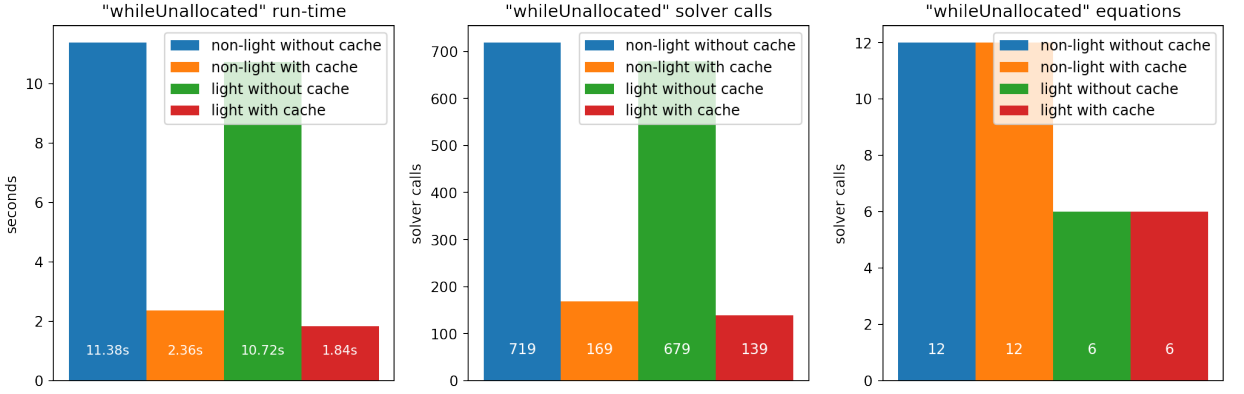

Fig. 5.9. Benchmark of "whileUnallocated" example

5.3.2 Validator core

The validator core is processing the program in two passes — so it makes sense to talk about complexities for these two passes separately.

The "pre"-pass' complexity is rather simple — the validator core just linearly processes each expression. Thus, the "pre"-pass' complexity is $\mathcal{O}(T_e)$, where T_e is the amount of input tokens that belong to expressions (input tokens are all terminals and non-terminals from the PEG defined in Appendix B).

The "main"-pass' complexity, however, is less trivial, but as all processing is happening within the *aft* transformer, it can be further broken down into several cases:

- **Case 1**. Nothing memory-related is happening.
  As $aft(D) = D$ in this case, it takes only $\mathcal{O}(1)$ to process this.

- **Case 2**. A single non-compound statement.
  Complexity here can also be broken down into two parts: in terms of simple operations and in terms of solver calls. For simple operations — the "heaviest" part is only $\mathcal{O}(N^2)$ where N is the amount of address expressions in the program. As for the solver calls-based complexity — the "heaviest" part of each are the implied equation calculations (the ones where new equations are added into the binary formula). These are calculated for each pair of expressions in the program, thus $\mathcal{O}(N^2)$, where N — amount of expressions in the program.

- **Case 3**. Compound statements.
  For compound statements (i.e. "if" and "while") it makes sense to calculate complexity in terms of "how many times does the validator pass through the statement's body". "If" statements only calculate "then" and "else"

branches, so at most twice the amount of computations, giving $\mathcal{O}(1)$. "While" statements calculate the body "until nothing changes", with operations only adding new expressions, which gives an upper bound equal to $\mathcal{O}(N)$ "while" loop body computations.

Taking worst cases of each statement gives a total complexity of $\mathcal{O}(N^2 S)$, where N is the amount of address expressions in the program and S is the amount of statements in the program.

5.3.3 Overall complexity

Combining all three parts — parser's complexity of $\mathcal{O}(2^T)$ operations, validator core's complexity of $\mathcal{O}(N^2 S)$ solver calls — gives an overall complexity of $\mathcal{O}(N^2 \cdot S)$ in terms of solver calls.

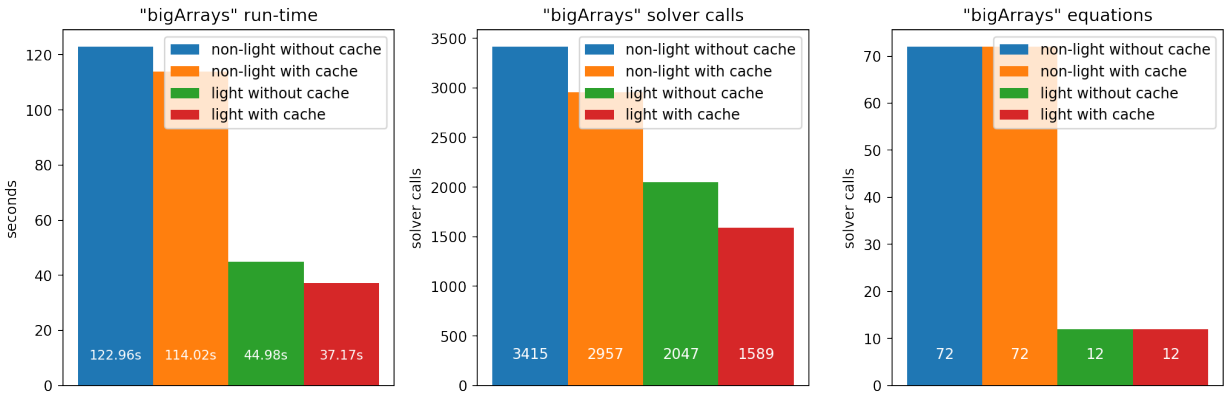

Fig. 5.10. Benchmark of “bigArrays” example

Chapter 6

Conclusion

Memory analysis is an extensively and intensively researched topic that is a part of a wider software verification problem. Complex static analysis is becoming more and more relevant today as software itself is becoming more and more complex — city management, fintech, blockchain, etc. Software static analysis is a complex and complicated problem in and of itself and consists of many sub-problems, one of which is aliasing analysis.

In this Thesis, the existing literature on the aliasing analysis problem was analyzed, and memory analysis in general; definitions for an alias calculus variant with support for pointer arithmetic were presented with slightly changed definitions and additional examples to aid readers; a validation tool based on the presented calculi was prototyped with aid of the z3 solver; and evaluated its performance on a set of program examples.

The alias calculus is almost exactly as described in [1]. Here a light version of Alias Calculus was suggested for better efficiency of memory leak detection, while its soundness and relation to the original Calculus need further theoretical studies. The only changes for the language, *MoRe* — additional

syntax extensions to simplify parsing and provide a way to specify additional address expressions from an outer context outside of the considered program.

The implementation includes a binary equation formula solver powered by z3 written in python, the validator itself written in TypeScript, and the *MoRe* parser generated from a PEG ruleset. TypeScript allows the validator to run asynchronously, and improve performance in that way.

6.1 Limitations

Testing has proven perspectives of the Calculi for static analysis, while scalability and utility for industrial code analysis need further research.

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.

One other possible improvement is scalability of the theory — the current calculus is *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.

6.2 Contribution Summary and future research topics

- A light version of Alias Calculus was suggested for better efficiency of memory leak detection, while its soundness and relation to the original Calculus need further theoretical studies.
- Both light and the original versions were implemented (prototyped) using Z3 SMT-solver to solve systems of memory constraints.
- Both implementations had been tested for detecting memory leaks in small (up to 20 loc) specially-designed examples.
- Testing has proven perspectives of the Calculi for static analysis, while scalability and utility for industrial code analysis need further research.

Bibliography cited

[1] N. V. Shilov, A. Satekbayeva, and A. P. Vorontsov, "Alias calculus for a simple imperative language with decidable pointer arithmetic," *Bulletin of the Novosibirsk Computing Center*, vol. 37, pp. 131–148, 2014.

[2] B. Meyer, "Steps towards a theory and calculus of aliasing," *arXiv preprint arXiv:1001.1610*, 2010.

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

[4] V. Rivera and B. Meyer, "Autoalias: Automatic variable-precision alias analysis for object-oriented programs," *SN Computer Science*, vol. 1, no. 1, p. 12, 2020.

[5] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, IEEE, 2002, pp. 55–74.

[6] J. Berdine, C. Calcagno, and P. W. O'hearn, "Smallfoot: Modular automatic assertion checking with separation logic," in *International Symposium on Formal Methods for Components and Objects*, Springer, 2005, pp. 115–137.

[7] J. A. Navarro Pérez and A. Rybalchenko, "Separation logic+ superposition calculus= heap theorem prover," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 556–566.

[8] J. Berdine, C. Calcagno, and P. W. O'hearn, "Symbolic execution with separation logic," in *Asian Symposium on Programming Languages and Systems*, Springer, 2005, pp. 52–68.

[9] А. Прокопенко and А. Тормасов, "Дедуктивный метод анализа логических гонок с использованием сепарационной логики," *Труды Московского физико-технического института*, vol. 2, no. 3, 2010.

[10] R. Krebbers, R. Jung, A. Bizjak, J.-H. Jourdan, D. Dreyer, and L. Birkedal, "The essence of higher-order concurrent separation logic," in *European Symposium on Programming*, Springer, 2017, pp. 696–723.

[11] M. Parkinson, "The next 700 separation logics," in *Verified Software: Theories, Tools, Experiments*, G. T. Leavens, P. O'Hearn, and S. K. Rajamani, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 169–182, ISBN: 978-3-642-15057-9.

[12] N. Heintze and O. Tardieu, "Ultra-fast aliasing analysis using cla: A million lines of c code in a second," *ACM SIGPLAN Notices*, vol. 36, no. 5, pp. 254–263, 2001.

[13] S. Debray, R. Muth, and M. Weippert, "Alias analysis of executable code," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1998, pp. 12–24.

[14] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, Citeseer, 1994.

[15] A. A. Mathiasen and A. Pavlogiannis, "The fine-grained and parallel complexity of andersen's pointer analysis," *arXiv preprint arXiv:2006.01491*, 2020.

[16] J. Research. (2021). "The fine-grained and parallel complexity of andersen's pointer analysis," Youtube, [Online]. Available: https://youtu.be/SJ1UyDUr3TE.

[17] J. Zhang, "Symbolic execution of program paths involving pointer structure variables," in *Fourth International Conference onQuality Software, 2004. QSIC 2004. Proceedings.*, IEEE, 2004, pp. 87–92.

[18] Z. Xu, T. Kremenek, and J. Zhang, "A memory model for static analysis of c programs," in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, Springer, 2010, pp. 535–548.

Appendix A

Program Examples

This appendix chapter provides the source code for the programs mentioned in Chapter 5 (Evaluation).

A.1 Empty

Listing A.1. "Empty" program without any warnings

```
program begin
  var x = 0
  var y = 0
  x := cons(1, 3)
  y := cons(2)
  dispose x
  dispose y
end
```

A program without any warnings that is completely correct is displayed in Listing A.1.

A.2 Double Dispose

Listing A.2. Running dispose twice on the same variable

```
program begin
  var x = 0
  x := cons(1, 3)
  dispose x
  dispose x
end
```

A program with a "double-dispose" - two dispose statements performed on the same variable - is displayed in Listing A.2.

This program should produce a warning with the following text: "x does not point onto any allocated memory on statement dispose x line 4".

A.3 Dispose Unallocated

Listing A.3. Running dispose on a variable that has not been allocated

```
program begin
  var x = 0
  dispose x
end
```

A program that disposes an unallocated address is displayed in Listing A.3.

This program should produce a warning with the following text: "x does not point onto any allocated memory on statement dispose x line 2".

A.4 Indirect Dispose

Listing A.4. Disposing a variable that holds reference to a piece of memory that was allocated for another variable

```
program begin
  var x = 0
  var y = 0
  x := cons(1, 2)
  y := x
  dispose y
  dispose x
end
```

A program that disposes a piece of allocated memory "indirectly" (i.e., through another variable that holds a reference to memory) is displayed in Listing A.4. In this Listing, there is a small array being allocated and its address then stored in x. Then, x is additionally assigned to y. Finally, the program first disposes of y and then disposes of x.

This program should result in only one warning from the validator: "x does not point onto any allocated memory on statement dispose x line 6", because the array was already taken care of on line 6 by the *dispose* statement for y.

A.5 Overwrite

Listing A.5. Overwriting a variable that is the only reference to some memory

```
program begin
```

```
    var x = 0
    x := cons(1, 3)
    x := 3
end
```

A program that overwrites a variable that holds the only reference to an allocated piece of memory is displayed in Listing A.5.

This program should produce two following warnings:

- "Memory leak - x is no longer reachable after assignment on statement x := 3 line 3".
- "Memory leak - x + 1 is no longer reachable after assignment on statement x := 3 line 3".

Two warnings are produced as a result of x pointing to a two-element array, and the array elements being accounted for using separate addresses x and $x + 1$ inside the calculus.

A.6 Overwrite Remembered

Listing A.6. Overwriting a variable that is the not only reference to some memory

```
@assume address-expression y, y + 1
program begin
    var x = 0
    var y = 0
    x := cons(1, 3)
    y := x
```

```
    x := 3
  end
```

A program that overwrites a variable that holds not the only reference to an allocated piece of memory is displayed in Listing A.6. As opposed to the program in Listing A.5, this program is storing the array reference into y before overwriting x.

This program should not result in any warnings from the validator.

A.7 Overwrite Remembered Second

Listing A.7. Overwriting a variable that is the not only reference to some memory, but the stored reference is partial

```
@assume address-expression y, y + 1
program begin
  var x = 0
  var y = 0
  x := cons(1, 3)
  y := x + 1
  x := 3
end
```


A program that overwrites a variable that holds the only reference to an allocated memory, but the target memory is partially "remembered" in another variable, is displayed in Listing A.7. As opposed to the program in Listing A.6, this program is storing only the reference to the second array element.

This program should result in only one warning from the validator: "Memory leak - x is no longer reachable after assignment on statement x := 3 line 5".

The address $x + 1$ is not a part of the memory leak, because $y = x + 1$ is still provable and indicates that we can still reach memory located at $x + 1$ from the program.

A.8 Condition Overwrite

Listing A.8. Overwriting a part of an array conditionally

```
@assume address-expression y, y + 1, y + 2, y + 3
program begin
  var x = 0
  var y = 0
  x := cons(1, 2, 3, 4)
  if condition then begin
    y := x + 1
  end else begin
    y := x + 2
  end
  x := 3
end
```

A program that conditionally (using an if statement) overwrites a piece of the array is displayed in Listing A.8.

This program should produce two following warnings:

- "Memory leak - x is no longer reachable after assignment on statement x := 3 line 5".

- "Memory leak - x + 1 is no longer reachable after assignment on statement x := 3 line 5".

Even though there is a branch that correctly "remembers" $x + 1$, the calculus is flow-insensitive, meaning that results from both branches of the if statement are merged.

A.9 While Unallocated

Listing A.9. A while loop that might have 0 loops executed, because the calculus is flow-insensitive

```
program begin
  var x = 0
  while condition do begin
    x := cons(1, 2)
  end
  dispose x
end
```

A program that allocates new memory in a while loop is displayed in Listing A.9. The calculus is flow-insensitive, so the loop's condition is replaced with the literal word condition. The allocated memory is then disposed of after the loop, but the loop's body might run 0 times.

That is why this program should result in the following warning from the validator: "x does not point onto any allocated memory on statement dispose x line 3".

A.10 Big Arrays

Listing A.10. A correct program without any warnings that has "big" arrays

```
program begin
  var x = 0
  var y = 0
  x := cons(1, 2, 3, 4, 5, 6)
  y := cons(1, 2, 3, 4, 5, 6)
  dispose x
  dispose y
end
```

A simple correct program that just has big arrays is displayed in Listing A.10. This program is in the test sample set because it is a more representative sample to test performance on — having big arrays decreases significance of the start-up time.

Appendix B

Extended MoRe syntax as PEG

The full PEG (tspeg) for the MoRe syntax extension is provided in Listing B.1

Listing B.1. PEG syntax definitions for the extended MoRe version

```
start := program=moReProgram $

moReMetaBlock := firstMeta=moReMeta?
  metas={sep content=moReMeta}*
moReMeta := moReAssume
moReAssume := '@assume' sep 'address-expression'
  sep expressions=moReExpressionList

moReProgram := wsep meta=moReMetaBlock wsep
  'program' sep block=moReProgramBlock wsep
moReProgramBlock := 'begin' sep
```

```
  firstStatement=moReStatement
  statements={sep content=moReStatement}*
  sep 'end'
moReStatement := !'end' statement={moReIf |
  moReWhile | moReSkip | moReDeclare | moReDispose |
    moReAlloc | moReStore | moReLoad | moReAssign}

moReSkip := skip='skip'

moReDeclare := 'var' sep variable=moReVariable
  wsep '=' wsep expression=moReExpression

moReAssign := variable=moReVariable wsep ':='
  wsep expression=moReExpression

moReLoad := variable=moReVariable wsep ':=' wsep
  '\[' wsep address=moReVariable wsep '\]'

moReStore := '\[' wsep address=moReVariable wsep
  '\]' wsep ':=' wsep variable=moReVariable

moReDispose := 'dispose' wsep variable=moReVariable

moReAlloc := variable=moReVariable wsep ':=' wsep
  'cons' wsep '\(' wsep
    expressions=moReExpressionList wsep '\)'
```

```
moReIf := 'if' sep condition=moReCondition sep
  'then' sep then=moReProgramBlock
  elseBlock={sep 'else' sep else_=moReProgramBlock}?

moReWhile := 'while' sep condition=moReCondition
  sep 'do' sep program=moReProgramBlock

moReCondition := 'condition'

moReExpressionList :=
  expressions={content=moReExpression wsep ',' wsep}*
  lastExpression=moReExpression

moReExpression := moReExprSum | moReExprDiff |
  moReVariable | moReLiteral | moReExprParen |
  moReExprNeg
moReExprSum := left=moReExpression wsep '\+'
  wsep right=moReExpression
moReExprDiff := left=moReExpression wsep '-'
  wsep right=moReExpression
moReExprNeg := '-' wsep expr=moReExpression
moReExprParen := '\(' wsep expr=moReExpression wsep '\)'

keywords := 'while' | 'if' | 'begin' | 'end'
```

```
moReLiteral := literal='[0-9]+'
moReVariable := variable='[a-zA-Z_][a-zA-Z0-9_]*'

wsep := '[\s]*'
sep := '[\s]+'
```