Combining Range and Inequality Information for Pointer Disambiguation

Maroua Maalej, Vitor Paisante, Fernando Magno Quintão Pereira, Laure Gonnord

▶ **To cite this version:**

Maroua Maalej, Vitor Paisante, Fernando Magno Quintão Pereira, Laure Gonnord. Combining Range and Inequality Information for Pointer Disambiguation. [Research Report] RR-9009, ENS Lyon; CNRS; INRIA. 2016. hal-01429777v1

HAL Id: hal-01429777
https://inria.hal.science/hal-01429777v1

Submitted on 9 Jan 2017 (v1), last revised 11 Jun 2017 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Combining Range and Inequality Information for Pointer Disambiguation

Maroua Maalej, Vitor Paisante , Fernando Magno Quintão Pereira,
Laure Gonnord

ISSN 0249-6399 ISRN INRIA/RR--9009--FR+ENG

RESEARCH REPORT
N° 9009
December 2016
Project-Team ROMA

Combining Range and Inequality Information for Pointer Disambiguation

Maroua Maalej[*,†], Vitor Paisante [‡], Fernando Magno Quintão Pereira[‡], Laure Gonnord [§]

Project-Team ROMA

Research Report n° 9009 — December 2016 — 31 pages

Abstract: Pentagons is an abstract domain invented by Logozzo and Fähndrich to validate array accesses in low-level programming languages. This algebraic structure provides a cheap "less-than check", which builds a partial order between the integer variables used in a program. In this paper, we show how we have used the ideas available in Pentagons to design and implement a novel alias analysis. This new algorithm lets us disambiguate pointers with offsets, so common in C-style pointer arithmetics, in a precise and efficient way. Together with this new abstract domain we describe several implementation decisions that lets us produce a practical pointer disambiguation algorithm on top of the LLVM compiler. Our alias analysis is able to handle programs as large as SPEC's gcc in a few minutes. Furthermore, we have been able to improve the percentage of pairs of pointers disambiguated, when compared to LLVM's built-in analyses, by a four-fold factor in some benchmarks.

Key-words: Points-to Analysis Pentagons Less-Than Check Abstract Interpretation Compiler Construction Static Analysis

* This work was supported by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR).

† University of Lyon, France & LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA)
‡ Department of Computer Science, UFMG, Brazil
§ Univ. Lyon1, France & LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA)

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Combinaison des Ranges et d'Inégalités Strictes pour la Désambiguisation de Pointeurs

Résumé : Les Pentagons est un domaine abstrait inventé par Logozzo et Fähndrich pour la validation des accès tableaux dans les langages de programmation à bas niveau. Cette structure algébrique fournit une relation d'ordre partiel entre les variables entières du programme. Il s'agit d'une relation "plus petit que". Dans ce papier, nous montrons comment utiliser l'idée des Pentagons pour concevoir et implémenter une nouvelle analyse d'alias. Ce nouvel algorithme nous permet de désambiguiser, d'une manière précide et efficace, des pointeurs avec des offsets fréquemment utlisés dans l'arithmétique des pointeurs en C. Nous décrivons, en plus de ce nouveau domaine abstrait, plusieurs détails d'implémentation qui nous ont permis de produire un algorithme d'analyse de pointeurs dans LLVM. Notre analyse est capable de traiter des programmes aussi larges que gcc de SPEC en quelques minutes seulement. Par ailleurs, pour certains benchmarks, nous avons réussi à améliorer par facteur de quatre le pourcentage de paires de pointeurs désambiguisés, étant comparé aux analyses existantes dans LLVM.

Mots-clés : Analyse Points-to Pentagons Interprétation Abtraite "plus petit que" Analyses statiques Compilateurs

1 Introduction

Pentagons is an abstract domain invented by Logozzo and Fähndrich to infer symbolic bounds to the integer variables used in programs [20, 21]. This abstract domain is formed by the combination of two lattices. The first lattice is the *integer interval domain* [9], which maps integer variables to ranges $[l, u]$ of numeric lower (l) and upper (u) bounds. The second lattice is the *strict upper bound*, which maps each variable v to a set $L_<$ of other variables, so that if $u \in L_<(v)$, at a given program point `p`, then $u < v$ at `p`.

Since their debut [20], Pentagons have been used in several different ways. For instance, Logozzo and Fähndrich have employed this domain to eliminate array bound checks in strongly typed programming languages, and to ensure absence of division by zero or integer overflows in programs. Moreover, Nazaré *et al.* [23] have used Pentagons to reduce the overhead imposed by AddressSanitizer [31] to guard C against out-of-bounds memory accesses. The appeal of pentagons comes from two facts. First, this abstract domain can be computed efficiently – in quadratic time on the number of program variables. Second, as an enabler of compiler optimizations, Pentagons have been proven to be substantially more effective than other forms of abstract interpretation of similar runtime [13].

In this paper, we present a novel use of Pentagons: the disambiguation of pointers in languages with pointer arithmetics, such as C, C++ and assembly languages. We show that Pentagon's key idea: a cheap and relatively precise "less-than" check, is effective to distinguish memory locations at compilation time. However, far from simply applying Pentagons onto a different problem, we demonstrate that it is necessary to adapt this abstract domain in non-trivial ways. First, it is necessary to separate pointers from integer offsets in the resolution of the constraints that build less-than relations between variables. Therefore, the constraints that we produce to analyze programs are very different than in the original description of this lattice.

As further contributions, while in the process of designing our new pointer disambiguation algorithm, we realized that several implementation details of the less-than check could be improved without changing its key theoretical elements. We describe these observations in Section 6. Among the changes in the original implementation of Pentagons, we show how to use them in a sparse analysis, which is more economic in terms of space, and runs in less time. A sparse analysis associates information to variable names. By information we mean the produces of the static analysis, e.g., less-than relations, ranges, etc. In contrast, a dense analysis associates information with pairs formed by variables and program points. Therefore, our implementation maps $O(|V|)$ points to less-than relations – each point representing a single variable name. On the other hand, the original description of pentagons maps $O(|V| \times |P|)$ entries, formed by variable names and program points, to less-than relations.

Additionally, we opted for decoupling the range analysis from the less-than analysis that forms the Pentagons' abstract domain. This choice was motivated more by engineering pragmatism, than by a need for precision or efficiency: it lets us combine different implementations of these static analyses more modularly. Our new algorithm also handles a few programming constructions that the original description of Pentagons did not touch, such as operations involving two variables on the right side, e.g., $v = v_1 + v_2$. We illustrate these differences with examples in the rest of this paper, and further discuss them in Section 6.

All this infra-structure gives us the means to perform different pointer disambiguation checks. As we explain in Section 4.5, we use three different checks to verify if two pointers might alias. All these checks emerge naturally from the data-structures that we build in the effort to find less-than relations between variables. In a nutshell, we can prove that two pointers, p_1 and p_2, do not overlap if either: (i) they come from different memory allocation sites; (ii) $p_1 < p_2$ or $p_2 < p_1$; or (iii) the range of memory regions that these pointers might dereference do not overlap.

```
1  void ins_sort(int* v, int N) {
2    int i, j;
3    for (i = 0; i < N − 1; i++) {
4      for (j = i + 1; j < N; j++) {
5        if (v[i] > v[j]) {
6          int tmp = v[i];
7          v[i] = v[j];
8          v[j] = tmp;
9        }
10     }
11   }
12 }
```

(a) Insertion sort

```
1  void partition(int *v, int N) {
2    int i, j, p, tmp;
3    p = v[N/2];
4    for (i = 0, j = N − 1;; i++, j−−) {
5      while (v[i] < p) i++;
6      while (p < v[j]) j−−;
7      if (i >= j)
8        break;
9      tmp = v[i];
10     v[i] = v[j];
11     v[j] = tmp;
12   }
13 }
```

(b) Partition

Figure 1: Two programs that challenge traditional pointer disambiguation analyses.

In order to validate our ideas, we have implemented them in the LLVM compilation infrastructure [19]. Our new alias analysis increases the ability of this compiler to disambiguate pointers in non-trivial ways. As an example, we increase by almost 2.5x the number of pairs of pointers that we distinguish in SPEC's `hmmer` and `bzip2`, and almost 3.5x this quantity for SPEC's `gcc`. Furthermore, contrary to several algebraic pointer disambiguation techniques, our implementation scales up to programs with millions of assembly instructions. We emphasize that these numbers have not been obtained by comparing our implementation against a straw man: LLVM is an industrial-quality compiler, which provides its users with a rich collection of pointer analyses already heavily tested.

2 Overview

To motivate the need for a new points-to analysis we shall use two well-known algorithms, whose implementation in C appears in Figure 1. The figure displays the C implementation of two sorting routines that make heavy use of pointers. In both cases, we know that memory positions `v[i]` and `v[j]` can never alias within the same iteration of the loop. However, traditional points-to analyses cannot prove this fact. Typical implementations of these analyses, built on top of the work of Andersen [4] or Steensgaard [32], can distinguish pointers that dereference different memory blocks. However, they do not say much about references to the same array.

There exist points-to analyses designed specifically to deal with pointer arithmetics [5, 36, 25, 28]. Nevertheless, none of them works satisfactorily for the two examples seen in Figure 1. The reason for this ineffectiveness lays on the fact that these analyses use range intervals to disambiguate pointers. In our examples, the ranges of integer variables `i` and `j` overlap. Therefore, any conservative range analysis, à la Cousot [9], once applied on Figure 1a, will conclude that `i` exists on the interval $[0, \texttt{N} - 2]$, and that `j` exists on the interval $[1, \texttt{N} - 1]$. Because these two intervals have non-empty intersections, points-to analyses based on the interval lattice will not be able to disambiguate the memory accesses at lines 6-8 of Figure 1a.

The technique that we introduce in this paper can disambiguate every use of `v[i]` and `v[j]` in both examples. Key to this success is the observation that $\texttt{i} < \texttt{j}$ at every program point where we have an access to `v`. We conclude that $\texttt{i} < \texttt{j}$ by means of a "less-than check". A less-than

check is a relation between two variables, e.g., v_1 and v_2, that is true whenever we can prove – statically – that one holds a value less than the value stored into the other. In Figure 1a, we know that $\texttt{i} < \texttt{j}$ because of the way that `j` is initialized, within the for statement at line 4. In Figure 1b, we know that $\texttt{i} < \texttt{j}$ due to the conditional check at line 7.

There are many ways to build a less-than relation between program variables. In this paper, we have decided to use the Pentagon lattice to achieve this end. However, a straightforward application of Pentagons, as originally defined by Logozzo and Fähndrich, would not give us the facts that we need. The syntax `v[i]` denotes a memory address given by `v + i`. Pentagons combine range analysis with a less-than check. The range of `v`, a pointer, is $[0, +\infty]$. Therefore, this will be also the range of `v + i` and `v + j`. In other words, even though we know that $\texttt{i} < \texttt{j}$, we cannot conclude that $\texttt{v} + \texttt{i} < \texttt{v} + \texttt{j}$. A way to solve this problem is to separate the analysis of pointers from the analysis of integers. In Section 4, we shall describe an abstract interpreter that is aware of this distinction.

```
1  void ins_sort_goal(int* v, int N) {
2    int i, j,tmp_i,tmp_j,tmp;
3    for (i = 0; i < N − 1; i++) {
4      tmp_i=v[i];
5      for (j = i + 1; j < N; j++) {
6        tmp_j=v[j];
7        if (tmp_i > tmp_j) {
8          tmp = tmp_i;
9          tmp_i = tmp_j;
10         v[j] = tmp;
11       }
12     }
13     v[i]=tmp_i;
14   }
15 }
```

Figure 2: Implementation of insertion sort, seen in Figure 1a, after optimization via scalar replacement.

A more precise alias analysis brings many advantages to compilers. One of such benefits is optimizations: the extra precision gives compilers information to carry out more extensive transformations in programs. Figure 2 illustrates this benefit. The figure shows the result of applying Surendran's [34] inter-iteration scalar replacement on the insertion sort algorithm seen in Figure 1. Scalar replacement is a compiler optimization that consists in moving memory locations to registers as much as possible. This optimization tends to speedup programs because it removes memory accesses from their source code. In this example, if we can prove that `v[i]` and `v[j]` do not reference overlapping memory locations, we can move these locations to temporary variables. For instance, we have loaded location `v[j]` into `tmp_j` at line 6. We update the value of `v[i]` at line 13.

3 Preliminary Definitions

The technique that we introduce in this paper lets us compare two pointers, p_1 and p_2, and show that they are different, whenever we can prove a *core property*, which we define below:

Definition 3.1 (The Strict Inequality Property) *We say that two pointers, p_1 and p_2 are strictly different whenever we can prove that either $p_1 < p_2$, or $p_2 < p_1$, at every program point where these two pointers are simultaneously alive.*

Definition 3.1 touches several concepts pertaining to the argot of compilation theory. A *program point* is a region between two consecutive instructions in assembly code. We say that a variable v is *alive* at a program point x if, and only if, there exist a path from x to another program point where v is used, and this path does not cross any redefinition of v. Computing live ranges of variables is a classical dataflow analysis [24, Sec-2.1.4].

As the reader can infer from these definitions, in the rest of this paper we shall work at a low-level representation of programs. Thus, we shall abandon the high-level C notation seen in

Integer constants	::=	$\{c_0, c_1, \ldots\}$
Integer variables	::=	$\{i_0, i_1, \ldots\}$
Pointer variables	::=	$\{p_0, p_1, \ldots\}$
Instructions (I)	::=	
– Allocate memory	|	$p_0 = \text{malloc}(i_0)$
– Free memory	|	$p_0 = \text{free}(p_1)$
– Pointer plus int	|	$p_0 = p_1 + i_0$
– Pointer plus const	|	$p_0 = p_1 + c_0$
– Bound intersection	|	$p_0 = p_1 \cap [l, u]$
– Load into pointer	|	$p_0 = *p_1$
– Store from pointer	|	$*p_0 = p_1$
– ϕ-function	|	$p_0 = \phi(p_1 : \ell_1, p_2 : \ell_2)$
– Branch if not zero	|	$\text{bnz}(v, \ell)$
– Unconditional jump	|	$\text{jump}(\ell)$

Figure 3: The syntax of our language of pointers.

our early examples, in favour of an assembly-like language, whose syntax is defined in Figure 3 and whose semantics will be informally discussed in Section 3.1. Throughout the paper, we shall return to the high-level notation, because it gives us the opportunity to write more readable examples.

3.1 Sparse Analysis

The implementation that we discuss in this paper is a *sparse analysis*. According to Tavares *et al.* [35], a data-flow analysis is sparse if the abstract state associated with a variable is constant throughout the entire live range of that variable. Notice that this information might not be true for every program. As an example, consider the program in Figure 1b. We know that `i < j` within lines 8-10. However, this fact is false at line 7. To make a data-flow analysis sparse, we resort to *live range splitting*.

The live range of a variable v is the set of program points where v is alive. To split the live range of a variable v, at a program point x, we insert a new instruction $v' = v$ at x, where v' is a fresh name. After that, we rename every use of v that is *dominated* by x to v'. We say that a program point x dominates a program point y if every path from the program's entry point to y must go across x. Live range splitting is a well-known technique to sparcify data-flow analyses. There are several program representations that naturally implement this trick. The most celebrated among these representations is the Static Single Information form [2, 12] (SSA), which ensures that each variable has at most one definition point.

The SSA format is not enough to ensure sparcity to the analysis that we propose in this paper. Therefore, we use a different flavour of this program representation: the *Extended Static Single Assignment form* [8] (e-SSA). The e-SSA form can be computed cheaply from an SSA form program. This extension introduces σ-functions to redefine program variables at split points such as branches and switches. These special instructions, the σ-functions, rename variables at the out-edges of conditional branches. They ensure that each outcome of a conditional is associated with a distinct name. In our case, this property lets us bind to each variable name the information that we learn about it as the result of comparisons performed in conditional statements.

Example 3.2 *Figure 4 shows the* Control Flow Graph *(cfg) of the program seen in Figure 1a, in e-SSA form. Notice how σ-functions rename every variable used in a comparison.*

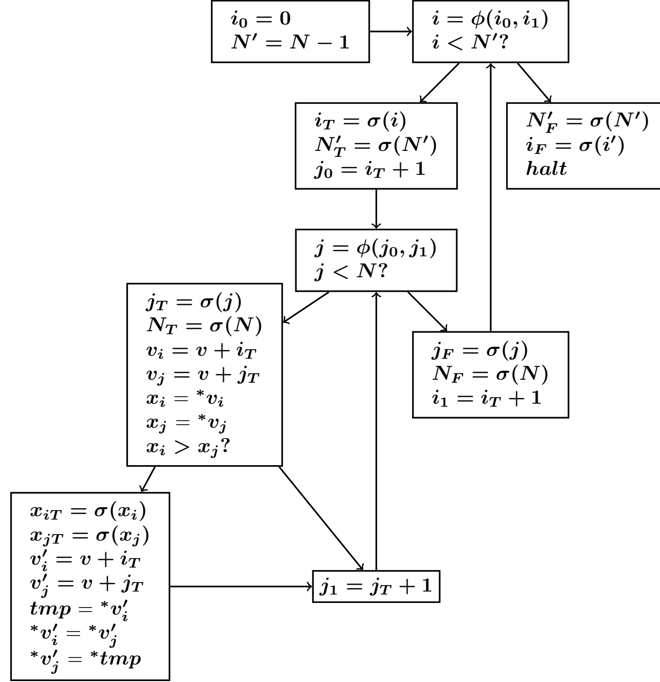


Figure 4: Control Flow Graph of program in Figure 1a. $i_T = \sigma(i)$ denotes the "true" version of i after the test $i < N$. $i = \phi(i_0, i_1)$ denotes the value of i inside the first loop, its value is i_0 if the flow comes from the first block, or i_1 if the flow comes from the returning edge $i_1 = i_T + 1$.

4 Pointer Disambiguation Based on Strict Inequalities

The alias analysis that we describe in this paper consists of the following five parts, which we describe in the rest of this section:

1. Find symbolic ranges for integer variables in the program. We explain this step in Section 4.1.
2. Group pointers related to the same memory location. We describe this step in Section 4.2
3. Collect constraints by traversing the program's control flow graph. Section 4.3 provides more details about this stage.
4. Solve the constraints produced in phase 3. Section 4.4 explains this phase of our approach.
5. Answer pointer disambiguation queries. We describe our method of answering queries in Section 4.5.

4.1 Range Analysis

A core component of our pointer disambiguation method is a *range analysis*, which can be symbolic or numeric. In this context, a symbol is any name in the program syntax that cannot be built as an expression of other names. Range analysis is not a contribution of this work. There are several different implementations already described in the compiler-related literature [3, 7, 23, 27, 29]. In this paper, we have adopted a non-relational range analysis [27] on the classic integer interval [9]. By non-relational, we mean that this range analysis associates variable names with ranges. Relational analyses, on the other hand, associate sets of variable names

with ranges. As an example, Miné's Octagons create relations between pairs of variables and range information [22]. Our approach is less precise than a relational approach, but has lower asymptotic complexity. In the sequel, we let $\mathrm{R}(\mathtt{v}) = [l, u]$ be the (symbolic) range of variable `v` computed by any range analysis.

Example 4.1 *Figure 4 shows the Control Flow Graph (cfg) of the insertion sort algorithm seen in Figure 1a. The ranges of integer variables given by Blume* et al. *[7] in this program are* $R(i_0) = [0,0], R(i_1) = [1, N-1], R(i) = [0, N-1], R(j_0) = [1, N-1], R(j_1) = [2, N], R(j) = [1, N]$. *The ranges of integer variables given by Rodrigues* et al. *[27] are* $R(i_0) = [0,0], R(i_1) = [1, +\infty], R(i) = [0, +\infty], R(j_0) = [1, +\infty], R(j_1) = [2, +\infty], R(j) = [1, +\infty]$.

We would like to emphasize that the range analysis that we shall use to obtain intervals for integer variables is *immaterial* for the correctness of our work. The only difference they make is in terms of precision and scalability. The more precise the range analysis that we use, the more precise the pointer analysis that we produce. However, precision has a cost in time. Given that range analysis is not a contribution of this work, we shall omit details related to its implementation, and refer the reader to the original discussion about our particular choice [27].

4.2 Grouping Pointers in Pointer Digraphs

Recalling Definition 3.1, we know that we can disambiguate two pointers, p_1 and p_2, whenever we can prove that $p_1 < p_2$. This relation is only meaningful for pointers that are offsets from the same *base pointer*[1]. A base pointer is a reference to the zeroth address of a memory block. As an example, a statement like "`u = malloc(4)`" will create a base pointer referenced by `u`. Formal arguments of functions, such as `v` in Figure 1a, are also base-pointers.

We call a *Pointer Dependance Digraph* (PDD) a directed acyclic graph (DAG) with origin at one or more base-pointers. All other nodes in this data-structure that are not base-pointers are variables that can be defined by an offset `o` from the base pointers. We let `o` be a symbol, such as a constant or the name of a variable, as defined in Section 4.1. For instance, the relation $\mathtt{p} = \mathtt{p}_0 + \mathtt{o}$ is represented, in the PDD, by an edge from the node `p` to the node $\mathtt{p}_0$ labeled by $\omega_{\mathtt{p},\mathtt{p}_0} = \mathtt{o}$. We denote this edge by $p \rightarrow_{\mathtt{o}} p_0$.

When analyzing a program, we build as many PDDs as the number of pointer definition sites in the program. Notice that this number is a *static* concept. A memory allocation site within a loop still give us only one allocation site, even if the loop iterates many times. Such a data-structure is constructed according to the rules in Figure 5, during a traversal of the program's control flow graph. Notice that we have a special treatment for ϕ-functions. A ϕ-function is a special instruction used in the SSA format to join the live ranges of variable that represent the same name in the original program, before the SSA transformation. We mark nodes created by ϕ-functions as dashed edges in the PDD. In this way, we ensure that a PDD has no cycles, because in a SSA-form program, the only way a variable can update itself is through a ϕ-function.

Example 4.2 *The rules seen in Figure 5, once applied onto the control flow graph given in Figure 4, gives us the PDD shown in Figure 6.*

4.3 Collecting constraints

During this phase, we collect a set C of constraints according to the rules in Figure 7. Recall that $\mathrm{R}(\mathtt{v})$ denotes the numeric range of variable `v` given by a pre-analysis. We let $R(\mathtt{v})_\downarrow$ and

[1] The ISO C Standard forbids relational comparisons between pointers, even with the same type, to different allocated objects [18, Sec-6.5.8p5].

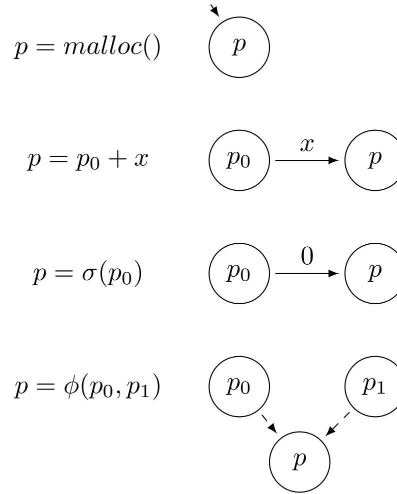


Figure 5: Rules to generate the pointer dependance digraph.

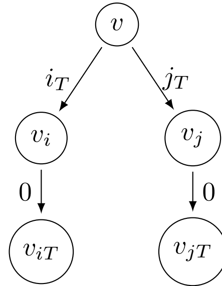


Figure 6: Pointer dependance graph of program in Figure 1a.

$R(\mathtt{v})_{\uparrow}$ be the lower and upper bounds of interval $\mathrm{R}(\mathtt{v})$, respectively. All the constraints that we produce in this stage follow the template $p_1 \mathscr{R} p_2$, where $\mathscr{R} \in \{<, \leq, =\}$. The rules in Figure 7 are syntax-directed. For instance, an assignment `p=q+v` let us derive the fact $\mathtt{p} > \mathtt{q}$, if the range analysis of Section 4.1 is capable of proving that `v` is always strictly greater than zero. The "=" equality models set inclusion, and is assymetric as we shall explain in details in Section 4.4.

Constraints are simple, and the reader will understand their meaning from their syntax; however, there are a few remarks that we would like to make. Firstly, the difference between *add* and *gep*[2] is the fact that the latter represents an addition on a pointer p plus an offset v_1, whereas the former represents general integer arithmetics. We distinguish both because the C standard forbids any arithmetic operation on pointers other than adding or subtracting an integer to it, e.g., $q_1 = p + v_1$ so that q_1 can't be compared to v_1. The construction of constraints for tests is more involved. The e-SSA form, which we have discussed in Section 3, provides us explicit new versions of variables, e.g.: $\mathtt{p_T}, \mathtt{q_T}, \mathtt{p_F}, \mathtt{q_F}$, after a test such as $\mathtt{p} < \mathtt{q}$. We use T as a subscript for the new variable name created at the *true* branch; F has similar use for the *false* branch. Thus, this conditional test for instance gives us two constraints: $\mathtt{p_T} < \mathtt{q_T}$ and $\mathtt{p_F} \geq \mathtt{q_F}$.

[2]The name *gep* is a short form for *get element pointer*, the expression used to define a new pointer address in the LLVM compiler.

$$\begin{array}{rcl}
\textit{Initialization} & \Rightarrow & \mathsf{C} = \emptyset \\
\textit{gep}: q_1 = p + v_1 & \Rightarrow & \begin{cases} \text{if } R(v_1)_\downarrow > 0 \ \mathsf{C} \cup = \{p < q_1\} \\ \text{if } R(v_1)_\downarrow \geqslant 0 \ \mathsf{C} \cup = \{p \leqslant q_1\} \\ \text{if } R(v_1)_\uparrow < 0 \ \mathsf{C} \cup = \{q_1 < p\} \\ \text{if } R(v_1)_\uparrow \leqslant 0 \ \mathsf{C} \cup = \{q_1 \leqslant p\} \end{cases} \\
\begin{array}{r} \textit{add}: v = v_1 + v_2 \\ \text{similar for } v \text{ and } v_2 \\ \text{with } v = v_2 + v_1 \end{array} & \Rightarrow & \begin{cases} \text{if } R(v_2)_\downarrow > 0 \ \mathsf{C} \cup = \{v_1 < v\} \\ \text{if } R(v_2)_\downarrow \geqslant 0 \ \mathsf{C} \cup = \{v_1 \leqslant v\} \\ \text{if } R(v_2)_\uparrow < 0 \ \mathsf{C} \cup = \{v < v_1\} \\ \text{if } R(v_2)_\uparrow \leqslant 0 \ \mathsf{C} \cup = \{v \leqslant v_1\} \end{cases} \\
\textit{sub}: v = v_1 - v_2 & \Rightarrow & \begin{cases} \text{if } R(v_2)_\downarrow > 0 \ \mathsf{C} \cup = \{v < v_1\} \\ \text{if } R(v_2)_\downarrow \geqslant 0 \ \mathsf{C} \cup = \{v \leqslant v_1\} \\ \text{if } R(v_2)_\uparrow < 0 \ \mathsf{C} \cup = \{v_1 < v\} \\ \text{if } R(v_2)_\uparrow \leqslant 0 \ \mathsf{C} \cup = \{v_1 \leqslant v\} \end{cases} \\
\textit{icmp}: p^1 \mathscr{R} p^2 & \Rightarrow & \begin{cases} \text{if } \mathscr{R} = \text{``}<\text{''}, \text{ then } \mathsf{C} \cup = \{p_1^{\mathrm{T}} < p_2^{\mathrm{T}}\} \cup \{p_2^{\mathrm{F}} \leqslant p_1^{\mathrm{F}}\} \\ \text{if } \mathscr{R} = \text{``}\leqslant\text{''}, \text{ then } \mathsf{C} \cup = \{p_1^{\mathrm{T}} \leqslant p_2^{\mathrm{T}}\} \cup \{p_2^{\mathrm{F}} < p_1^{\mathrm{F}}\} \\ \mathsf{C} \cup = \{p_1^{\mathrm{T}} = p^1\} \cup \{p_1^{\mathrm{F}} = p^1\} \cup \{p_2^{\mathrm{T}} = p^2\} \cup \{p_2^{\mathrm{F}} = p^2\} \end{cases} \\
\textit{union}: v = \phi(v_i) & \Rightarrow & \mathsf{C} \cup = \{v = \phi(v_i)\} \\
\begin{array}{r} v = c - v_1, c \in \mathbb{N} \\ q = *p \\ *q = p \end{array} & \Rightarrow & \textit{nothing}
\end{array}$$

Figure 7: Constraints produced for different statements in our language. The notation $\mathsf{C} \cup = S$ is a shorthand for $\mathsf{C} = C \cup S$. v_1 and v_2 are constants or scalar variables.

Example 4.3 *The rules in Figure 7, when applied onto the cfg seen in Figure 4, give us that* $C = \{N' < N,\ i = \phi(i_0, i_1),\ i_T < N'_T,\ N'_F \leq i_F;\ i_T < j_0,\ j = \phi(j_0, j_1),\ i_T = i,\ j_T = j,\ j_T < N_T,\ N_F \leq j_F,\ v \leq v_i,\ v < v_j,\ v \leq v'_i,\ v < v'_j,\ j_T < j_1,\ i_T < i_1,\ x_{jT} < x_{iT},\ x_{iF} \leq x_{jF}\}$

The Relation between PDDs and Constraints The purpose of the PDDs of Section 4.2 is to avoid comparing pointers that are not related by C-style arithmetics. They do not bear influence on the production of constraints; rather, they are used only in the queries that we shall describe in Section 4.4. This fact means that the grouping of pointers in digraphs is not an essential part of the idea of using strict inequalities to disambiguate pointers. However, PDDs are important from an operational standpoint: they provide a way of separating pointers that are not related by arithmetic operations; hence, are incomparable. Thus, the phases described in Section 4.2, and in this section are independent, and can be performed in any order.

4.4 Solving constraints

The objective of this phase is to obtain Pentagons-like abstract values for each variable or pointer of the target program. Henceforth, we shall call the set of program variables $\mathcal{V}$. The product of solving constraints is a relation LT. Given $\mathtt{v} \in \mathcal{V}, \mathsf{LT}(\mathtt{v})$ will keep track of all the variables that are *stricly* less than $\mathtt{v}$. In addition to LT, we build an auxiliary set GT. $\mathsf{GT}(\mathtt{v})$ keeps track of

$$strict\ less\ than : x < y \quad \Rightarrow \quad \begin{cases} \mathsf{LT}(y) \cup= \mathsf{LT}(x) \cup \{x\} \\ \mathsf{GT}(x) \cup= \mathsf{GT}(y) \cup \{y\} \end{cases}$$

$$less\ than : x \leqslant y \quad \Rightarrow \quad \begin{cases} \mathsf{LT}(y) \cup= \mathsf{LT}(x) \\ \mathsf{GT}(x) \cup= \mathsf{GT}(y) \end{cases}$$

$$non\ reflexive\ eq : x = y \quad \Rightarrow \quad \begin{cases} \mathsf{LT}(x) \cup= \mathsf{LT}(y) \\ \mathsf{GT}(x) \cup= \mathsf{GT}(y) \end{cases}$$

Figure 8: Rules to solve constraints.

all the variables that are *stricly* greater than `v`. Ordering relations are reflexive; hence, if $x < y$, then $y > x$. This fact means that we can build GT from LT, or vice-versa; hence, we could write our solver with only one of these relations. However, using just one of them would result in an unnecessarily heavy notation. Consequently, throughout the rest of this section we shall assume that the following two equations are always true:

$$\mathsf{GT}(v) = \bigcup \{v_i\},\ \forall v_i\ where\ v \in \mathsf{LT}(v_i)$$

$$\mathsf{LT}(v) = \bigcup \{v_i\},\ \forall v_i\ where\ v \in \mathsf{GT}(v_i)$$

Figures 8 and 10 contain the definition of our constraint solver. Constraints are solved via chaotic iterations: we compute for each program variable a sequence of abstract values until reaching a fix point. The non-reflexive equal is used to model relations that are known to be true before sigma nodes. Relational information that are true about variables before a conditional branch continue to hold also in the "then" and "else" paths that sprout from that branch. Example 4.4 illustrates this fact:

Example 4.4 *If the relation* ($\mathtt{x} < \mathtt{y}$) *holds before a conditional node that uses the predicate* ($\mathtt{x} < \mathtt{z}$?), *then these two relations are also true:* ($\mathtt{x_T} < \mathtt{y}$ *and* $\mathtt{x_F} < \mathtt{y}$).

Dealing with ϕ-functions *Join nodes* are program points that denote loops and conditionals. In SSA-form programs, these nodes are created by ϕ-functions. These instructions give us special constraints, as Figure 7 shows. A ϕ-function such as $v = \phi(v_1, v_2)$ yields the constraint $\{v = \phi(v_i)\}, 1 \leq i \leq 2$. To solve this kind of constraint, we must find out how the value of v evolves during the execution of the program. It might increase, it might decrease, or it might oscillate. To obtain this information, we perform a *growth check*. On the resolution of each join constraint, we check if the variable defined in it (the left-side operand) is present in the LT or GT sets of some right-side operand. If the left-side operand is present in the LT of at least one right-side operand, and not present in any GT of these right-side operands, then the program contains only execution paths in which the left-side operand grows: it is receiving a value that is greater than itself. Dually, if the left-side operand is present in the GT of at least one operand and in the LT of any, then there exists only paths in the program along which the left-side operand decreases. Example 4.5 illustrates these observations.

Example 4.5 *Variable x in Figure 9 (a) increases along the execution of the program. Differently, variable x in Figure 9 (b) oscillates: it might increase or decrease, depending on the path along which the program flows. We show how to deal with these different cases to solve constraints in Figure 10.*

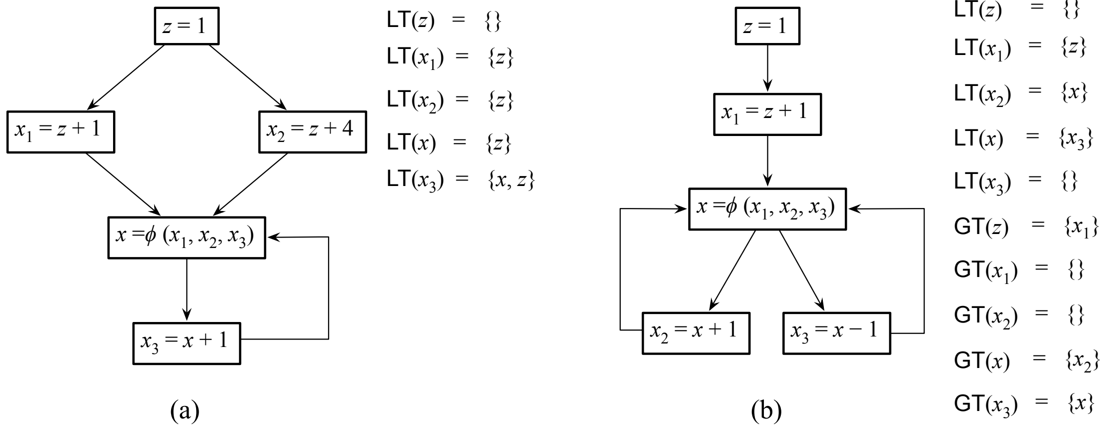


Figure 9: Less-than and greater-than sets built by our analysis.

Figure 10 shows how we handle constraints that exist due to ϕ-functions. We use *conditional constraints* to deal with them. A conditional constraint is formed by a *trigger* T and an *action* a, denoted by the notation $T \Rightarrow a$. The action a is evaluated only if the trigger T is true. The constraint $x = \phi(x_i)$ has three different triggers, and at any moment, at least one of them is true. Triggers check if particular LT sets contain specific variables. Example 4.6 explains how we compute the less-than set of x.

$$
\begin{array}{r} \mathit{join} : x = \phi(x_i) \\ 0 \leqslant i < k \end{array} \Rightarrow
\begin{cases}
i : (A) \wedge \neg(B) \Rightarrow \begin{cases} \mathsf{LT}(x) \cup = \bigcap\limits_{x \notin \mathsf{LT}(x_i)} \mathsf{LT}(x_i) \\ \mathsf{GT}(x) \cup = \bigcap\limits_{i=0}^{n} \mathsf{GT}(x_i) \end{cases} \\[2em]
ii : \neg(A) \wedge (B) \Rightarrow \begin{cases} \mathsf{LT}(x) \cup = \bigcap\limits_{i=0}^{n} \mathsf{LT}(x_i) \\ \mathsf{GT}(x) \cup = \bigcap\limits_{x \notin GT(x_i)} \mathsf{GT}(x_i) \end{cases} \\[2em]
iii : \text{Otherwise} \Rightarrow \begin{cases} \mathsf{LT}(x) \cup = \bigcap\limits_{i=0}^{n} \mathsf{LT}(x_i) \\ \mathsf{GT}(x) \cup = \bigcap\limits_{i=0}^{n} \mathsf{GT}(x_i) \end{cases}
\end{cases}
$$

Figure 10: Solving constraints for ϕ-functions. We let $x' \in \{x, x_T, x_F\}$, $(A) \equiv \exists k, x' \in \mathsf{LT}(x_k)$ and $(B) \equiv \exists k, x' \in \mathsf{GT}(x_k)$. (A) indicates that x increases in the loop, and (B) indicates that x decreases.

Example 4.6 *Variable x, defined in Figure 9 (a) has increasing value, because it belongs into LT (x_3) and GT $(x_i) = \emptyset$ $\forall i \in \{1..3\}$. Therefore, trigger i, in Figure 10 applies, and we know that $\mathsf{LT}(x) = \mathsf{LT}(x) \cup (\mathsf{LT}(x_1) \cap \mathsf{LT}(x_2))$. Thus, $\mathsf{LT}(x) = \mathsf{LT}(x) \cup (\{z\} \cap \{z\}) = \{z\}$.*

Implementation of the Constraint Solver We solve the constraint set C via the method of *Chaotic Iterations* [24, p-176]. We repeatedly insert constraints into a worklist W, and solve them in the order they are inserted. The evaluation of a constraint might lead to the insertion of other constraints into W. This insertion is guided by a *Constraint Dependence Graph* (CDG). Each vertex v of the CDG represents a constraint, and we have an edge from v_1 to v_2 if, and only if, the constraint represented by v_2 reads a set produced by the constraint represented by v_1. This algorithm has asymptotic complexity $\mathcal{O}(n^3)$. However, in practice our implementation runs in time linear on the number of constraints, as we show empirically in Section 5. The process of constraint resolution is guaranteed to terminate, as Theorem 4.8 proofs. Example 4.7 illustrates this approach.

Example 4.7 *We let $C = \{x < y, y < z\}$. The result of solving C to build LT sets is given in Figure 11. Our worklist is initialized to the constraint set C. Each variable v is bound to an abstract state $\mathsf{LT}(v) = \emptyset$ and $\mathsf{GT}(v) = \emptyset$. Resolution reaches a fix point when the worklist is empty. In each iteration, a constraint is popped, and abstract states are updated following the rules in Figures 8 and 10. We also update the worklist according to the constraint dependence graph.*

WL_i	$pop(WL_i)$	$\mathsf{LT}(x)$	$\mathsf{LT}(y)$	$\mathsf{LT}(z)$
$\{x < y, y < z\}$	$-$	$\emptyset$	$\emptyset$	$\emptyset$
$\{x < y, y < z\}$	$x < y$	$\emptyset$	$\{x\}$	$\emptyset$
$\{y < z, x < y\}$	$y < z$	$\emptyset$	$\{x\}$	$\{y, x\}$
$\{x < y, y < z\}$	$x < y$	$\emptyset$	$\{x\}$	$\{y, x\}$
$\{y < z\}$	$y < z$	$\emptyset$	$\{x\}$	$\{y, x\}$
$\emptyset$	$-$	$\emptyset$	$\{x\}$	$\{y, x\}$

Figure 11: Worklist steps for solving constraints of Example 4.7. Each line depicts a step of the algorithm.

Theorem 4.8 (Termination) *Constraint resolution is guaranteed to terminate.*

Proof: The worklist based solver reaches a fixed point, because of two reasons. First, the updating of abstract states is monotonic. The inspection of Figures 8 and 10 shows that the abstract state of variable v, e.g., $\mathsf{LT}(v)$ and $\mathsf{GT}(v)$ is only updated via union with itself plus extra information, if available. Second, abstract states are represented by points in a lattice of finite height: at most the less-than or greater-than set of a variable will contain all the other variables in the program. □

Example 4.9 *Figure 4 gives us the following constraint system: $\mathsf{C} = \{N' < N,\ i = \phi(i_0, i_1)$, $i_T < N'_T$, $N'_F \le i_F$; $i_T < j_0$, $j = \phi(j_0, j_1)$, $i_T = i$, $j_T = j$, $j_T < N_T$, $N_F \le j_F$, $v \le v_i$, $v < v_j$, $v \le v'_i$, $v < v'_j$, $j_T < j_1$, $i_T < i_1$, $x_{jT} < x_{iT}$, $x_{iF} \le x_{jF}\}$. The solution that we find for the LT sets is the following: $\mathsf{LT}(i_0) = \mathsf{LT}(N') = \mathsf{LT}(j_F) = \mathsf{LT}(i_F) = \mathsf{LT}(N'_F) = \mathsf{LT}(x_{jT}) = \mathsf{LT}(x_{iF}) = \mathsf{LT}(x_{jF}) = \mathsf{LT}(i) = \emptyset$, $\mathsf{LT}(i_1) = \{i_T\}$, $\mathsf{LT}(N) = \{N'\}$, $\mathsf{LT}(j_0) = \{i_T, i_0\}$, $\mathsf{LT}(j_1) = \{j_T, j_0, i_T\}$, $\mathsf{LT}(i_T) = \{i_0\}$, $\mathsf{LT}(j) = \{i_T\}$, $\mathsf{LT}(j_T) = \{j_0, i_T\}$, $\mathsf{LT}(N'_T) = \{i_T, i_0\}$, $\mathsf{LT}(x_{iT}) = \{x_{jT}\}$, $\mathsf{LT}(v_j) = \{v\}$, $\mathsf{LT}(v'_j) = \{v\}$. For brevity, we omit GT sets.*

The Semantics of Constraints The *concrete state* of a program variable v is the set of values that v can receive throughout the execution of a program. The *abstract state* of a variable v is $\mathsf{LT}(v)$. If $y \in \mathsf{LT}(v)$, then we know that y is strictly less than v at every program point where y and v are simultaneously alive. In other words, it is still possible that $y \in \mathsf{LT}(v)$, but $y \not< v$ if we look into the concrete values of these variables at different moments during the execution of the same program. Theorem 4.10 states the core property that our constraint system delivers.

Theorem 4.10 (Correctness of the Strict Less-Than Relations) *If $y \in LT(v)$, then $y < v$ at every program point where y and v are simultaneously alive.*

Proof: The proof of Theorem 4.10 consists in a case analysis on the different constraints that can create the relation $y \in \mathsf{LT}(v)$ in Figure 8. We go over a few cases. The proof goes by induction on the size of LT; if LT is empty, the property is trivially verified.

- The constraint *Strict less than* $x < v$ updates the strict less than set of v. $\mathsf{LT}(v) = \mathsf{LT}(v) \cup \mathsf{LT}(x) \cup \{x\}$. For the sake of clarity, we let $\mathsf{LT}'(v)$ be the strict less than set of v before the update introduced by the constraint $x < v$. Henceforth, $y \in \mathsf{LT}(v)$ if $y \in \mathsf{LT}'(v)$ or $y \in \mathsf{LT}(x)$, or $y = x$. If $y = x$, then we are done, due to the constraint $x < v$. Otherwise, if $y \in \mathsf{LT}(x)$, by induction, $y < x$, and by transitivity we get $y < x$. In case $y \in \mathsf{LT}'(v)$, by induction on $\mathsf{LT}'(v), y < v$. In fact, since the strict less than set of v is only growing up if updated, then if a property holds for its members once, it holds even after updates. In other words, $\mathsf{LT}'(v)$ may be an update of $\mathsf{LT}''(v)$ in which we have inserted y after resolving a constraint. In this case, $\mathsf{LT}''(v) \subseteq \mathsf{LT}'(v) \subseteq \mathsf{LT}(v)$ with $y \in \mathsf{LT}''(v)$. In the remaining of the proof, we shall be interested only on elements updating $\mathsf{LT}(v)$.
- The constraint *non reflexive equal* $v = x$, updates $\mathsf{LT}(v)$ with $\mathsf{LT}(x)$: $\mathsf{LT}(v) = \mathsf{LT}(v) \cup \mathsf{LT}(x)$. We focus on $y \in \mathsf{LT}(x)$. By induction we get $y < x$. From Figure 7 we know that the constraint comes from a test. Thus, w.l.o.g, we can assume that this constraint is $x_1^T = x^1$, coming from a test $x^1 < x^2$. The condition $y < x^1$ is true before the test. Because the e-SSA conversion does not change the semantics of the target program, the condition is still true after renaming the operands of the test, thus $y < x_1^T = v$.
- The constraint $v = \phi(v_i)$. We show the proof for the first case of Figure 10, the two others are similar. In this case we have $\exists i_0, v' \in \mathsf{LT}(v_{i_0})$ and $\forall i, v_i \notin \mathsf{LT}(v')$ implying $\mathsf{LT}(v) \cup = \bigcap_{v \notin \mathsf{LT}(v_i)} \mathsf{LT}(v_i)$. $v' \in \{v, v_T, v_F\}$. Recall that in the general case of ϕ function, we should have $\mathsf{LT}(v) \cup = \bigcap_{i=0}^{n} \mathsf{LT}(v_i)$. Consider the case where $v' = v$. We let $y \in \bigcap \mathsf{LT}(v_i)$ with $v \notin \mathsf{LT}(v_i)$.
  - For all i such that $v \notin \mathsf{LT}(v_i)$: By the induction hypothesis $y < v_i$
  - For the other i indices, i.e. $v \in \mathsf{LT}(v_i)$: Assumptions $\exists i_0, v \in \mathsf{LT}(v_{i_0})$ and $\forall i, v_i \notin \mathsf{LT}(v)$ imply that all redefinitions of v (after the ϕ function) are of the form $v_i = v_j + x$ with $x > 0$. (or any equivalent sequence of statements). Then: there exists a redefinition of v_i with a strictly less v_j. As there cannot be any infinite decreasing sequence of redefinitions, there exists a v_j such that $y < v_j$ and $v \notin \mathsf{LT}(v_j)$.

  Similar to v, in case $v' \in \{v_T, v_F\}$, we get $v_i = \sigma(v_j) + x$ and there exists v_j such that $y < v_j$ and $\sigma(v) \notin \mathsf{LT}(v_j)$. Proving $y < \sigma(v_j)$ is straightforward since $y < v_j$.

  Thus $y < v_i$ for all i indices. By property of the ϕ function, we finally get $y < x$. $\square$

4.5 Answering queries

The techniques discussed in the previous section lets us prove that some pairs of pointers cannot dereference overlapping memory regions. We call this process *pointer disambiguation*. We use three different tests to show that pointers cannot alias each other. Figure 12 illustrates the

relationship between these tests. If we cannot conclude that two pointers always refer to distinct regions, then we say – conservativelly – that they *may alias*.

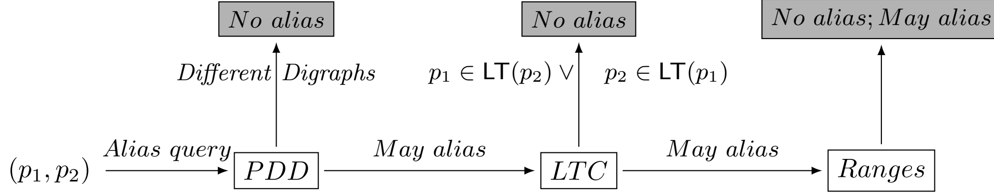


Figure 12: Steps used in the resolution of pointer disambiguation queries.

In the rest of this section, we provide further details about the three pointer disambiguation tests that we use. Briefly, we summarize them as follows, assuming that we want to disambiguate pointers p_1 and p_2:

PDD: If p_1 and p_2 belong into different pointer dependance digraphs, then they are said to be unrelated. PDD are described in Section 4.2.

Less-Than: We consider two disambiguation criteria:

1. If $p_1 \in \mathsf{LT}(p_2)$, or vice-versa, then these pointers cannot point to overlapping memory regions.
2. Memory locations $p_1 = p + x_1$ and $p_2 = p + x_2$ will not alias if $x_1 \in \mathsf{LT}(x_2)$ or $x_2 \in \mathsf{LT}(x_1)$.

Ranges: If we can reconstruct the ranges covered by p_1 and p_2, and these ranges do not overlap, then p_1 and p_2 cannot alias each other.

4.6 The Digraph Test

We have seen, in Section 4.2, how to group pointers that are related by C-style pointer arithmetics into digraphs. Pointers that belong to the same PDD can dereference overlapping memory regions. However, pointers that are in different digraphs cannot alias, because they reference different memory allocation blocks. Memory blocks are different if they have been allocated at different program sites. Example 4.11 illustrates this observation.

Example 4.11 *The program in Figure 13 contains two different memory allocation sites. The first is due to the argument* `argv`*. The second is due to the* `malloc` *operation that initializes pointer* `s1`*. These two different locations will give origin to two different pointer dependance digraphs, as we show in the figure.*

```
1  int main(int argc, char** argv) {
2    int n = strlen(argv[1]);
3    char* s1 = (char*) malloc(n);
4    char* s2 = argv[1];
5    char* s3 = s1;
6    char* s4 = s2 + n;
7    while (s2 < s4) {
8      *s3 = *s2;
9      s2++;
10     s3++;
11   }
12 }
```

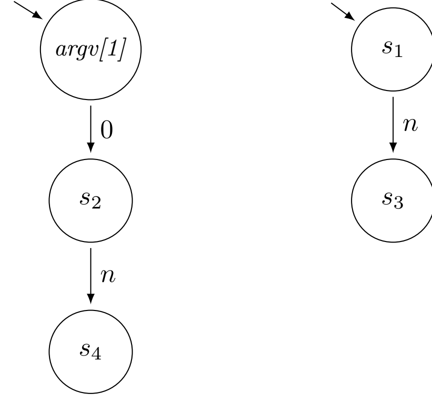


Figure 13: Program that contains two different memory locations, which will give origin to two unconnected pointer dependance digraphs. For the sake of brievty, the program is kept in high level language.

4.7 The Less-Than Test

The less-than test relies on the relations constructed in Section 4.4 to disambiguate pointers. This test is only applied onto pointers that belong to the same pointer dependance digraph. From Theorem 4.10, we know that if $p_1 \in \mathsf{LT}(p_2)$, then $p_1 < p_2$ whenever these two variables exist in the program. Along similar lines, we have $p_1 < p_2$ if $x < y$ with $p_1 = p+x$ and $p_2 = p+y$. Therefore, they cannot dereference aliasing locations. Example 4.12 illustrates the application of this test.

Example 4.12 *We want to show that locations v[i] and v[j] in Figure 1a cannot alias each other. The cfg of function ins_sort appears in Figure 4. In the cfg's low-level representation, v[i] corresponds to v_i, and v[j] corresponds to v_j. We know that $\mathsf{LT}(j_T) = \{j_0, i_T\}$ (as seen in Example 4.9) and $v_i = v+i_T$ and $v_j = v+j_T$. Therefore, we can conclude that these two pointers v_i and v_j do not alias.*

4.8 The Ranges Test

The so called *Ranges test* is a byproduct of the key components of the PDD previous test. This test consists in determining an expression of the range of intervals covered by two pointers, p_1 and p_2, which share the same pointer digraph. If these two ranges do not intersect, then we can conclude that p_1 and p_2 do not alias. Algorithmically, this test proceeds as follows:

1. Find the *closest common ancestor* p_a of p_1 and p_2. We say that p_a is the *closest common ancestor* of these pointers if, and only if, (i) it is an ancestor, e.g., dominates both p_1 and p_2; and (ii) for any other pointer $p' \neq p_a$ that dominates p_1 and p_2, we have that p' dominates p_a.

2. Rewrite p_1 and p_2 as function of p_a. To this end, we repeat the following re-writing rule:

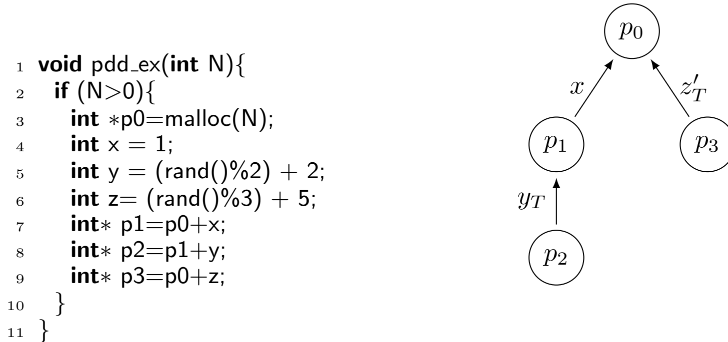
   (a) If $p_x \rightarrow_e p_i, i \in \{1, 2\}$ in the pointer dependance digraph, then we replace p_i by $p_x + e$.

   (b) If $p_x \neq p_a$, then repeat step 2a.

3. Let $p_a + e_1$ and $p_a + e_2$ be the final expressions that we obtain for pointers p_1 and p_2. If $R(e_1) \cap R(e_2) = \emptyset$, then we report that p_1 and p_2 do not alias. Otherwise, we report that these pointers might alias.

Step 2 above is collapsing a path $\mathcal{P}(p_i, p_a) = (p_i, \ldots, p_a)$ in the pointer digraph into a single edge $p_i \rightarrow p_a$. This technique relies on the same ideas introduced by Paisante *et al* [25] to disambiguate pointers: if two pointers cover non-overlapping ranges, then they cannot be alias. However, in terms of implementation, we use range analysis on the integer interval lattice. Paisante *et al.* use a symbolic range analysis. There is no theoretical limitation that prevents us from using a symbolic lattice to reuse Paisante *et al.*'s test. We have opted to use the simpler interval lattice because it is already available in LLVM, the compiler that we have used to implement our alias analysis. Example 4.13 shows how the range test works concretely.

Example 4.13 *Consider the program in Figure14a. Our goal in this example is to disambiguate pointers p_2 and p_3. We have that $\mathsf{LT}(p_0) = \emptyset, \mathsf{LT}(p_1) = \{p_0\}, \mathsf{LT}(p_2) = \{p_0, p_1\}$ and $\mathsf{LT}(p_3) = \{p_0\}$. $p_2 \notin \mathsf{LT}(p_3)$ and $p_3 \notin \mathsf{LT}(p_2)$. The simple less than check is not able to disambiguate these pointers; hence, we resort to the range test. The dependance graph of Figure14a reveals that the lowest common ancestor of p_3 and p_2 is p_0. $\mathcal{P}(p_2, p_0) = (p_2, p_1, p_0)$ and $\mathcal{P}(p_3, p_0) = (p_3, p_0)$. Our re-writing algorithm gives us that $p_2 = p_0 + y + x$, and $p_3 = p_0 + z$. Using range information, we get that: $R(y + x) = R(y) + R(x) = [3, 4]$ and $R(z) = [5, 7]$. These ranges do not intersect; therefore, p_2 and p_3 do not alias.*



(a) A program. (b) Its Pointer Dependance Digraph.

Figure 14: Disambiguating pointers with PDD and Ranges

We close this section with an example in which all the three tests that we have fail. In this case, we say that pointers *may alias*. In Example 4.14, below, we fail to disambiguate pointers, but they alias indeed. We might also fail to disambiguate pointers that do not alias. This type of omission is called a *false positive*.

Example 4.14 *The program in Figure 15 is the same as the program in Figure 14a, except that the ranges of variables* x, y *and* z *have been modified. Due to this modification, none of our three previous tests can prove that p_2 and p_3 do not alias. The tests fails for the following reasons:*

- *p_2 and p_3 are derived from the same base pointer p_0; hence, they may alias due to Section 4.6's test.*

- *Neither $p_3 \in \mathsf{LT}(p_2)$ nor $p_2 \in \mathsf{LT}(p_3)$. y and z, p_2 and p_3' offsets, are not compared since not related to the same base pointer; hence, pointers p_2 and p_3 may alias according to the less-than check of Section 4.7.*

- *The range test of Section 4.8 does not fare better. We have that $p_2 = p_0 + x + y$ and $p_3 = p_0 + z$. $R(x + y) = [3, 6]$ and $R(z) = [3, 7]$. These two interval intersect; hence, the range test reports that p_2 and p_3 may alias.*

```
1 int may_alias(int N, int C) {
2   int* p0 = malloc(N);
3   int x = 1;
4   int y = C ? 2 : 5;
5   int z = C ? 3 : 7;
6   int* p1 = p0 + x;
7   int* p2 = p1 + y;
8   int* p3 = p0 + z;
9 }
```

Figure 15: Our analysis reports that pointers p2 and p3 may alias.

4.9 On the complexity of our analysis

Our analysis can be divided into preprocessing steps and the actual alias tests. The first step in the preprocessing phase is the collection of constraints. This collection runs in linear time on the number of program instructions ($O(i)$), because it consists in going through the code, verifying if each instruction defines constraints. The second step of the preprocessing phase consists in building a pointer dependence graph. At this stage we go through the program instructions searching for pointers. Additionally, pointer attributes are propagated along the graph. This step's complexity is $O(i+p+e)$, with i being the number of program instructions, p the number of pointers and e the number of edges in the dependence graph. The final preprocessing step consists in running the work-list algorithm. This part of our implementation is equivalent to the problem of building transitive closures of graphs. The worst case of transitive closure is cubic and since every variable could be related to all others, the worst case complexity is $O(c^3 * v)$, c being the number of constraints and v the number of variables. In practice, however, the complexity of this phase is $O(c)$, as we demonstrate empirically in Section 5. This lower complexity is justified by a simple observation: a program variable tend to interact with only a handful of other variables. Thus, the number of possible dependences between variables, in practice, is limited by their scope in the source code of programs that we analyze.

After all the preprocessing, each of our three alias tests have constant complexity. In other words, the relevant computations have been already performed on the preprocessing steps. Thus, each test just checks pointers attributes in a table. Keeping this table requires $O(v^2)$ space, as the number of possible relations between variables is quadratic in the worst case. However, usually this table shall demand linear space.

5 Evaluation

We have implemented our alias analysis in the LLVM compiler, version 3.7. In this section, we discuss the empirical evaluation of this implementation. All our experiments have been performed on an Intel i7-5500U, with 16GB of memory, running Ubuntu 15.10. The goal of these experiments is to show that: (i) our alias analysis can increase non-trivially the capacity of LLVM, a mainstream compiler, to disambiguate pointers; (ii) the asymptotic complexity of our algorithm is linear in practice and (iii) the three pointer disambiguation tests are useful.

On the Precision of our Analysis In this section, we compare our analysis against a pointer analysis that is available in LLVM 3.7: the so called *basic alias analysis*, or `basicaa` for short. This algorithm is currently the most effective alias analysis in LLVM, and is the default choice at the -O3 optimization level. It relies on a number of heuristics to disambiguate pointers[3]:

- Distinct globals, stack allocations, and heap allocations can never alias.
- Globals, stack allocations, and heap allocations never alias the null pointer.
- Different fields of a structure do not alias.
- Indexes into arrays with statically distinct subscripts cannot alias.
- Many common standard C library functions never access memory or, if they do it, then these accesses are read-only.
- Stack allocations which never escape the function that allocates them cannot be referenced outside said function.

Program	#Queries	%basicaa	%sraa	%(s + b)
bzip2	2485116	20.83	**50.66**	51.88
omnetpp	13986242	18.71	1.49	19.67
hmmer	2894984	8.55	**19.60**	22.86
h264ref	19809278	12.36	6.95	14.59
gcc	187921335	4.07	**13.82**	15.60
sjeng	528928	67.50	51.89	73.29
astar	90686	40.68	**52.14**	62.14
xalancbmk	10437594	16.72	**27.52**	32.93
gobmk	3556322	45.15	11.99	52.76
mcf	66687	10.70	**31.68**	34.49
perlbench	31129433	7.37	**12.99**	16.28
libquantum	120479	39.40	30.26	53.04
lbm	31193	3.44	**40.17**	41.70

Figure 16: Comparison between two different alias analyses. We let **s + b** be the combination of our technique and the *basic* alias analysis of LLVM. Numbers in **basicaa**, **sraa** and **s+b** show percentage of queries that answer “no-alias".

Figure 16 shows how the LLVM's basic alias analysis and our approach fare when applied on the integer programs in SPEC CPU2006 [16]. Henceforth, we shall call our analysis `sraa`, to distinguish it from LLVM's `basicaa`. We notice that, even though the basic alias analysis disambiguates more pointers in several programs, our approach surpasses it in some benchmarks. It does better than basic on `lbm` and `bzip2`. Moreover, we improve considerably `basicaa`'s results when both analyses run together on `gobmk`. Visual inspection of `lbm`'s code reveals a large number of hardcoded constants. These constants, which are used to index memory, give our analysis the ability to go beyond what `basicaa` can do. This fact shows that LLVM still lack the capacity to benefit from the presence of constants in the target code to disambiguate pointers.

[3]This list has been taken from the LLVM documentation, available at `http://llvm.org/docs/AliasAnalysis.html` in November of 2016

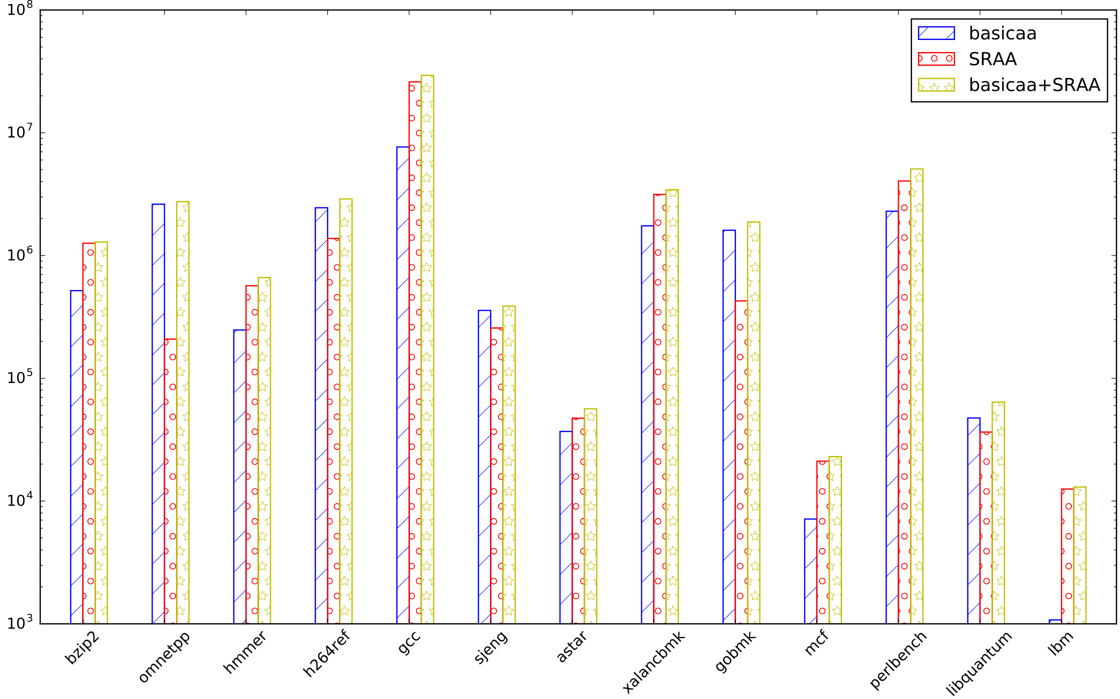


Figure 17: Comparison of SRAA against LLVM's basic alias analysis and how it increases it capacity to disambiguate pointers. The X-axis shows SPEC CPU2006 benchmarks. The Y-axis shows the number of queries answering no alias. The higher the bar, the better.

Figure 17 compares our analysis against `basicaa` in terms of the absolute number of queries that they can resolve. A *query* consists of a pair of pointers. We say that an analysis *solves* a query if it is able to answer *no alias* for the pair of pointers that the query represents. When applied onto the programs available in SPEC CPU2006, both analyses, `basicaa` and `sraa`, are able to solve several queries. There is not a clear winner in this competition, because for each analysis there exist a few benchmarks in which it outperforms the other. However, in absolute terms, `sraa` is able to solve about *twice* more queries ($3.6\ 10^7$ vs $1.9\ 10^7$) than `basicaa`. Because neither analysis is a superset of the other, when combined they deliver even more precision. The obvious conclusion of this experiment is that `sraa` adds a non-trivial amount of precision on top of `basicaa`. A measure of this precision is the number of queries missed by the latter analysis, and solved by the former.

Similar numbers are produced by benchmarks other than SPEC CPU2006. For instance, Figure 18 shows the same comparison between `basicaa` and `sraa`, this time on the 100 largest benchmarks available in the LLVM's test suite. Nevertheless, the results found in Figure 18 are similar to those found in Figure 17. Our `sraa` outperforms `basicaa` in the majority of the tests, and their combination outperforms each of these analyses separately in every one of the 100 samples. We have included the total number of queries in Figure 18, to show that, in general, the number of queries that we solve is proportional to the total number of queries found in each benchmark.

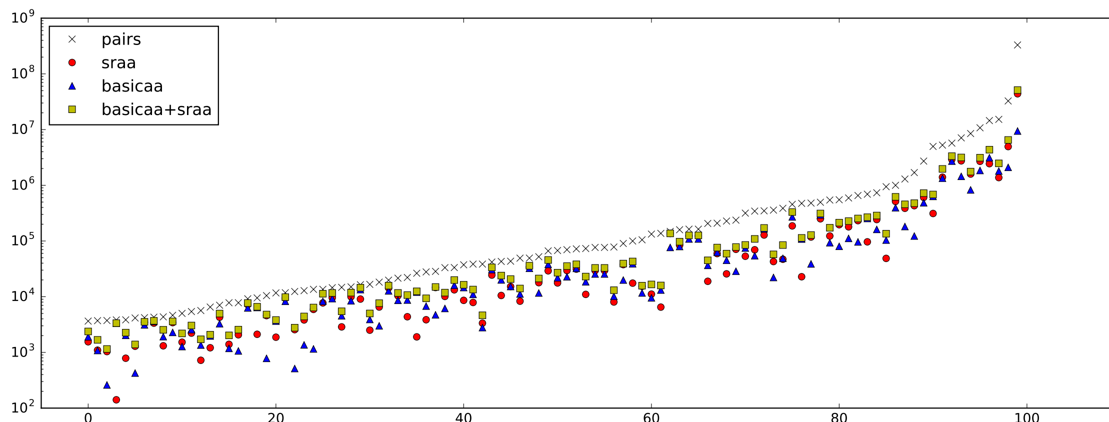


Figure 18: Effectiveness of our alias analysis (SRAA), when compared to LLVM's basic alias analysis on the 100 largest benchmarks in the LLVM test suite (TSCV removed). Each tick in the X-axis represents one benchmark. The Y-axis represents total number of queries (one query per pair of pointers), and number of queries in which each algorithm got a "no-alias" response.

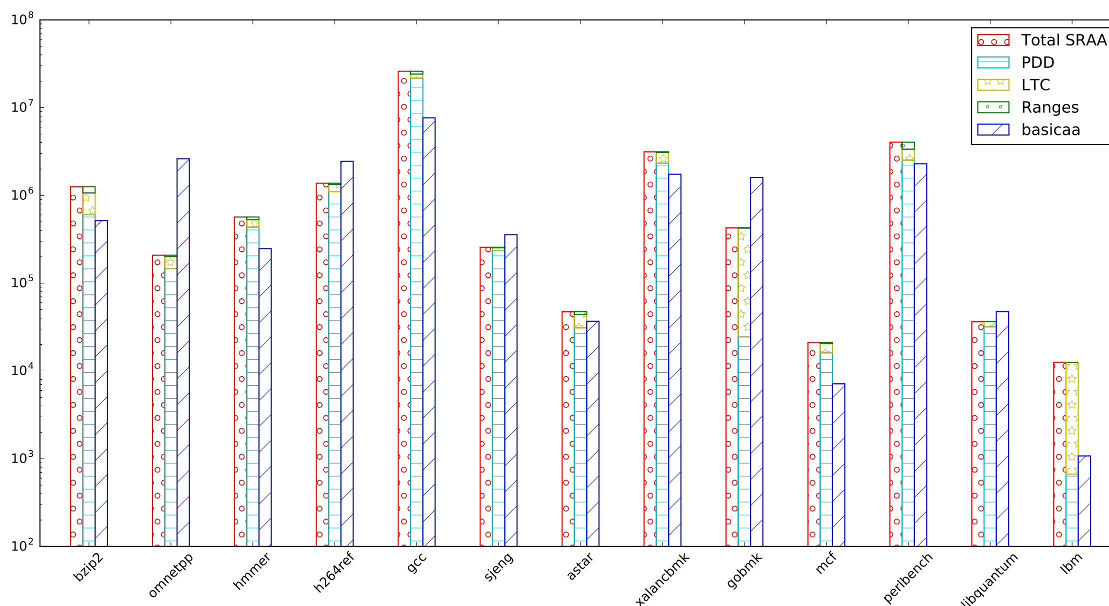


Figure 19: How the three tests in SRAA compete to disambiguate pairs of pointers.

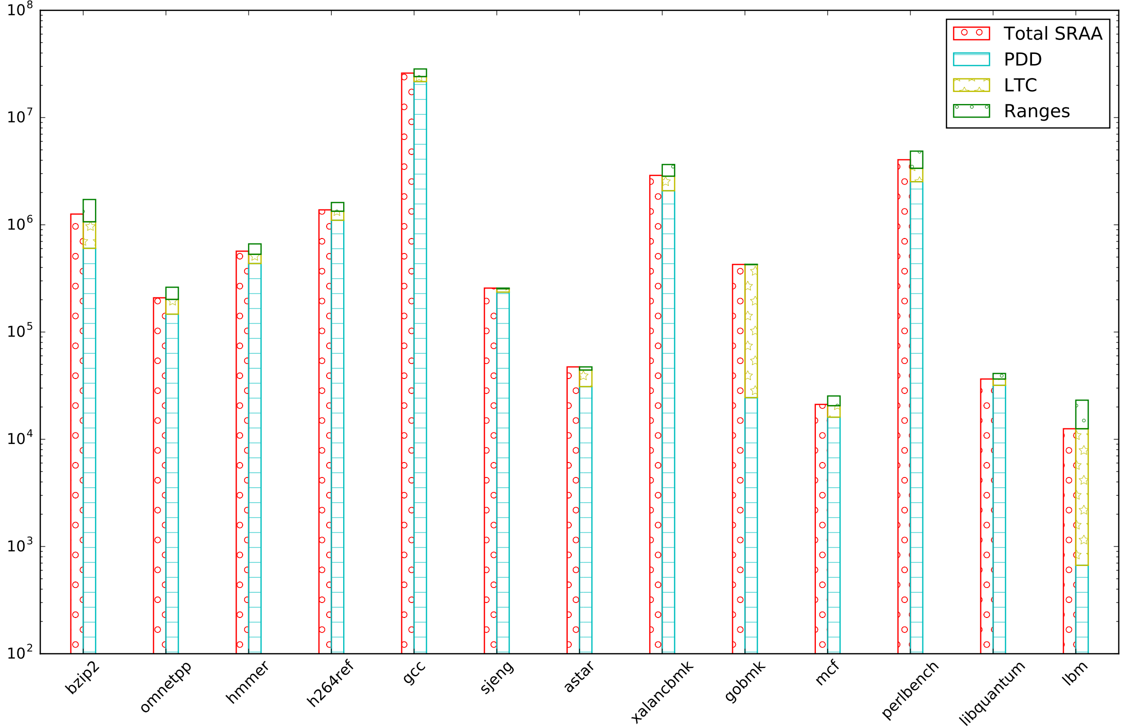


Figure 20: How three tests in SRAA *separately* compete to disambiguate pairs of pointers.

The Role of the Three Disambiguation Tests Figure 19 shows how effective is each one of the three disambiguation tests that we have discussed in Section 4.5. These tests work in succession: given a query q, first we use the PDD test of Section 4.6 to solve it. If this test fails, then we use the less-than test of Section 4.7 to solve q. If this second method still fails to disambiguate that pair of pointers, then we invoke the third test, based on range analysis and discussed in Section 4.8. We call this approach "staged `sraa`". As 19 shows, most of the queries can be answered by simply checking that different pointers belong into different pointer dependence digraphs (PDDs). Yet, the algebraic rules present in the other two tests are essential in some benchmarks. In particular, the less-than check increases the PDD test by more than 30% on average, and in some cases, such as SPEC's `lbm`, it more than doubles it. In most cases indeed, it is thanks to this test that we outperform `basicaa`.

Figure 19 shows that the range analysis test is not very effective when compared to the other two tests. This low effectiveness happens due to two reasons. First, the range test is based solely on a numeric range analysis. Even though a numeric range analysis is enough to distinguish p_1 and p_2 defined as $p_1 = p+1$ and $p_2 = p+2$, usually most of the offsets are symbols, not constants. The ability to use symbols to distinguish pointers is one of the key factors that led us to use a less-than check in this work. Second, the range test is the last one to be applied. Therefore, most of the queries have already been solved by the other two approaches by the time the range test is called. Specifically, there is a large overlap between the less-than test and the range test. We have observed that 11.97% of all the queries solved by the less-than check could also be solved by the range test. To fundament this last point, Figure 20 shows the total number of queries solved by each test. In this experiment, we run each test independently; hence, the success of

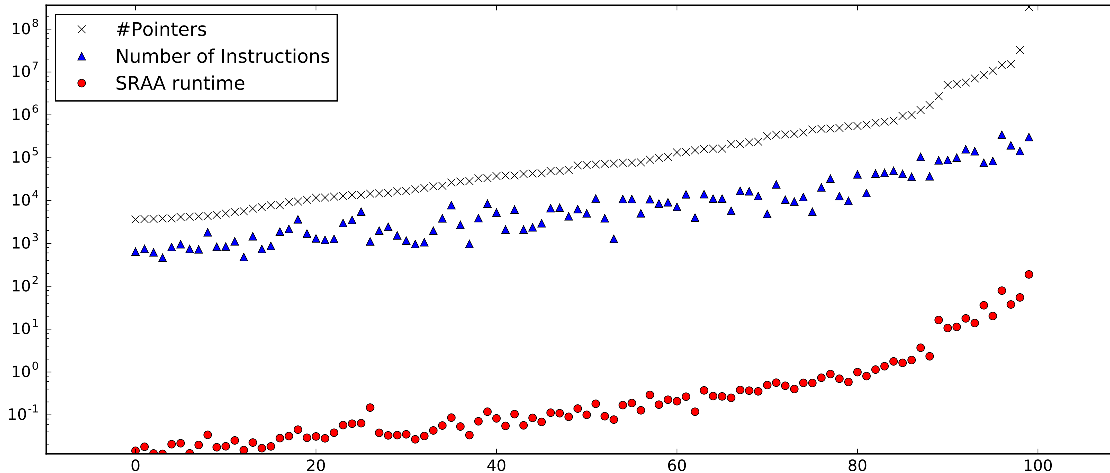


Figure 21: Comparison between the number of queries and total runtime (3 tests) per benchmark. X-axis represents LLVM benchmarks, sorted by number of instructions.

one resolution strategy does not prevent the others from being applied. Nevertheless, the range test of Section 4.8 is still the least effective of the three approaches that we have discussed in this paper.

Runtime Figure 21 shows the runtime of our alias analysis on the 100 largest benchmarks in the LLVM test suite. As the figure shows, we finish all the tests for all the benchmarks, but nine, in less than one second. Nevertheless, we observe a linear behavior. In Figure 21, the coefficient of determination (R^2) between the number of instructions and the runtime of our analysis is 0.8284. The closer to 1.0 is this metric, the more linear is the correlation between these two quantities.

Figure 22 discriminates the runtime of each disambiguation test. `403.gcc`, our largest benchmark, took approximately 200 seconds to finish. Such long times happen because, in the process of solving constraints, we build the transitive closure of the less-than relations between variables. The graph that represents this transitive closure might be cubic on the number of variables in the program. The figure also shows the total time taken by our alias analysis, which includes the time to construct the transitive closure of the pointer dependence digraph and collecting constraints. The PDD test accounts for most of the execution time within the pointer disambiguation phase of this experiment. This behavior happens because this test runs for every query. In Figure 23, the less-than check runs only if the PDD tests fails. Once we have built `LT` relations, this test amounts to consulting hash-tables. Finally, the range tests is the least time-intensive out of the three disambiguation strategies. It takes less time because it only runs for pairs of pointers within the same PDD that the less-than check could not disambiguate.

Figure 24 compares the time to run the three disambiguation tests in a staged fashion against the time to run these tests independently. The non-staged approach is theoretically slower, because it always runs $3 \times Q$ tests, whereas the staged approach runs $Q + (Q - S_1) + (Q - S_1 - S_2)$, where S_1 are the queries solved by the first test, and S_2 are the queries solved by the second. Nevertheless, Figure 24 does not show a clear winner. The fact that the three test are relatively fast explains this result.

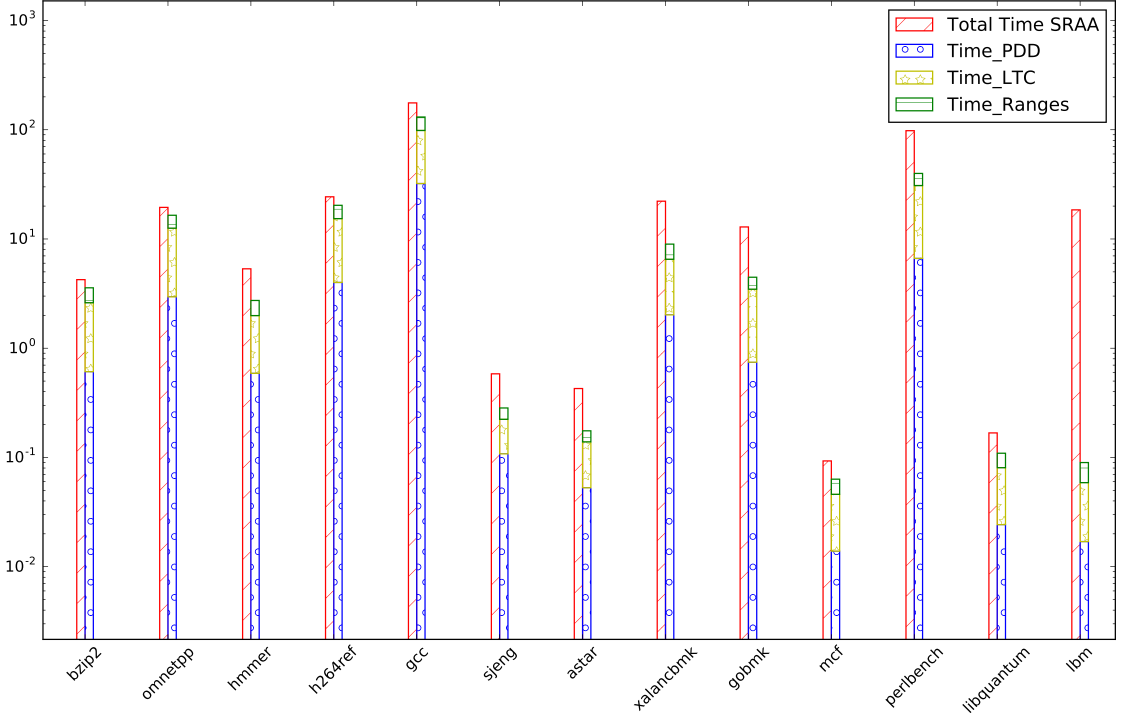


Figure 22: Time needed by each test (PDD, LTC, and Ranges) of our analysis SRAA to disambiguate SPEC-CPU2006 pairs of pointers.

6 Related Work

This work joins two techniques that compilers use to understand programs: alias and less-than analyses. Both these techniques have spurred a long string of publications within the programming languages literature. We do not know of another work that joins both these research directions into a single path. In the rest of this section we describe some of the key work along each of these lines.

6.1 Algebraic Pointer Disambiguation Techniques

We call *algebraic alias analyses* the many techniques that use arithmetics to disambiguate pointers. Much of the work on algebraic pointer disambiguation had its origins on the needs of automatic parallelization. For instance, several automatic parallelization techniques rely on some way to associate symbolic offsets, usually loop bounds, with pointers. Michael Wolfe [38, Ch.7] and Aho et al. [1, Ch.11] have entire chapters devoted to this issue. The key difference between our work and this line of research is the algorithm to solve pointer relations: they resort to integer linear programming (ILP) or the Greatest Common Divisor test to solve diophantine equations, whereas we do abstract interpretation. Even Rugina and Rinard [29], who we believe is the state-of-the-art approach in the field today, use integer linear programming to solve symbolic relations between variables. We speculate that the ILP approach is too expensive to be used in large programs; hence, concessions must be made for the sake of speed. For instance, whereas the previous literature that we know restrict their expriments to pointers within loops, we can

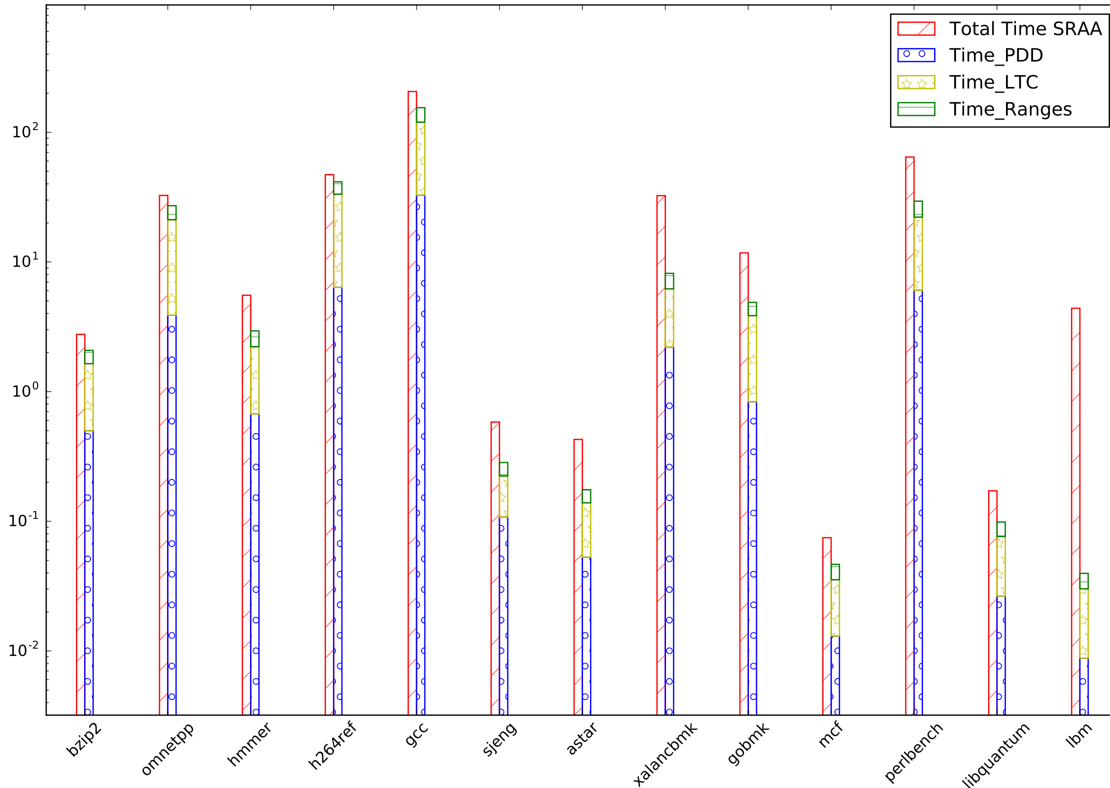


Figure 23: Time needed by each test (PDD, LTC, and Ranges) when run selectively to disambiguate SPEC-2006 pairs of pointers.

analyze programs with over one million assembly instructions.

There exist several different pointer disambiguation strategies that associate ranges with pointers [5, 6, 3, 36, 23, 25, 29, 30, 33, 37]. They all share a common idea: two memory addresses $p_1 + [l_1, u_1]$ and $p_2 + [l_2, u_2]$ do not alias if the intervals $[p_1 + l_1, p_1 + u_1]$ and $[p_2 + l_2, p_2 + u_2]$ do not overlap. These analyses differ in the way they represent intervals, e.g., with holes [5, 33] or contiguously [3, 30]; with symbolic bounds [23, 25, 29] or with numeric bounds [5, 6, 33], etc. None of these previous work is strictly better than ours. For instance, none of them can disambiguate `v[i]` and `v[j]` in Figure 1 (b), because these locations cover regions that overlap, albeit not at the same time. Nevertheless, range based disambiguation methods can solve queries that a simple less-than approach cannot. As an example, strict inequalities are unable to disambiguate p_1 and p_2, given these definitions: $p_1 = p + 1$ and $p_2 = p + 2$. We know that $p < p_1$ and $p < p_2$, but we do not relate p_1 and p_2. This observation has led us to incorporate a range-based disambiguation test in our framework.

Notice that there exist a vast literature on non-algebraic pointer disambiguation techniques. Our work does not compete against them; rather, it complements them. In other words, our representation of pointers can be used to enhance the precision of algorithms such as Steensgard's [32], Andersen's [4], or even the state-of-the-art technique of Hardekopf and Lin [15]. These techniques map pointers to sets of locations, but they could be augmented to map pointers to sets of locations plus ranges. Furthermore, the use of our approach does not prevent the employment of acceleration techniques such as lazy cycle detection [14], or wave propagation [26].

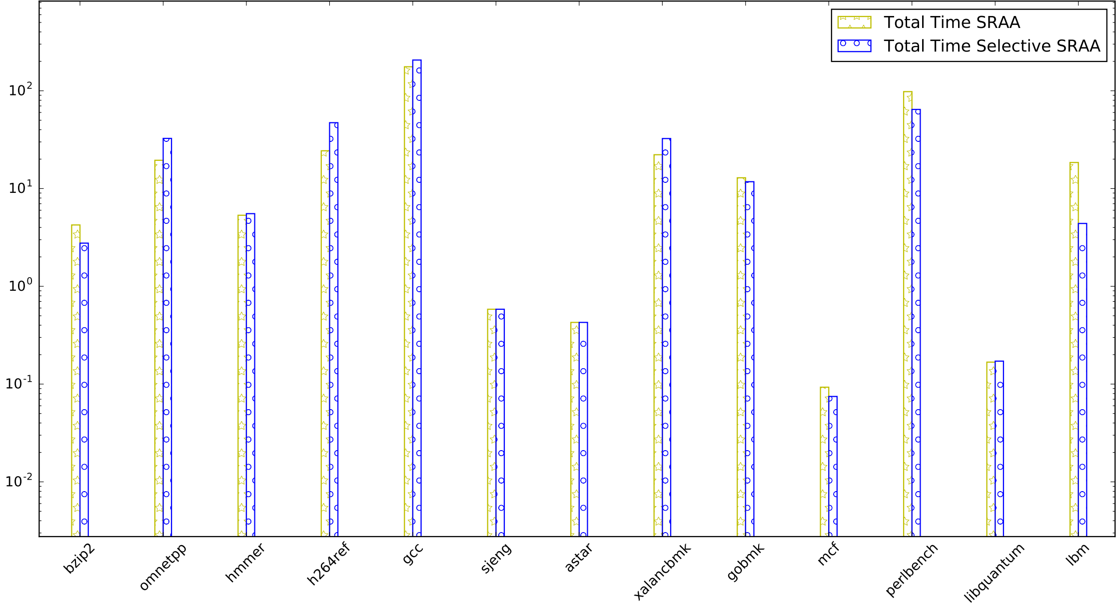


Figure 24: Comparison between time needed to run SRAA *selectively* (non overlapping tests) and *separately* (disjoint tests)

6.2 Less-Than Relations

The insight of using a less-than dataflow analysis to disambiguate pointers is an original contribution of this paper. However, such a static analysis is not new, having been used before to eliminate array bound checks. We know of two different approaches to build less-than relations: Logozzo's [20, 21] and Bodik's [8]. Additionally, there exist non-relational analyses that produce enough information to solve less-than equations [10, 22]. In the rest of this section we discuss the differences between such work and ours.

The ABCD Algorithm The work that most closely resembles ours is Bodik *et al.*'s ABCD (short for Array Bounds Checks on Demand) algorithm [8]. Similarities stem from the fact that Bodik *et al.* also build a new program representation to achieve a sparse less-than analysis. However, there are some key differences between that approach and ours. The first difference is a matter of presentation: Bodik *et al.* provide a geometric interpretation to the problem of building less-than relations, whereas we adopt an algebraic formalization. Bodik *et al.* keep track of such relations via a data-structure called the *inequality graph*. This graph is akin to the pointer dependence graph that we have used in this paper.

Bodik *et al.*'s technique, in principle could be used to implement our less-than check; however, they use a different algorithm to prove that a variable is less than another. In the absence of cycles in the inequality graph, their approach works like ours: a positive path between v_i to v_j indicates that $x_i < x_j$. This path is implicit in the transitive closure that we produce after solving constraints. However, they use an extra step to handle cycles, which, in our opinion, makes their algorithm difficult to reason about. Upon finding a cycle in the inequality graph, Bodik *et al.* try to mark this cycle as *increasing* or *decreasing*. Cycles always exist due to ϕ-functions. Decreasing cycles cause ϕ-functions to be abstractly evaluated with the *minimum* operator applied on the weights of incoming edges; increasing cycles invoke *maximum* instead.

Third, Bodik *et al.* do not use range analysis, because ABCD has been designed for just-in-time compilers, where runtime is an issue. Nevertheless, this limitation prevents ABCD from handling instructions such as $x_1 = x_2 + x_3$ if neither x_2 nor x_3 are constants. Finally, we chose to compute a transitive closure of less-than relations, whereas ABCD works on demand. This point is a technicality. In our experiments, we had to deal with millions of queries. If we tried to answer them on demand, like ABCD does, then said experiments would take too long. We build the transitive closure to answer queries in $O(1)$ in practice.

The Pentagon Lattice Logozzo and Fähndrich [20, 21] have proposed the Pentagon Lattice to eliminate array bound checks in type safe languages such as C#. This algebraic object is the combination of the lattice of integer intervals and the less-than lattice. Pentagons, like the ABCD algorithm, could be used to disambiguate pointers like we do. Nevertheless, there are differences between our algorithm and Logozzo's. First, the original work on Pentagons describe a dense analysis, whereas we use a different program representation to achieve sparsity. Contrary to ABCD, the Pentagon analysis infers that $x_2 > x_1$ given $x_1 = x_2 - x_3, x_3 > 0$ like we do, albeit on a dense fashion. Second, Logozzo and Fähndrich build less-than and range relations together, whereas our analysis first builds range information, then uses it to compute less-than relations. We have not found thus far examples in which one approach yields better results than the other; however, we believe that, from an engineering point of view, decoupling both analyses leads to simpler implementations.

Fully-Relational Analyses Our less-than analysis, ABCD and Pentagons are said to be *semi-relational*, meaning that they associate single program variables with sets of other variables. Fully-relational analysis, such as Octogons [22] or Polyhedrons [10], associate tuples of variables with abstract information. For instance, Miné's Octogons build relations such as $x_1 + x_2 \leq 1$, where x_1 and x_2 are variables in the target program. As an example, Polly-LLVM[4] uses fully-relational techniques to analyze loops. Polly's dependence analysis is able to distinguish `v[i]` and `v[j]` in Figure 1 (a), given that $\mathtt{j} - \mathtt{i} \geq 1$; however, it cannot analyze `v[i]` and `v[j]` in Figure 1 (b). These analyses are very powerful; however, they face scalability problems when dealing with large programs. Whereas a semi-relational sparse analysis generates $O(|\mathcal{V}|)$ constraints, $|\mathcal{V}|$ being the number of program variables, a relational one might produce $O(|\mathcal{V}|^k)$, k being the number of variables used in relations. As an example, current state-of-the art static analysers such as Astrée [11] or Pagai [17] assign *unknown values for base adresses*. These values permit such tools to derive relation between pointers, treating them as numerical values. In practise, these analyses work because they handle programs with very few pointers, like safety critical code.

7 Conclusion

This paper has described a novel algebraic method to disambiguate pointers. The technique that we have introduced in this paper uses a combination of less-than analysis and classical range analysis to show that two pointers cannot dereference the same memory location. We have demonstrated that our technique is effective and useful: its implementation on LLVM lets us increase the ability of this compiler to separate pointers by almost five times in some cases. And, contrary to previous algebraic approaches, our analysis scales up to very large programs, with millions of assembly instructions. We believe that this type of technique opens up opportunities

[4] Available at http://polly.llvm.org/

to new program optimizations. The implementation of such optimizations is a challenge that we hope to address thenceforth.

Acknowledgments

This project is supported by CNPq, Intel (The eCoSoC grant), FAPEMIG (The Prospiel project), and by the French National Research Agency - ANR (LABEX MILYON of Université de Lyon, within the program "Investissement d'Avenir" (ANR-11-IDEX-0007)).

Contents

1 Introduction 3

2 Overview 4

3 Preliminary Definitions 5
3.1 Sparse Analysis 6

4 Pointer Disambiguation Based on Strict Inequalities 7
4.1 Range Analysis 7
4.2 Grouping Pointers in Pointer Digraphs 8
4.3 Collecting constraints 8
4.4 Solving constraints 10
4.5 Answering queries 14
4.6 The Digraph Test 15
4.7 The Less-Than Test 16
4.8 The Ranges Test 16
4.9 On the complexity of our analysis 18

5 Evaluation 19

6 Related Work 24
6.1 Algebraic Pointer Disambiguation Techniques 24
6.2 Less-Than Relations 26

7 Conclusion 27

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL*, pages 1–11. ACM, 1988.

[3] Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. Runtime pointer disambiguation. In *OOPSLA*, pages 589–606. ACM, 2015.

[4] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[5] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *CC*, pages 5–23. Springer, 2004.

[6] George Balatsouras and Yannis Smaragdakis. Structure-sensitive points-to analysis for C and C++. In *SAS*, pages 84–104. Springer, 2016.

[7] William Blume and Rudolf Eigenmann. Symbolic range propagation. In *IPPS*, pages 357–363, 1994.

[8] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM, 2000.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[10] P. Cousot and N.. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM, 1978.

[11] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *European symposium on programming (ESOP)*, number 3444 in Lecture Notes in Computer Science, pages 21–30, 2005.

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *POPL*, pages 25–35, 1989.

[13] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, pages 10–30. Springer, 2010.

[14] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, pages 290–299. ACM, 2007.

[15] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO*, pages 265–280, 2011.

[16] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[17] Julien Henry, David Monniaux, and Matthieu Moy. Succinct representations for abstract interpretation: Combined analysis algorithms and experimental evaluation. In *SAS*, pages 283–299. Springer, 2012.

[18] ISO. *Programming Languages – C*. IEC, 9899:2011.

[19] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.

[20] Fancesco Logozzo and Manuel Fähndrich. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *SAC*, pages 184–188. ACM, 2008.

[21] Francesco Logozzo and Manuel Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. *Sci. Comput. Program.*, 75(9):796–807, 2010.

[22] Antoine Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19:31–100, 2006.

[23] Henrique Nazaré, Izabela Maffra, Willer Santos, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. Validation of memory accesses through symbolic analyses. In *OOPSLA*, pages 791–809. ACM, 2014.

[24] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2005.

[25] Vitor Paisante, Maroua Maalej, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. Symbolic range analysis of pointers. In *CGO*, pages 171–181. ACM, 2016.

[26] Fernando Magno Quintao Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *CGO*, pages 126–135. IEEE, 2009.

[27] Raphael Ernani Rodrigues, Victor Hugo Sperle Campos, and Fernando Magno Quintao Pereira. A fast and low overhead technique to secure programs against integer overflows. In *CGO*, pages 1–13. ACM, 2013.

[28] Radu Rugina and Martin C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *PLDI*, pages 182–195. ACM, 2000.

[29] Radu Rugina and Martin C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *TOPLAS*, 27(2):185–235, 2005.

[30] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid analysis: Static and dynamic memory reference analysis. In *ICS*, pages 251–283. IEEE, 2002.

[31] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: a fast address sanity checker. In *ATC*, pages 28–28. USENIX, 2012.

[32] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, 1996.

[33] Yulei Sui, XIaokang Fan, Hao Zhou, and Jingling Xue. Loop-oriented array- and field-sensitive pointer analysis for automatic SIMD vectorization. In *LCTES*, pages 41–51. ACM, 2016.

[34] Rishi Surendran, Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. Inter-iteration scalar replacement using array SSA form. In *CC*, pages 40–60. Springer, 2014.

[35] Andre L. C. Tavares, Benoit Boissinot, Fernando M. Q. Pereira, and Fabrice Rastello. Parameterized construction of program representations for sparse dataflow analyses. In *CC*, pages 2–21. Springer, 2014.

[36] Robert A. van Engelen, J. Birch, Y. Shou, B. Walsh, and Kyle A. Gallivan. A unified framework for nonlinear dependence testing and symbolic analysis. In *ICS*, pages 106–115. ACM, 2004.

[37] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *PLDI*, pages 1–12. ACM, 1995.

[38] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Adison-Wesley, 1st edition, 1996.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399