Combining Range and Inequality Information for Pointer Disambiguation

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

Author Version of «Combining Range and Inequality Information
for Pointer Disambiguation », accepted in Science of Computer Programming, 2017

Combining Range and Inequality Information for Pointer Disambiguation

Maroua Maalej[a], Vitor Paisante[b], Fernando Magno Quintão Pereira[b], Laure Gonnord[a]

[a] *University of Lyon & LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA)*
F-69000 Lyon, France
[b] *Laboratório de Compiladores, UFMG, Brazil*

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.

With this new algorithm, we are able to disambiguate pointers with offsets, that commonly occur in C programs, in a precise and efficient way. Together with this new abstract domain we describe several implementation decisions that let 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 CPU2006's `gcc` in a few minutes. Furthermore, it improves on LLVM's industrial quality analyses. As an extreme example, we have observed a 4x improvement when analyzing SPEC's `lbm`.

Keywords: Points-to Analysis, Pentagons, Less-Than Check, Abstract Interpretation, Compiler Construction, Static Analysis

1. Introduction

Pointers are a staple feature in imperative programming languages. Among the many abstractions that exist around this notion, *pointer arithmetics* stands out. Although a popular feature, it yields programs that are particularly difficult to analyze statically. We call pointer arithmetics the ability to add an offset to a pointer, as in `p[i]`, or `*(p + i)`. Pointer arithmetics is

present in programming languages including C, C++ and some assembly dialects; thus, it is commonly found in some of the most popular software used in the world, such as operating system kernels, browsers, compilers and webservers. Therefore, it is important that compilers, as well as static analysis tools, understand and analyze programs supporting this kind of construction.

However, designing and implementing alias analyses that disambiguate pointers with offsets is very challenging. The problem of statically disambiguating pointers $p_0 + e_0$ and $p_1 + e_1$ involves two challenges: distinguishing references p_0 and p_1, and comparing the integer expressions e_0 and e_1. The first task is the traditional alias analysis problem, and bears a vast set of difficulties well studied in the literature [Hind (2001)]. The second task amounts to solving Hilbert's Tenth Problem [Davis et al. (1976)], which is undecidable in general. Whereas we find plenty of techniques to deal with the former task in the literature, the latter is not as popular. Testimony to this observation is the fact that the most well-known alias analyses, such as Andersen's [Andersen (1994)] or Steensgard's [Steensgaard (1996)] formulation, do not address this problem. This paper, which subsumes three years of research partially published in two previous works [Maalej et al. (2017); Paisante et al. (2016)], provides a theoretical and practical framework that fills this omission.

Our tool of choice to disambiguate pointers with offsets is *Pentagons*, an abstract domain invented by Logozzo and Fähndrich to infer symbolic bounds to integer variables [Logozzo and Fähndrich (2008); Logozzo and Fähndrich (2010)]. This abstract domain is formed by the combination of two lattices. The first lattice is the *integer interval domain* [Cousot and Cousot (1977)], which maps integer variables to an approximation of the range of values that they can assume throughout the program's execution. The second lattice is the *strict upper bound domain*, which maps each variable v to $\mathsf{LT}(v)$, a set of other variables with the following property: if $u \in \mathsf{LT}(v)$, at a given program point p, then $u < v$ at p.

In this paper, we show that the key idea behind the Pentagon domain: a cheap and relatively precise "less-than" check, is effective to distinguish memory locations at compilation time. However, we had to adapt the definition of the abstract domain in non-trivial ways instead of simply reusing it. Firstly, we have added new operations to the abstract domain, that came from the necessity to separate variables used as pointers from variables used as integers. Secondly, we have adapted the original transfer functions and the control flow graph statements into simpler structures, which enable the information to be stored in a more economic way (in terms of space), and to

be run in less time. The output is a *sparse Pentagon Analysis*. A sparse analysis associates information with variable names. By information we mean the product 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 $O(|V|)$ objects to less-than relations – each object representing a single variable name. On the other hand, the original description of Pentagons maps $O(|V| \times |P|)$ objects, formed by variable names and program points, to less-than relations.

Our apparatus lets us perform different pointer disambiguation checks. As we explain in Section 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 reference overlapping memory locations if one of three conditions is met. First, there will be no aliasing if p_1 and p_2 point to different abstract objects created at separate program sites. This check is what the compiler literature typically calls *alias analysis*. The other two checks give us the directions along which our work departs from previous state-of-the-art, because we analyze pointers that reference parts of the same abstract object. Our second check tells that p_1 and p_2 cannot alias if either $p_1 < p_2$ or $p_2 < p_1$, whenever these two variables are simultaneously alive in the program. Notice that the inequality $p_1 < p_2$ means that the address pointed by p_1, i.e., the contents of the variable p_1, is less than the address pointed by p_2. To solve this inequality, we use the "less-than" domain of Pentagons. Finally, our third check says that p_1 and p_2 refer to distinct chunks of memory if these chunks comprise ranges of addresses that never overlap at any moment during the execution of the program. To demonstrate this fact, we rely on the "range" domain of Pentagons.

To validate our ideas, we have implemented them in LLVM [Lattner and Adve (2004)]. Our new alias analysis increases the ability of this compiler to disambiguate pointers by a significant margin. As an example, we increase by almost 2.5x the number of pairs of pointers that we distinguish in SPEC's `hmmer` and `bzip2`. In SPEC's `lbm`, this growth is almost 4-fold. 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.

2. Overview

The algorithms in Figure 1 shall motivate the need for a new points-to analysis. These two C programs make heavy use of arrays. 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 that are built on top of the work of Andersen [Andersen (1994)] or Steensgaard [Steensgaard (1996)], can distinguish pointers that dereference different memory blocks. However, they do not say much about references to the same array.

```
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.

A more precise alias analysis brings many advantages to programming language tools, including 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 [Surendran et al. (2014)] inter-iteration scalar replacement on the insertion sort algorithm shown 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 speed up 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.

```
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: Scalar replacement applied on Figure 1a. This kind of optimization is enabled by more precise alias analyses.

There exist points-to analyses designed specifically to deal with pointer arithmetics [Balakrishnan and Reps (2004); van Engelen et al. (2004); Paisante et al. (2016); Rugina and Rinard (2000)]. Nevertheless, none of them works satisfactorily for the two examples seen in Figure 1 and thus fail to enable the optimization in Figure 2. The reason for this ineffectiveness lies 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 [Cousot and Cousot (1977)], once applied on Figure 1a, will conclude that `i` exists on the interval $[0, \mathtt{N} - 2]$, and that `j` exists on the interval $[1, \mathtt{N} - 1]$. 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 techniques from this paper disambiguate `v[i]` and `v[j]` in both examples. Key to this ability is the observation that $\mathtt{i} < \mathtt{j}$ at every program point where we access `v`. We conclude that $\mathtt{i} < \mathtt{j}$ 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 $\mathtt{i} < \mathtt{j}$ because of the way that `j` is initialized, within the for statement at line 4. In Figure 1b, we know that $\mathtt{i} < \mathtt{j}$ due to the conditional check at line 7.

The design space of constructing less-than relations between program variables is large. In this paper, we chose to use the Pentagon lattice to

build such relations. However, a straightforward application of Pentagons, as originally defined by Logozzo and Fähndrich, would not be able to deduce the disambiguation between `v[i]` and `v[j]`. 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 does it.

In terms of implementation, our approach focuses on "top-level variables": variables that we can represent in Static Single Assignment (SSA) form [Alpern et al. (1988); Cytron et al. (1989)]. In this representation, each variable has a single definition site, and special instructions, the ϕ-functions, join the live ranges of names that, in the original program, represent the same variable. Compilers such as `gcc`, `icc` and `clang` represent pointers with offsets in this format. However, memory itself, i.e., regions pointed by pointers, are not in the SSA representation. Thus, the developments that we shall introduce in this paper do not disambiguate pointers to pointers: this problem pertains to the domain of standard alias analyses, à la Andersen [Andersen (1994)]. This fact does not compromise our implementation in any way: range and inequality checks are only used to disambiguate pointers that belong into the same memory unit, such as an array or a C-like struct.

This Paper in Five Examples. This section has explained the need for techniques able to disambiguate pointers with offsets. The rest of this paper describes an implementation of such a technique. Our implementation involves several steps. Throughout the paper, we use five examples to illustrate each of these steps, when applied onto the program in Figure 1a. The first step consists in converting the program to a suitable intermediate representation. Example 1 illustrates this step for function ins_sort. From this representation we extract a data-structure called a *Program Dependence Graph*, as shown in Example 3. The main purpose of this graph is to give us the means to disambiguate non related pointers and to enable comparing related ones using the less-than relations. Such process is the subject of Example 4. The solution of this constraint system gives us, for each variable v, a set of other variables that are known to be less than v. Example 9 shows the sets that we obtain for Figure 1a's routine. Finally, we have different ways to perform queries on these "less-than" sets. Example 11 clarifies this usage of our analysis.

3. Program representation for sparsity and range pre-analysis

In this paper we shall work at a low-level representation of programs. The syntax of this assembly-like language is given in Figure 3. The figure also gives, informally, the semantics of each instruction. Throughout the paper, we shall return to the high-level C notation whenever necessary to explain examples; however, our formalization will happen onto the syntax in Figure 3. Most of that syntax bears an intuitive semantics, except for conditionals and ϕ-functions, which we use to make our analysis sparse, as we discuss in Section 3.1. The instruction $p_0 = \text{malloc}(i_0)$ creates a block of i_0 bytes of memory, and assigns the address of the first byte to pointer p_0. A load such as $v = *p$ reads the contents of the memory cell pointed by p, and assigns that value to v. The store instruction $*p = v$ puts the value of variable v into the address pointed by p. The notation $v_0 = \phi(v_1 : \ell_1, v_2 : \ell_2)$ assigns either v_1 or v_2 to v_0, depending on the program flow reaching this instruction through ℓ_1 or ℓ_2, respectively. The syntax of branches is more convoluted, because it lets us infer "less-than" facts about the variables used in the conditional test. We shall provide more details about this syntax in the next section.

Prog	::=	I^*	; Program
I	::=		; Instruction
	|	$p_0 = \text{malloc}(i_0)$	; Memory allocation
	|	$v_0 = v_1 + i_0$	; Addition
	|	$v = *p$	; Load
	|	$*p = v$	; Store
	|	$v_0 = \phi(v_1 : \ell_1, v_2 : \ell_2)$	; Phi-function (*Phi*) (See [Cytron et al. (1989)])
	|	$(v_0 \mathrel{\mathscr{R}} v_1)\ ?\ S^*B\ S^*B$	; Branch ($\mathscr{R} \in \{<, \leq, =, \neq, >, \geq\}$)
B	::=	$\ell : Phi^* I^*$	; Basic Block
S	::=	$v_0 = \sigma(v_1)$	; Sigma-function (See [Singer (2006)])

Figure 3: The syntax of our language of pointers. Whenever a variable must be explicitly a pointer, we name it p. Variables that need to be integers are named i. Variables that can be either an integer or a pointer are named v. We use the Kleene Star (*) to indicate zero or more repetitions of a non-terminal.

3.1. Sparse Analysis

The implementation that we discuss in this paper is a *sparse analysis*. According to Tavares *et al.* [Tavares et al. (2014)], 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 sparsify 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, which ensures that each variable has at most one definition point.

The SSA format is not enough to ensure sparsity to the analysis that we propose in this paper. Therefore, we use a different flavor of this program representation: the *Extended Static Single Assignment form* [Bodik et al. (2000)] (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[1]. 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 1. *Figure 4 shows the* Control Flow Graph *(CFG) of the program seen in Figure 1a, in e-SSA form. The σ-functions rename every variable used in a comparison. Renaming lets us, for instance, infer that $x_{iT} > x_{jT}$,*

[1]The original description of e-SSA form uses the name π-function. We have adopted the name σ-functions, instead, following the work of Jeremy Singer on Static Single Information (SSI) form [Singer (2006)]. Both notations define live range splitting instructions, which work for the same purpose

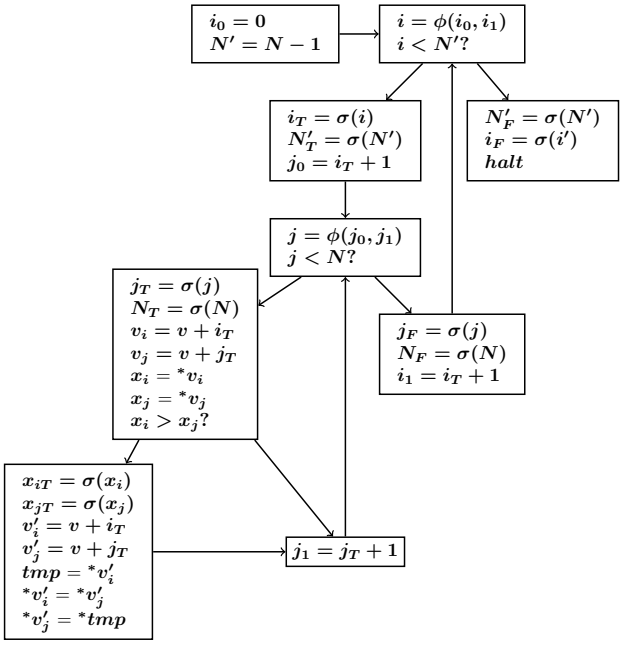


Figure 4: Control Flow Graph of program in Figure 1a. $i_T = \sigma(i)$ denotes the "true" version of i after the test $i < N$. $i = \phi(i_0, i_1)$ denotes the value of i inside the first loop, its value is i_0 if the flow comes from the first block, or i_1 if the flow comes from the returning edge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

because: (i) these variables are copies of x_i and x_j; and (ii) they only exist in the true side of the test $x_i > x_j$.

3.2. Range Analysis for integer variables

A core component of our pointer disambiguation method is a *range analysis* for scalar variables, 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 [Alves et al. (2015); Blume and Eigenmann (1994); Nazaré et al. (2014); Rodrigues et al. (2013); Rugina and Rinard (2005)]. In this paper, we have adopted a non-relational range analysis [Rodrigues et al. (2013)] on the classic integer interval [Cousot and Cousot (1977)]. A non-relational range analysis associates variable names with ranges that are not function of other variables. 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 [Miné (2006)]. Our approach to build this range analysis 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 2. *Considering the program in Figure 4, the analysis of Blume* et al. *[Blume and Eigenmann (1994)] finds the following ranges for the variables in that program:* $R(i_0) = [0, 0], R(i_1) = [1, N-1], R(i) = [0, N-1], R(j_0) = [1, N-1], R(j_1) = [2, N], R(j) = [1, N]$. *The ranges of integer variables given by Rodrigues* et al. *[Rodrigues et al. (2013)] 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, as we will see in Section 5.3. However, precision has an impact on the runtime of the analysis. 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 [Rodrigues et al. (2013)].

4. Pointer Disambiguation Based on Strict Inequalities

The technique that we introduce in this paper to disambiguate pointers is semi-relational, i.e., it maps variables to ranges that might contain other variables as symbolic limits; however, contrary to relational analysis, it does not map groups of variables to abstract states. Our analysis 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 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 1 touches several concepts pertaining to the vocabulary 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 exists 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 (Nielson et al., 2005, Sec-2.1.4).

We call the alias analysis approach we describe in this paper "staged", because it works in successive stages. It consists of the following four parts, which we describe in the rest of the paper:

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

4.1. Grouping Pointers in Pointer Digraphs

Recalling Definition 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*[2]. A base pointer is a reference to the beginning of a memory block (address zero). 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 Dependence 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 3.2. 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 gives us only one allocation site, even if the loop iterates many times. Such a data-structure

[2]The ISO C Standard forbids relational comparisons between pointers that refer to different allocation blocks, even if said pointers have the same type (ISO, 9899:2011, Sec-6.5.8p5).

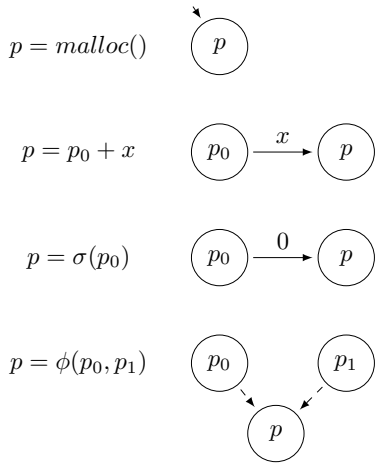


Figure 5: Rules to generate the pointer dependence digraph.

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 variables 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 an SSA-form program, the only way a variable can update itself is through a ϕ-function.

Example 3. *The rules seen in Figure 5, once applied onto the control flow graph given in Figure 4, give us the PDD shown in Figure 6.*

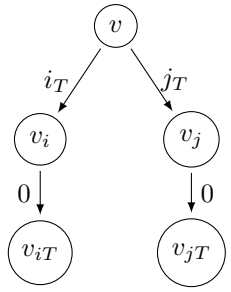


Figure 6: Pointer dependence graph of program in Figure 1a.

4.2. Constraint System

We solve the less-than analysis via a constraint system. This constraint system is formed by five different kinds of constraints, which involve two variables, and a relational operator. Valid relational operators are $\mathscr{R} \in \{=, <\leq, \geq, >\}$, each one giving origin to one of the five types of constraints. If *Prog* is a program formed according to the syntax seen in Figure 3, and $\{v_1, v_2\} \in Prog$ are two variables, then we let $v_1 \mathscr{R} v_2$ be a constraint. We use the same names to denote program variables and constraint variables. Program variables are elements that exist in the syntax of *Prog*; constraint variables are symbols, which, as we shall see in Section 4.4, are mapped to "less-than" information. Because it is always possible to know which entity we refer to, given the context of the discussion, we shall not use any notation to distinguish them.

To explain the semantics of constraints, we describe the semantics of programs. A program *Prog* is formed by instructions, which operate on states $\sigma : Var \mapsto Int$. We say that a variable v is *defined* at a state σ, if $\sigma(v) = n$; otherwise, the variable is undefined at that state, a fact that we denote by $\sigma(v) = \bot$. Constraints determine relations between defined variables, given a state σ. Definition 2 express this notion.

Definition 2 (Intuitive Semantics). *We say that a constraint $v_1 \mathscr{R} v_2$ satisfies a state σ if $\sigma(v_1) \mathscr{R} \sigma(v_2)$ whenever v_1 and v_2 are defined for σ.*

4.3. Collecting Constraints

During this phase, we collect a set C of constraints according to the rules in Figure 7. Recall that R(v) denotes the numeric range of variable v given by a pre-analysis. We let $R(\mathtt{v})_\downarrow$ and $R(\mathtt{v})_\uparrow$ be the lower and upper bounds of interval R(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` lets us derive the fact $\mathtt{p} > \mathtt{q}$, if the range analysis of Section 3.2 is capable of proving that v is always strictly greater than zero. The "=" equality models set inclusion, and is asymmetric as we shall explain in details in Section 4.4.

Constraints are simple, the syntax of the constraints is self-explanatory using standard semantics ; however, some comments follow.

$$
\begin{array}{rcl}
\mathit{Initialization} & \Rightarrow & \mathsf{C} = \emptyset \\
\\
\mathit{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} \mathit{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} \\
\\
\mathit{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} \\
\\
\mathit{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} \\
\\
\mathit{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 & \mathit{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.

Firstly, the difference between *add* and *gep*[3] 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. Therefore, to avoid comparing a pointer to an

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

integer variable, the constraints that we produce for arithmetic operations involving pointers are different from those we produce for similar instructions that involve only integers, as Figure 7 shows.

A second aspect of our constraint system, which is worth mentioning, is the fact that we do not try to track less-than information throughout pointers. Hence, as Figure 7 shows, we do not generate constraints for loads and stores, which means that we do not build less-than relations for data stored in memory. Consequently, our current implementation of the less-than lattice misses opportunities to disambiguate *second-order* pointers, i.e., pointers to pointers. This omission is not a limitation of the theory that we present in this paper. Rather, it is an implementation decision, which we took to simplify the design of our algorithm. Handling pointers to pointers is possible in different ways. For instance, we can use some pre-analysis, à la Andersen [Andersen (1994)] or à la Steensgaard [Steensgaard (1996)], to group memory references into single locations. After this bootstrapping phase, we can treat these memory locations as variables, and extract constraints for them using the rules in Figure 7.

The construction of constraints for tests is more involved. The e-SSA form, which we have discussed in Section 4, provides us explicit new versions of variables, e.g.: $\mathtt{p}_\mathtt{T}, \mathtt{q}_\mathtt{T}, \mathtt{p}_\mathtt{F}, \mathtt{q}_\mathtt{F}$, 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: $\mathtt{p}_\mathtt{T} < \mathtt{q}_\mathtt{T}$ and $\mathtt{p}_\mathtt{F} \geq \mathtt{q}_\mathtt{F}$.

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

The Relation between PDDs and Constraints. The purpose of the PDDs of Section 4.1 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 only used 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 4.1 and in this section are independent, and can be performed in any order.

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

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

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

Figure 8: Rules to solve constraints.

4.4. Solving Constraints

The objective of this phase is to obtain Pentagon-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 *strictly* less than $\mathtt{v}$. In addition to LT, we build an auxiliary set GT. Set $\mathsf{GT}(\mathtt{v})$ keeps track of all the variables that are *strictly* greater than $\mathtt{v}$. 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:

$$\mathsf{GT}(v) = \{v_i \mid v \in \mathsf{LT}(v_i)\} \;\wedge\; \mathsf{LT}(v) = \{v_i \mid v \in \mathsf{GT}(v_i)\}$$

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 fixpoint. The non-reflexive equal is used to model relations that are known to be true before sigma nodes. A given relational information which is true about variables before a conditional branch continues to hold also in the "then" and "else" paths that sprout from that branch. Example 5 illustrates this fact:

Example 5. *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*

$\mathtt{x_F} < \mathtt{y}$*), where* $\mathtt{x_T}$ *is the new name of* $\mathtt{x}$ *in the "then" branch of the conditional, and* $\mathtt{x_F}$ *is the new name of* $\mathtt{x}$ *in the "else" branch.*

Dealing with ϕ-functions. Join nodes are program points that denote loops and conditional branches. 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 $\{v = \phi(v_i)\}, 1 \leq i \leq 2$. When used in the head of loops, ϕ-functions may join variables that represent initialization values and loop variant values for a given variable v. To build a less-than relation between these variables, 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. If v only decreases, then it will be less than or equal to the maximum among its initialization values. In the opposite case, in which v only increases, the variable produced by the ϕ-function will be greater than or equal to the minimum among its initialization values.

To infer relations between the variable defined by a ϕ-function, and the variables used as arguments of that ϕ-function, we perform a *growth check*. On the resolution of each join constraint such as $x = \phi(x_1, \ldots, x_n)$, we check if x is present in the LT or GT sets of some x_i which is loop variant. If x is present in the LT of each loop-variant right-side operand, then the program contains only execution paths in which x grows. That is to say that x is always receiving a value that is greater than itself. Dually, if x is present in the GT of each loop-variant x_i, then there exist only paths in the program along which x decreases.

In order to detect if a given x_i is loop-variant, we use the dominance information: if the definition of a ϕ-function, e.g., x, dominates the definition point of one of its arguments, e.g., x_i, then we know that this ϕ-function occurs in a loop, and we say that x_i is loop-variant. This observation holds for the so called *natural loops*, which characterize *reducible control flow graphs*. For the definition of these concepts, we refer the reader to the work of Ferrante *et al.* [Ferrante et al. (1987)]. Example 6 illustrates these observations.

Example 6. *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. Both ϕ-functions exist in natural loops.*

Figure 10 illustrates the use of constraints imposed by ϕ-functions. We use *conditional constraints* to deal with them. A conditional constraint is

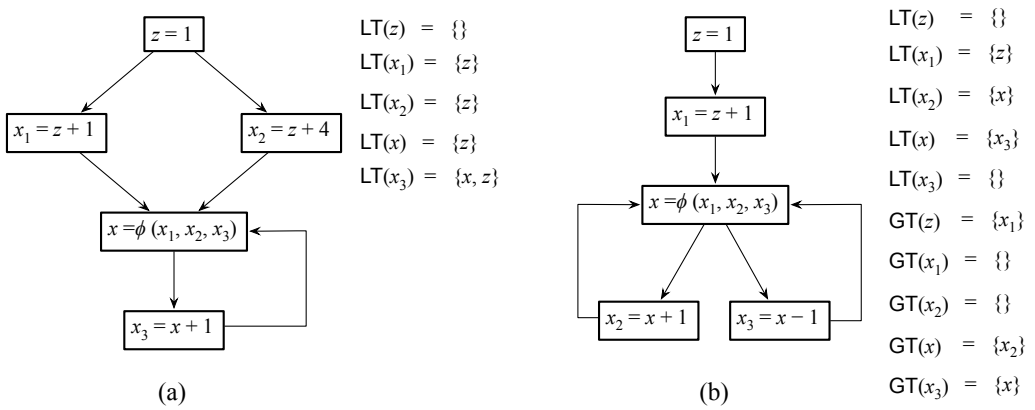


Figure 9: Less-than and greater-than sets built by our analysis.

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 and GT sets contain specific variables. Example 7 explains how we compute the less-than set of x.

$$join : x = \phi(x_i) \quad 0 \leqslant i \leqslant n \quad \Rightarrow \begin{cases} i : A \Rightarrow \mathsf{LT}(x) \cup = \bigcap_{x_j \notin Dom(x)} \mathsf{LT}(x_j) \\ ii : B \Rightarrow \mathsf{GT}(x) \cup = \bigcap_{x_j \notin Dom(x)} \mathsf{GT}(x_j) \\ iii : \text{Otherwise} \Rightarrow \begin{cases} \mathsf{LT}(x) \cup = \bigcap_{i=0}^{n} \mathsf{LT}(x_i) \\ \mathsf{GT}(x) \cup = \bigcap_{i=0}^{n} \mathsf{GT}(x_i) \end{cases} \end{cases}$$

Figure 10: Solving constraints for ϕ-functions. We let $x' \in \{x, x_T, x_F\}$ and $Dom(x) = \{x_i \text{ s.th. } x \text{ dominates } x_i\}$, $A \equiv \forall x_j \in Dom(x),\ x' \in \mathsf{LT}(x_j)$ and $B \equiv \forall x_j \in Dom(x), x' \in \mathsf{GT}(x_j)$. A indicates that x increases in the loop, and B indicates that x decreases. Variables x_T and x_F are new names of x created by σ-functions after conditional branches.

Example 7. *Variable x, defined in Figure 9 (a) has increasing value, because it belongs into $\mathsf{LT}(x_3)$ and $Dom(x) = \{x_3\}$. 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* (Nielson et al., 2005, 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. Upon popping a constraint such as $x < y$, we verify if its resolution changes the current state of LT sets. If it does, then we insert back into the worklist every constraint that depends on y, i.e., that reads $\mathsf{LT}(y)$. 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 7. The process of constraint resolution is guaranteed to terminate as shown in Theorem 1. Example 8 illustrates this approach.

Example 8. *We let* $\mathsf{C} = \{x < y, y < z\}$. *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 fixpoint 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	$pop(WL_i)$	$\mathsf{LT}(x)$	$\mathsf{LT}(y)$	$\mathsf{LT}(z)$
$\{x < y, y < z\}$	–	$\emptyset$	$\emptyset$	$\emptyset$
$\{x < y, y < z\}$	$x < y$	$\emptyset$	$\{x\}$	$\emptyset$
$\{y < z, y < z\}$	$y < z$	$\emptyset$	$\{x\}$	$\{y, x\}$
$\{y < z\}$	$y < z$	$\emptyset$	$\{x\}$	$\{y, x\}$
$\{\}$	–	$\emptyset$	$\{x\}$	$\{y, x\}$

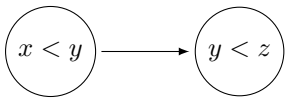


Figure 11: Resolution of the constraints produced in Example 8. The order in which we solve constraints is dictated by their dependences in the CDG shown on the right. Each line depicts one step of the algorithm. WL_i shows the worklist before we pop it.

Theorem 1 (Termination). *Constraint resolution always terminates.*

> **Proof:**[4] The worklist based solver reaches a fixpoint, 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 9. *Example 4 showed us the constraints that we produce for the program in Figure 4. The solution that we find for that constraint system 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.*

Valid Solutions. We say that LT is a valid solution for a constraint system C, which models a program *Prog* if it meets the requirements in Definition 3. Program states, i.e., σ, are defined in Section 4.2.

Definition 3 (Valid Solution). *We say that* $\mathsf{LT} \models \sigma$ *if, for any* v_1 *and* v_2*, well-defined at* σ*, we have that:* $\mathsf{LT} \models \sigma \wedge v_1 \in \mathsf{LT}(v_2) \Rightarrow \sigma(v_1) < \sigma(v_2)$

Theorem 2 states that the solution that we find for a constraint system is valid. The proof of the theorem relies on the semantics of instructions. Each instruction I transforms a state σ into a new state σ'. We shall not provide transition rules for these instructions, because they have an obvious semantics, which has been described elsewhere [(Nazaré et al., 2014, Fig.3)].

Theorem 2 (Correctness of the Strict Less-Than Relations). *Our algorithm produces valid solutions to the constraint system.*

[4]We have not mechanized any of the proofs we present in this paper; hence, they remain mostly informal. However, we believe that the intuition they provide helps readers to understand the essential components of our technique.

Proof: The proof of Theorem 2 consists in a case analysis on the different constraints that can create the relation $v_1 \in \mathsf{LT}(v_2)$ in Figure 8 and 10. 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* $x < v_2$ updates the strict less than set of v_2. $\mathsf{LT}(v_2) = \mathsf{LT}(v_2) \cup \mathsf{LT}(x) \cup \{x\}$. For the sake of clarity, we let $\mathsf{LT}'(v_2)$ be the strict less than set of v_2 before the update introduced by the constraint $x < v_2$. Henceforth, $v_1 \in \mathsf{LT}(v_2)$ if $v_1 \in \mathsf{LT}'(v_2)$ or $v_1 \in \mathsf{LT}(x)$, or $v_1 = x$. If $v_1 = x$, then we are done, due to the constraint $x < v_2$. Otherwise, if $v_1 \in \mathsf{LT}(x)$, by induction, $v_1 < x$, and by transitivity we get $v_1 < v_2$. In case $v_1 \in \mathsf{LT}'(v_2)$, by induction on $\mathsf{LT}'(v_2), v_1 < v_2$. In fact, since the strict less than set of v_2 is only growing up if updated, then if a property holds for its members once, it holds even after updates. In other words, $\mathsf{LT}'(v_2)$ may be an update of $\mathsf{LT}''(v_2)$ in which we have inserted v_1 after resolving a constraint. In this case, $\mathsf{LT}''(v_2) \subseteq \mathsf{LT}'(v_2) \subseteq \mathsf{LT}(v_2)$ with $v_1 \in \mathsf{LT}''(v_2)$. In the remaining of the proof, we shall be interested only on elements updating $\mathsf{LT}(v_2)$.
- The constraint *non reflexive equal* $v_2 = x$, updates $\mathsf{LT}(v_2)$ with $\mathsf{LT}(x)$: $\mathsf{LT}(v_2) = \mathsf{LT}(v_2) \cup \mathsf{LT}(x)$. We focus on $v_1 \in \mathsf{LT}(x)$. By induction, we get $v_1 < x$. From Figure 7 we know that the constraint comes from a test. Thus, w.l.o.g, we can assume that this constraint is $x_1^T = x$, coming from a test $x < x^2$. The condition $v_1 < x$ 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 $v_1 < x_1^T = v_2$.
- The constraint $x = \phi(x_i)$. We show the proof for the first case of Figure 10, the second one is similar and the third one is straightforward. In this case we have $\forall x_j \in Dom(x),\ x' \in \mathsf{LT}(x_j)$ implying $\mathsf{LT}(x) \cup = \bigcap_{x_j \notin Dom(x)} \mathsf{LT}(x_j)$. $x' \in \{x, x_T, x_F\}$. In the general case we would have: $\mathsf{LT}(x) \cup = \bigcap_{i=0}^{n} \mathsf{LT}(x_i)$ that we can write

$$\mathsf{LT}(x) \cup = \left(\bigcap_{x_j \notin Dom(x)} \mathsf{LT}(x_j)\right) \bigcup \left(\bigcap_{x_j \in Dom(x)} \mathsf{LT}(x_j)\right)$$

since x may take any of x_i values. We want to prove that if

$y \in \bigcap_{x_j \notin Dom(x)} \mathsf{LT}(x_j)$, then $y < x$ under condition A. We let $x_{j_0} \in Dom(x)$ such that $x' \in \mathsf{LT}(x_{j_0})$. This means that x_{j_0} is defined as follows: $x_{j_0} = x' + a; a > 0$. Hence, there exists a redefinition x'_i of x among ϕ operands such that $\langle x = x'_i \rangle$ and $x_{j_0} = x'_i + a$. Since $a > 0$ we obtain $x'_i < x_{j_0}$ with two possible situations for x'_i:

- $x'_i \in Dom(x)$, then $x' \in \mathsf{LT}(x'_i)$ which gives again $\exists x''_i$ such that $x''_i < x'_i < x_{j_0}$.
- $x'_i \notin Dom(x)$, then $y \in \mathsf{LT}(x'_i)$. As there cannot be any infinite descending sequence of ϕ redefinitions, all instances of variable x are greater than x'_i. By induction on LT, $y \in \mathsf{LT}(x'_i)$ gives $y < x_i$ and therefore $y < x$.

5. Answering Alias Queries

The final product of the techniques discussed in the previous section, is a table that associates each variable v with a set $\mathsf{LT}(v)$ of other variables that are less than v. This table gives us the means to 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. We define a *query* as the process of applying such test on a given pair of pointers.

Figure 12 illustrates the relationship between the three tests that we use to answer queries. If we cannot conclude that two pointers always refer to distinct regions, then we say – conservatively – that they *may alias*. Currently, we do not use our infra-structure to prove that two pointers *must alias* each other.

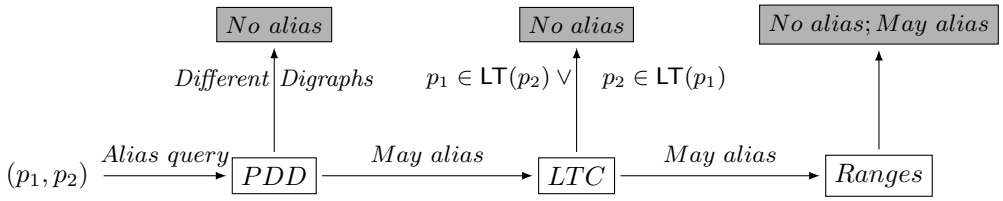


Figure 12: Steps used in the resolution of pointer disambiguation queries.

The three pointer disambiguation checks that Figure 12 outlines are complementary and definitive. By complementary, we mean that none of them

subsumes the others. By definitive, we mean that they all are conservatively safe: if any of these checks reports that pointers p_1 and p_2 do not alias, then this information, regardless of the results produced by the other checks, is enough to ensure that p_1 and p_2 refer to different locations. 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 dependence digraphs, then they are said to be unrelated. PDDs are described in Section 4.1.

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.

5.1. The Digraph Test

We have seen, in Section 4.1, 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 10 illustrates this observation.

Example 10. *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 dependence 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 }
```

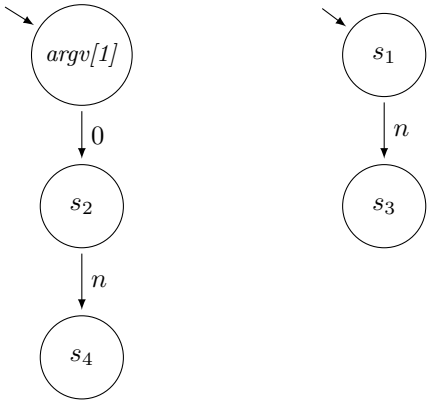


Figure 13: Program that contains two different memory locations, which will give origin to two unconnected pointer dependence digraphs. For the sake of brevity, the program is kept in high level language.

We do not track abstract information that flows through memory locations (cf. Section 4.3). In other words, we do not generate constraints for information that comes out of load and store instructions. Similarly, our current implementation of the Digraph Test also does not consider pointers to pointers when building PDDs. Thus, our PDDs contain vertices that represent only top-level variables, e.g., variables that our baseline compiler, LLVM, represents in Static Single Assignment form. The consequence of this implementation decision is that currently we do not disambiguate pointers `p1` and `p2` in the following code snippet: `{p1 = v + 4; *x = v; p2 = *x + 8;}`, because we forgot the fact that `v` and `*x` represent the same memory location.

5.2. 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 dependence digraph. From Theorem 2, 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 11 illustrates the application of this test.

Example 11. *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 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.*

5.3. The Ranges Test

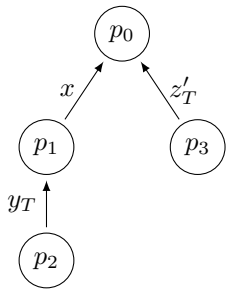
The so called *Ranges test* is a byproduct of the key components of the previous PDD test. This test consists in determining an expression of the range of intervals covered by two pointers, p_1 and p_2, which share the same pointer digraph. If these two ranges do not intersect, then we can conclude that 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 dependence 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* [Paisante et al. (2016)] to disambiguate pointers: if two pointers cover non-overlapping ranges, then they cannot 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 12 shows how the range test works concretely.

Example 12. *Consider the program in Figure14a. Our goal in this example is to disambiguate pointers* p_2 *and* p_3*. We have that* $LT(p_0) = \emptyset, LT(p_1) = \{p_0\}, LT(p_2) = \{p_0, p_1\}$ *and* $LT(p_3) = \{p_0\}$*.* $p_2 \notin LT(p_3)$ *and* $p_3 \notin LT(p_2)$*. The simple less than check is not able to disambiguate these pointers; hence, we resort to the range test. The dependence graph of Figure14a reveals that the lowest common ancestor of* p_3 *and* p_2 *is* p_0*.* $\mathcal{P}(p_2, p_0) = (p_2, p_1, p_0)$ $\mathcal{P}(p_3, p_0) = (p_3, p_0)$*. 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.*

```
1  void pdd_ex(int N){
2    if (N>0){
3      int *p0=malloc(N);
4      int x = 1;
5      int y = (rand()%2) + 2;
6      int z= (rand()%3) + 5;
7      int* p1=p0+x;
8      int* p2=p1+y;
9      int* p3=p0+z;
10   }
11 }
```



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

Figure 14: Disambiguating pointers with PDD and Ranges

The impact of range analysis on the precision of our method. In Section 3.2 we emphasized that the more precise the range analysis for integer variables that we use, the more precise the pointer analysis that we produce. Let us now detail how the difference in precision affects our analysis of pointers. In Figure 15, we give a simple program where our goal is to disambiguate memory locations $p[i]$ and $p[i + 1]$. Suppose that we have renamed variable i to i_0 at line ℓ_3 and to i_1 at ℓ_4 and memory locations $p[i]$ and $p[i + 1]$ to respectively p_0 and p_1. We have $\mathcal{P}(p_0, p) = (p_0, p)$ and $\mathcal{P}(p_1, p) = (p_1, p)$. The re-writing algorithm gives us that $p_0 = p + i_0$ and $p_1 = p + i_1$.

The range analysis we use gives: $R(i_0) = [0, N - 1]$ and $R(i_1) = [1, N]$. These ranges are over-approximated and make i_0 and i_1 possibly have the same value. Therefore, pointers p_0 and p_1 may dereference overlapping memory regions which is clearly not the case. However, if we adopt a more

precise abstract interpretation to generate i_0 and i_1 possible values such as $\mathcal{A}(i_0) = [0, N-1] \ \cap \ 2x; x \in \mathbb{N}$ and $\mathcal{A}(i_1) = [1, N] \cap (2x+1); x \in \mathbb{N}$, then we can disambiguate p_0 and p_1 since $\mathcal{A}(i_0) \cap \mathcal{A}(i_1) = \emptyset$

```
1 void range(int N, int* p){
2   for(int i = 0; i < N ; i += 2){
3         p[i] = i;             // i0 = i; p0 = p + i0
4         p[i + 1] = i + 1; // i1 = i; p1 = p + i1
5   }
6 }
```

Figure 15: Program to show the effect of precision of integer analysis on pointer disambiguation.

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 13, 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 13. *The program in Figure 16 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 fail for the following reasons:*

- *Pointers* p_2 *and* p_3 *are derived from the same base pointer* p_0*; hence, they may alias due to Section 5.1's test.*
- *Neither* $p_3 \in \mathit{LT}(p_2)$ *nor* $p_2 \in \mathit{LT}(p_3)$*. Variables* y *and* z*,* p_2 *and* p_3*'offsets, are not compared since the pointers are not directly related to the same base pointer; hence,* p_2 *and* p_3 *may alias according to the less-than check of Section 5.2.*
- *The range test of Section 5.3 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 intervals 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 16: Our analysis reports that pointers p2 and p3 may alias.

6. 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. It runs in linear time on the number of program instructions ($O(i)$), because it traverses the code, verifying if each instruction defines constraints. The second step of the preprocessing phase consists in building a pointer dependence graph. 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, and is equivalent to the problem of building transitive closures of graphs. Its worst case complexity is $O(c^3 * v)$; c being the number of constraints and v the number of variables. Nevertheless, we have not observed this complexity empirically. The experiments that we perform in Section 7.2 will demonstrate that our algorithm runs in $O(c)$ in general. This lower complexity is justified by a simple observation: a program variable tends to interact with only a handful of other variables. Thus, the number of possible dependencies 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 has 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.

7. Evaluation

In this section, we discuss the empirical evaluation of our analysis and its implementation in the LLVM compiler, version 3.7.

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 increases the capacity of LLVM to disambiguate pointers; (ii) the asymptotic complexity of our algorithm is linear in practice and (iii) the three pointer disambiguation tests are useful.

7.1. Evaluation of 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[5]:

- 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.

Figure 17 shows how the LLVM's basic alias analysis, Andersen's inclusion-based analysis [And94], and our approach fare when applied on the integer programs in SPEC CPU2006 [Henning (2006)]. Henceforth, we shall call our analysis `sraa`, to distinguish it from LLVM's `basicaa` and Andersen's analysis called `CF` (because it uses context free languages (CFL) to model the inclusion-based resolution of constraints). Our algorithm `sraa` is implemented in LLVM 3.7.1. However, `CF` is distributed in LLVM 3.9.0. Thus, in Figure 17, `CF`'s numbers have been produced via LLVM 3.9.0. We emphasize that both versions of this compiler produce exactly the same number

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

Program	#Queries	%basicaa	%sraa	%(s + b)	%CF
473.astar	90686	40.68	52.15	62.14	X
445.gobmk	3556322	45.16	12.01	48.81	49.91
458.sjeng	528928	67.51	49.65	73.29	73.82
456.hmmer	2894984	8.55	19.60	22.86	53
464.h264ref	19809278	12.37	6.96	14.59	18.64
471.omnetpp	13987157	18.72	1.49	19.67	X
429.mcf	66687	10.70	31.56	34.37	14.05
462.libquantum	120479	39.41	30.27	53.04	42.16
401.bzip2	2485116	20.84	50.67	51.88	22.75
403.gcc	198350037	3.97	13.09	15.51	12.3
483.xalancbmk	11674531	15.35	27.40	32.41	X
400.perlbench	31129433	7.37	12.98	16.28	21.11
470.lbm	31193	3.45	40.17	41.70	3.81

Figure 17: Comparison between three different alias analyses. We let **s + b** be the combination of our technique and the *basic* alias analysis of LLVM. Numbers in **basicaa**, **sraa**, **s+b**, and **CF** show percentage of queries that answer "no-alias". The X sign indicates that the exhaustive evaluator could not run with Andersen's analysis and aborted.

of alias queries. 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 `basicaa` on `lbm` and `bzip2`. Moreover, we improve `basicaa`'s results considerably when both analyses run together on `gobmk`. Visual inspection of `lbm`'s code reveals a large number of hard-coded 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 lacks the capacity to benefit from the presence of constants in the target code to disambiguate pointers. LLVM 3.9.0 uses the basic alias analysis by default, in contrast to LLVM 3.7.1, which does not use any pointer disambiguation strategy by default. Therefore, to fairly compare against `CF`, we consider numbers that both analyses get when combined with `basicaa`. This experiment reveals that there is no clear winner between our analysis and `CF`. In terms of total number of pointers disambiguated, our analysis is slightly more precise than Andersen's. We did not combine the two analyses, because they do not run in the same version of LLVM. However, we expect our analysis to increase the precision of Andersen's, as the latter does not deal with pointers with offsets.

Figure 18 compares our analysis against `basicaa` in terms of the absolute number of queries that they can resolve. A *query* consists of a pair of point-

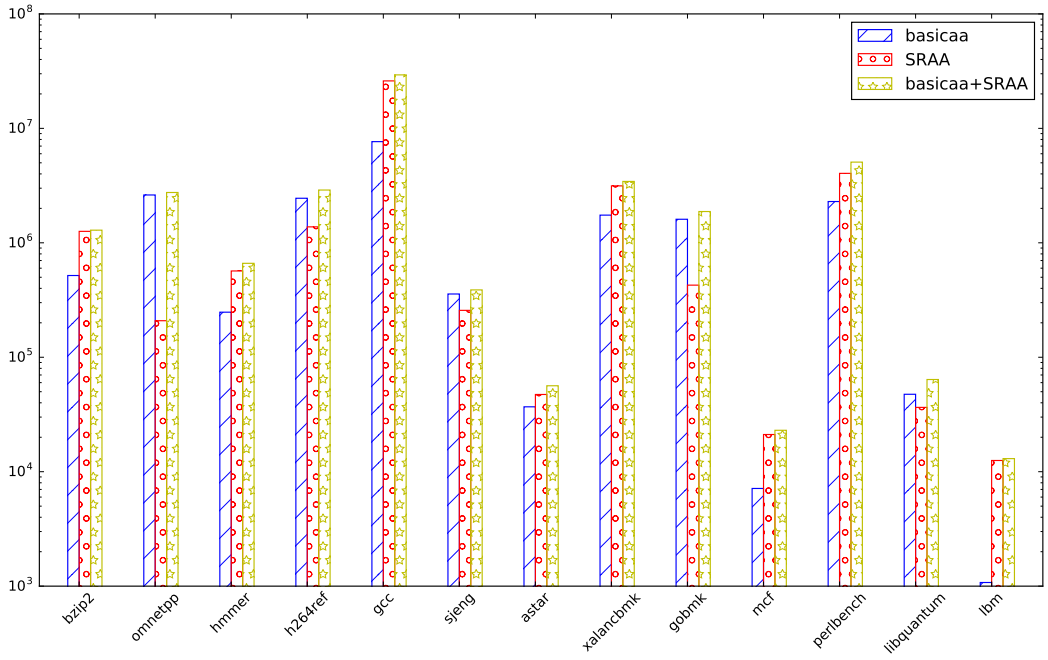


Figure 18: Comparison between SRAA and LLVM's basic alias analysis, showing how it increases LLVM's capacity to disambiguate pointers. The X-axis shows SPEC CPU2006 benchmarks. The Y-axis shows the number of queries answering no alias. The higher the bar, the better for the algorithm that we introduce in this paper.

ers. 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 no clear winner in this competition, because each analysis outperforms the other in some benchmarks. 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 19 shows the same comparison between `basicaa` and

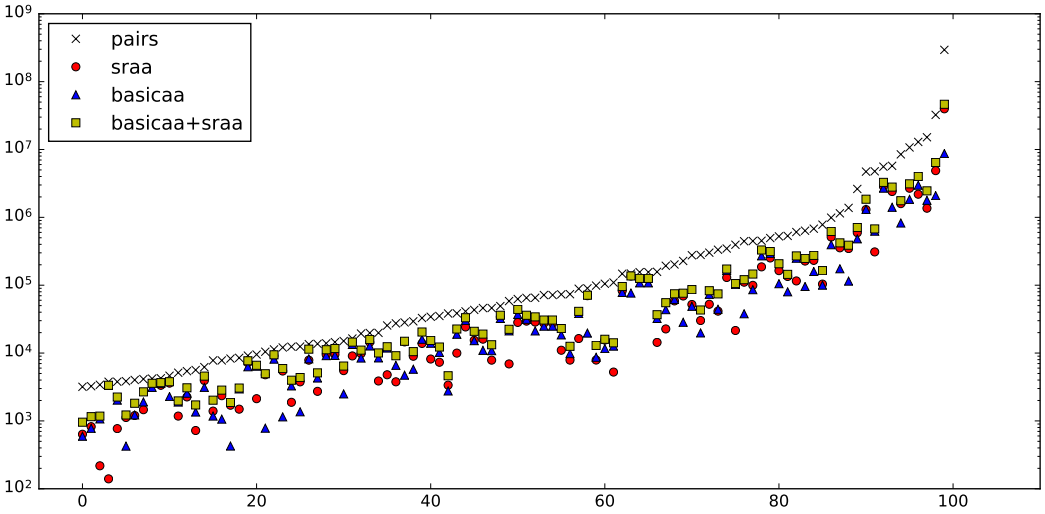


Figure 19: Effectiveness of our alias analysis (SRAA), when compared to LLVMs basic alias analysis on the 100 largest benchmarks in the LLVM test suite (TSCV removed). Each tick in the X-axis represents one benchmark. The Y-axis represents total number of queries (one query per pair of pointers), and number of queries in which each algorithm got a no-alias response.

`sraa`, this time on the 100 largest benchmarks available in the LLVM's test suite. Nevertheless, the results found in Figure 19 are similar to those found in Figure 18. 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 19, to show that, in general, the number of queries that we solve is proportional to the total number of queries found in each benchmark.

The Role of the Three Disambiguation Tests. Figure 20 shows how effective is each one of the three disambiguation tests that we have discussed in Section 5. In our staged approach, these tests work in succession: given a query q, first we use the PDD test of Section 5.1 to solve it. If this test fails, then we use the less-than test of Section 5.2 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 5.3. As Figure 20 shows, most of the queries can be answered by simply checking that different pointers belong into different pointer dependence digraphs (PDDs). We should remind the reader that, out of our three checks, only PDD can disambiguate pointers to pointers: the other two checks are applied only on top-level variables, e.g.,

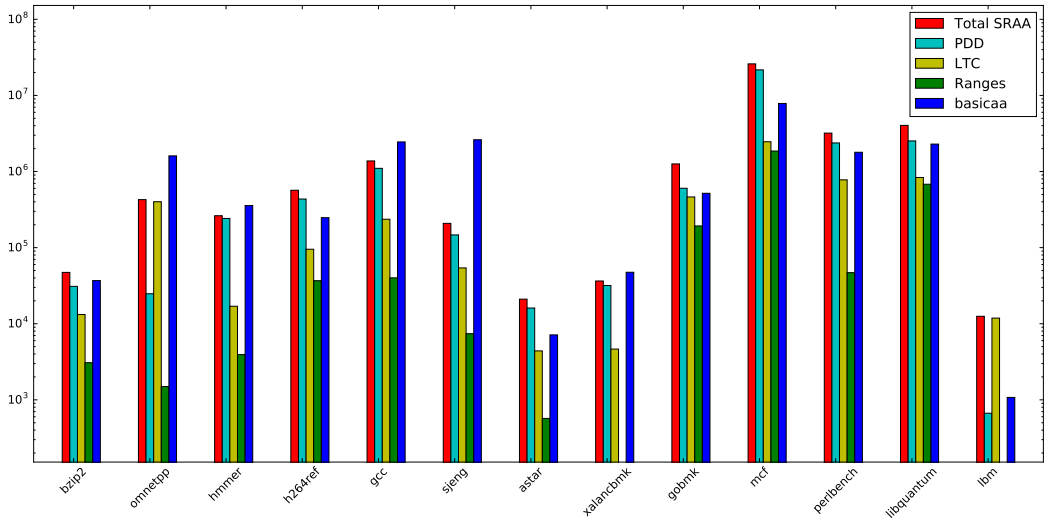


Figure 20: How the three tests in SRAA compete to disambiguate pairs of pointers. The Y-axis denotes the number of disambiguated pairs of pointers.

pointers allocated on the stack, which are represented in SSA form. Thus, it is expected that they will be less effective, as they have a more constrained data space to operate. Yet, the algebraic rules present in these two tests are essential in some benchmarks. In particular, the less-than check improves the PDD test by more than 30% on average, and in some cases, such as SPEC's `lbm`, it more than doubles its precision. In some cases, e.g., `astar`, `perlbench` and `lbm`, it is thanks to this test that we outperform `basicaa`.

Figure 20 shows that the range analysis test is not very effective when compared to the other two tests. This behavior is 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 21 shows the total number of queries solved by each test. In this experiment,

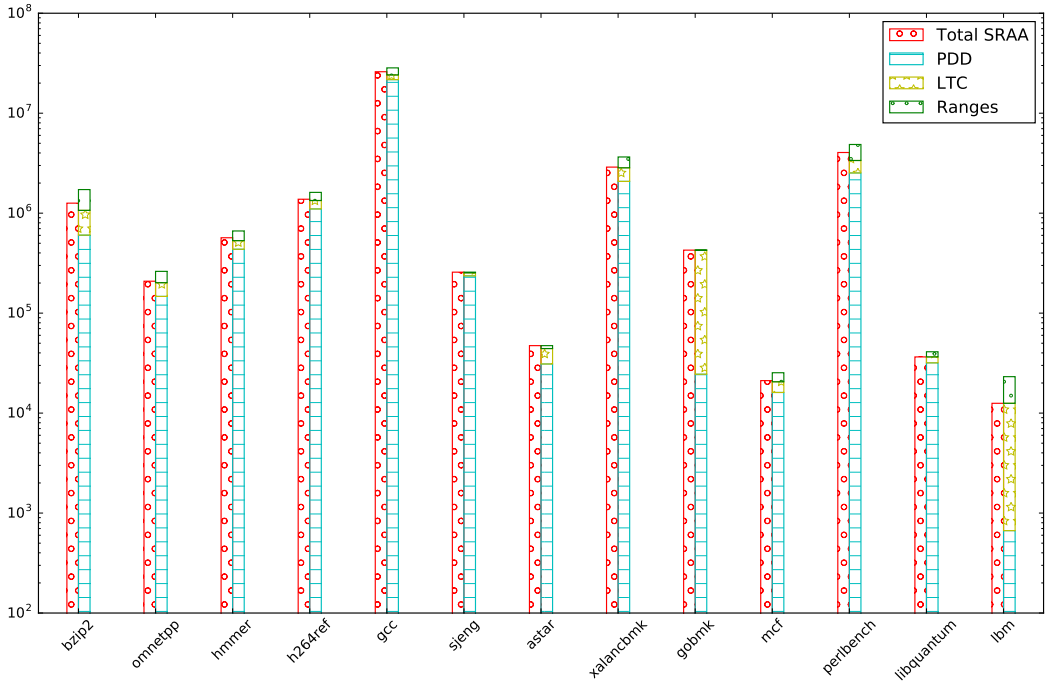


Figure 21: How three tests in SRAA *separately* compete to disambiguate pairs of pointers. The Y-axis denotes the number of disambiguated pairs of pointers.

we run each test independently; hence, the success of one resolution strategy does not prevent the others from being applied. Nevertheless, the range test of Section 5.3 is still the least effective of the three approaches that we have discussed in this paper.

7.2. Runtime experiments

Figure 22 shows the runtime of our alias analysis on the 100 largest benchmarks in the LLVM test suite. As the figure shows, we are able to run our three tests for all the benchmarks, but eleven, in less than one second. Nevertheless, we observe a linear behavior. In Figure 22, 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 23 discriminates the runtime of each disambiguation test. Our largest benchmark, `403.gcc`, took about 200 seconds to finish. Such long time happens because, in the process of solving constraints, we build the

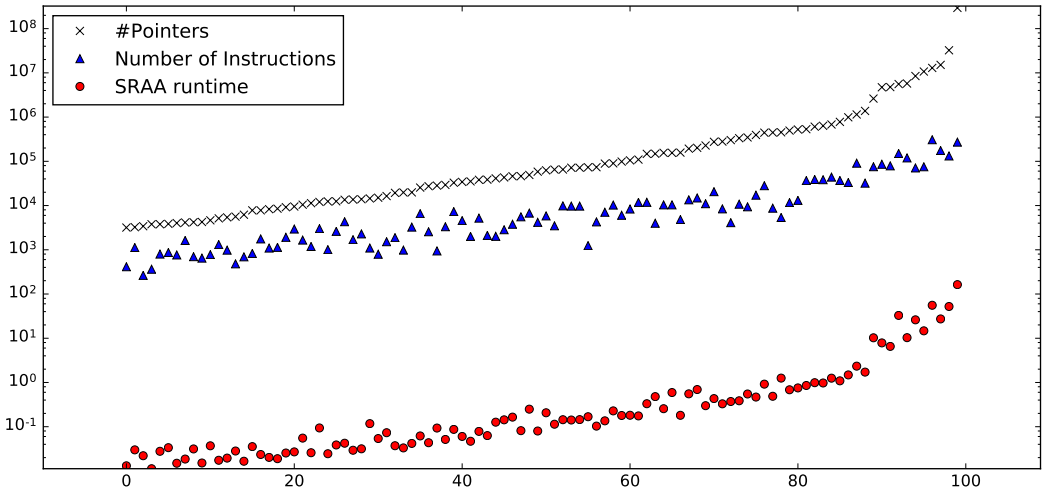


Figure 22: Comparison between the number of queries and total runtime (3 tests) per benchmark. X-axis represents LLVM benchmarks, sorted by number of instructions. We measure the SRAA runtime in seconds.

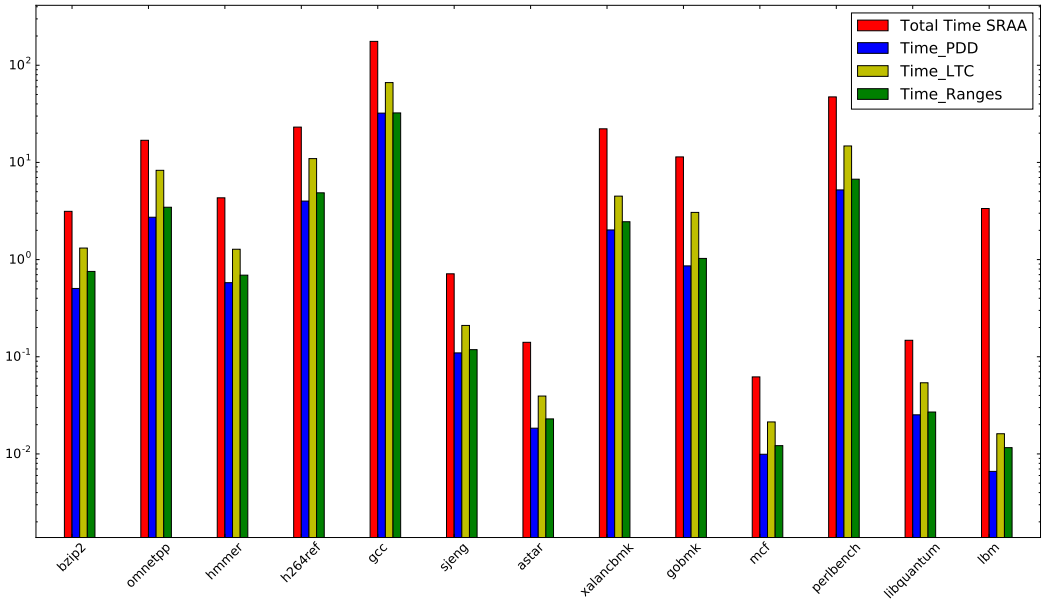


Figure 23: Time in seconds that each check (PDD, LTC, and Ranges) of our analysis takes to disambiguate all SPEC CPU2006 pairs of pointers.

transitive closure of the less-than relations between variables. The graph that represents this transitive closure might be cubic on the number of program

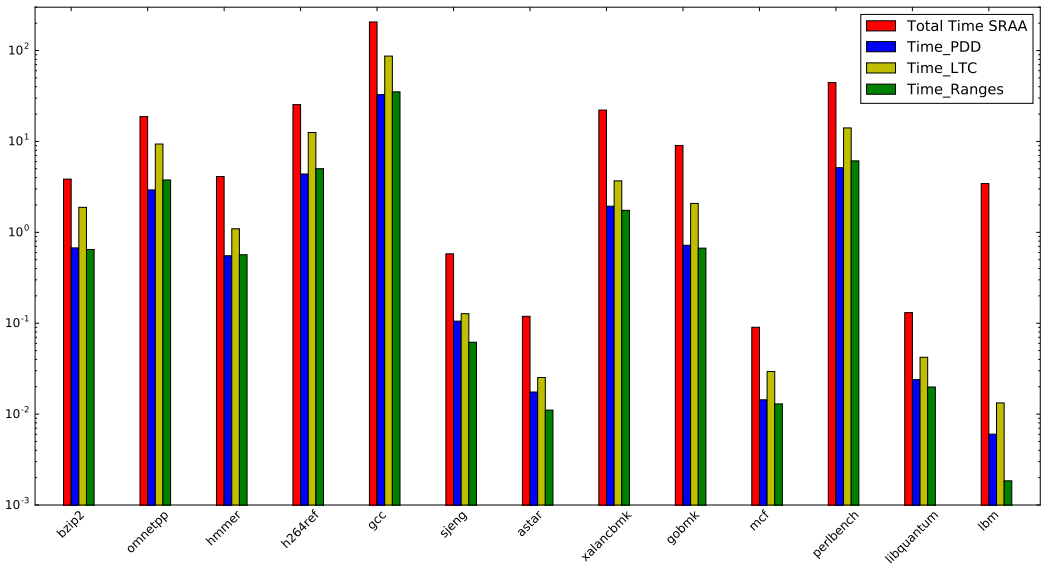


Figure 24: Time in seconds needed by each test (PDD, LTC, and Ranges) when run selectively to disambiguate pairs of pointers in SPEC CPU2006.

variables. 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 LTC test accounts for most of the execution time within the pointer disambiguation phase of this experiment. This behavior is due to the growth check that we use to solve ϕ-functions. The range and PDD tests take similar runtimes. In Figure 24, the less-than check runs only if the PDD test fails. Once we have built LT relations, this test amounts to consulting hash-tables. As a final observation, we notice that the range test tends to be the least time-intensive among the three disambiguation strategies that we have. It takes less time because it only runs on pairs of pointers within the same PDD.

Figure 25 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 25 does not show a clear winner. The fact that the three test are relatively fast explains this result.

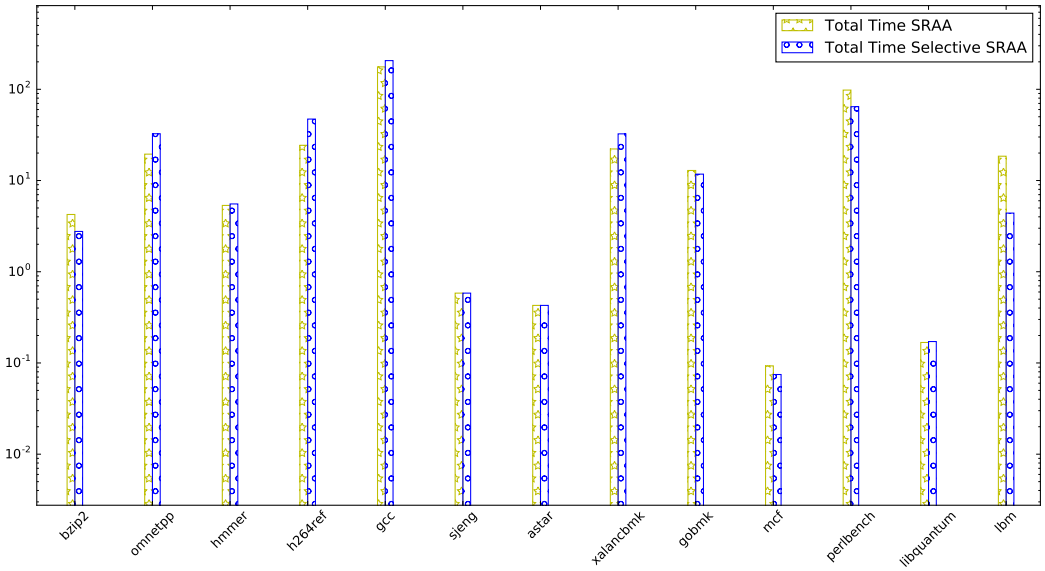


Figure 25: Comparison between time (in seconds) needed to run SRAA *selectively* (non overlapping tests) and *separately* (disjoint tests)

8. Related Work

Although alias analysis is an old subject in the compiler literature, it still draws the attention of researchers, who are interested in extending it to new domains [Petrashko et al. (2016)], making it yet more efficient [Dietrich et al. (2015)], or using it as a fundamental building block of new optimizations [Alves et al. (2015)] and static analyses [Johnson et al. (2015)]. In such context, this work has the goal of designing a more precise alias analysis for languages with pointer arithmetics. To this end, we join 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, except for the two papers that we have recently published, and that this document expands [Maalej et al. (2017); Paisante et al. (2016)]. In the rest of this section we discuss how we extend that previous work or ours, and how we differ from the current state-of-the-art literature on alias and less-than analyses.

8.1. A Comparison Against our Previous Work

As we have mentioned before, this paper extends two previous publications of ours. We believe that the current materialization of our ideas is more solid than our first two endeavors in the field of pointer disambiguation. To fundament this claim, in what follows we describe our previous work, and explain how we have extended it.

Symbolic Range Analysis of Pointers. Our first attempt to disambiguate pointers with offsets appeared in CGO'16 [Paisante et al. (2016)]. In that work, we have used a symbolic range analysis to place bounds on memory regions. Here, we use a combination of numeric range analysis and a less-than analysis to achieve such a goal. Therefore, whereas in [Paisante et al. (2016)] we have adopted an algebraic implementation, based on symbolic ranges, to associate pointers with allocation sites, in this work we provide a geometric interpretation of such relations. Hence, we build the pointer dependence digraph to track relations between pointers and offsets. The experiments seen in Section 7 indicate that the **Less-Than** and **Ranges** checks, described in Section 5, are not only enough to solve every query that our previous implementation could handle, but lets us go beyond it. In Example 14 we show how the geometric formalization allows us to disambiguate some pairs of pointers that our previous art would not handle.

Example 14. *Consider the control flow graph in Figure 26 in which our goal is to disambiguate pointers c_1 and d_1. The allocation site loc_1, associated to a_1 is unique in this program. It is created at $a_1 = malloc()$ and propagated to other pointers. Using the analysis that we have introduced in [Paisante et al. (2016)] we would conclude that c_1 and d_1 "may alias" since $GR(c_1) = \mathsf{loc}_1 + [2, 4]$ and $GR(d_1) = \mathsf{loc}_1 + [4, 5]$, and the two ranges have a non-null intersection. The **Ranges** test that we have developed in this work, on the other hand, is able to disambiguate pointers c_1 and d_1, because their common immediate ancestor is a_3. Rewriting the pointers based on a_3 gives: $c_1 = a_3+2$ or $c_1 = a_3+3$, and $d_1 = a_3+4$. Since $[2, 3] \cap [4, 4] = \emptyset$, then c_1 and d_1 cannot alias.*

Pointer Disambiguation via Strict Inequalities. A precursor of our **Less-Than** check is described in our previous work [Maalej et al. (2017)]. Our past experience has let us evolve that disambiguation technique along three different directions:

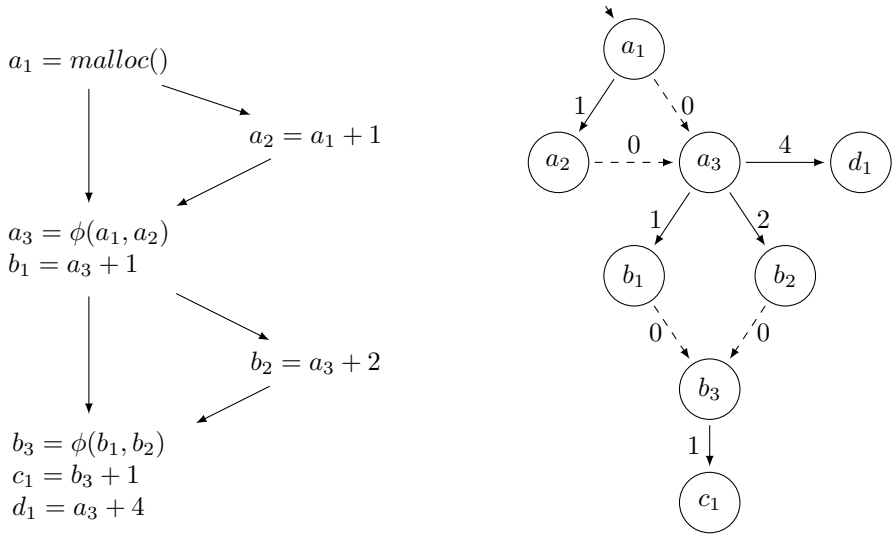


Figure 26: Illustration of the difference in precision between [Paisante et al. (2016)] and the Ranges test of this work: the program in SSA form (left) and its PDD (right)

- Given the instruction $v = v_1 + v_2$, plus the facts: $v_1 > 0$ and $v_2 > 0$, the final knowledge we obtain is different: in [Maalej et al. (2017)], we only generate the constraint $\mathsf{LT}(v) = \{v_1\} \cup \mathsf{LT}(v_1)$. In this work, we also generate the constraint $v_2 < v$ (which is equivalent to $\mathsf{LT}(v) = \{v_2\} \cup \mathsf{LT}(v_2)$.
- In [Maalej et al. (2017)] we consider integers and pointers as variables of scalar type while in the present implementation we differentiate the two types and handle separately the *add* and *gep* instructions present in the LLVM IR. This approach lets us avoid performing unnecessary comparisons. Hence, if v and v_1 are pointers and v_2 is an integer in $v = v_1 + v_2$, then following the C standard, the comparison between v and v_2 is not allowed.
- The third difference refers to the way we obtain a sparse implementation. In [Maalej et al. (2017)], the new information associated with a variable is created at definition sites and is invariant throughout the analysis. In this work we update the abstract state of a variable whenever new facts are available in the program code. The main benefit of this approach is precision. Because we compute a transitive closure for the less-than relations that we collect, we can disambiguate more pointers. Example 15 illustrates this improvement.

Example 15. *In this example we illustrate the difference in precision between our current and previous work. We consider the program snippet in figure 27. In [Maalej et al. (2017)], since* $x2 = x1 - 4$ *is a subtraction, we rename variable* $x1$ *to* $x1'$*, where* $x1'$ *is a fresh name. The final result of the analysis gives:* $LT(x1) = \{x0\}$, $LT(x1') = \{x0, x2\}$, $LT(x2) = \{\}$, $LT(x3) = \{x0, x1', x2\}$. *Using the rules of Figure 7 and Figure 8, we generate and solve constraints which gives us:* $LT(x1) = \{x0, x2\}$, $LT(x2) = \{\}$, $LT(x3) = \{x0, x1, x2\}$. *We want to disambiguate pointers* $p1$ *and* $p3$ *which are both defined from the same base pointer* p*. Our previous disambiguation check would answer "may alias" for those two pointers: however, in this work we answer "no alias". The imprecision in our previous technique happens because we lost the relation between* $x1$ *and* $x3$ *after the renaming at the split point.*

```
1  int x0;
2  int* p = malloc((x3+1)*sizeof(int));
3  int x1 = x0 + 2;
4  int* p1 = p + x1;
5  int x2 = x1 − 4;
6  int x3 = x1 + 6;
7  int* p3 = p + x3;
```

Figure 27: Illustration of the difference of precision between [Maalej et al. (2017)] and the Less-Than test of this work.

8.2. 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 [(Wolfe, 1996, Ch.7)] and Aho et al. [(Aho et al., 2006, 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 [Rugina and Rinard (2005)], 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 experiments to pointers within loops, we can analyze programs with over one million assembly instructions.

There exist several different pointer disambiguation strategies that associate ranges with pointers [Balakrishnan and Reps (2004); Balatsouras and Smaragdakis (2016); Alves et al. (2015); van Engelen et al. (2004); Nazaré et al. (2014); Paisante et al. (2016); Rugina and Rinard (2005); Rus et al. (2002); Sui et al. (2016); Wilson and Lam (1995)]. 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 [Balakrishnan and Reps (2004); Sui et al. (2016)] or contiguously [Alves et al. (2015); Rus et al. (2002)]; with symbolic bounds [Nazaré et al. (2014); Paisante et al. (2016); Rugina and Rinard (2005)] or with numeric bounds [Balakrishnan and Reps (2004); Balatsouras and Smaragdakis (2016); Sui et al. (2016)], 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 exists 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 [Steensgaard (1996)], Andersen's [Andersen (1994)], or even the state-of-the-art technique of Hardekopf and Lin [Hardekopf and Lin (2011)]. 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 [Hardekopf and Lin (2007)], or wave propagation [Pereira and Berlin (2009)].

8.3. 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 [Logozzo and Fähndrich (2008); Logozzo and Fähndrich (2010)] and Bodik's [Bodik et al. (2000)]. Additionally, there exist non-relational analyses that produce enough information to solve less-than equations [Cousot and Halbwachs (1978); Miné (2006)]. 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 [Bodik et al. (2000)]. 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. Since their debut [Logozzo and Fähndrich (2008); Logozzo and Fähndrich (2010)], 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.* [Nazaré et al. (2014)] have used Pentagons to reduce the overhead imposed by AddressSanitizer [Serebryany et al. (2012)] 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 [Fähndrich and Logozzo (2010)].

Pentagon, when seen as an 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, similarly to the method we propose in this paper. Nevertheless, there are differences between our algorithm and Logozzo's. First, it is necessary to separate pointers from integer offsets in the resolution of the constraints that build less-than relations between variables. Second, the original work on Pentagons describe a dense analysis, whereas we use a different program representation to achieve sparsity. Therefore, the constraints that we produce to analyze programs are very different from in the original description of this lattice.

Third, 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. That is to say, we have 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. 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. Finally, from a technical standpoint, we have added minor improvements on the original description of Pentagons.

For instance, our new algorithm handles some programming constructions that the original description of Pentagons did not touch, such as operations involving two variables on the right side, e.g., $v = v_1 + v_2$.

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 Octagons [Miné (2006)] or Polyhedrons [Cousot and Halbwachs (1978)], associate tuples of variables with abstract information. For instance, Miné's Octagons 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[6] 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 $\mathtt{j} - \mathtt{i} \geq 1$. Nevertheless, there are situations in which we are still more precise than Polly: for instance, it cannot disambiguate `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 analyzers such as Astrée [Cousot et al. (2005)] or Pagai [Henry et al. (2012)] assign *unknown values for base addresses*. These values permit such tools to derive relation between pointers, treating them as numerical values. In practice, these analyses work because they handle programs with very few pointers, like safety critical code.

9. 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 four times in some cases. And, contrary to previous algebraic approaches, our analysis scales up to very large programs, with millions of assembly instructions. We

[6] Available at `http://polly.llvm.org/`

believe that this type of technique opens up opportunities 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)).

References

Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D., 2006. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley.

Alpern, B., Wegman, M. N., Zadeck, F. K., 1988. Detecting equality of variables in programs. In: POPL. ACM, pp. 1–11.

Alves, P., Gruber, F., Doerfert, J., Lamprineas, A., Grosser, T., Rastello, F., Pereira, F. M. Q. a., 2015. Runtime pointer disambiguation. In: OOPSLA. ACM, New York, NY, USA, pp. 589–606.

Andersen, L. O., 1994. Program analysis and specialization for the c programming language. Ph.D. thesis, DIKU, University of Copenhagen.

Balakrishnan, G., Reps, T., 2004. Analyzing memory accesses in x86 executables. In: CC. Springer, pp. 5–23.

Balatsouras, G., Smaragdakis, Y., 2016. Structure-sensitive points-to analysis for C and C++. In: SAS. Springer, pp. 84–104.

Blume, W., Eigenmann, R., 1994. Symbolic range propagation. In: IPPS. pp. 357–363.

Bodik, R., Gupta, R., Sarkar, V., 2000. ABCD: eliminating array bounds checks on demand. In: PLDI. ACM, pp. 321–333.

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

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X., 2005. The ASTRÉE analyzer. In: European symposium on programming (ESOP). No. 3444 in Lecture Notes in Computer Science. pp. 21–30.

Cousot, P., Halbwachs, N., 1978. Automatic discovery of linear restraints among variables of a program. In: POPL. ACM, pp. 84–96.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., Zadeck, F. K., 1989. An efficient method of computing static single assignment form. In: POPL. pp. 25–35.

Davis, M., Matiyasevich, Y., Robinson, J., 1976. Hilbert's tenth problem: Diophantine equations: positive aspects of a negative solution. In: Symposia in Pure Mathematics. Vol. 28. AMS, Providence, RI, USA, pp. 323–378.

Dietrich, J., Hollingum, N., Scholz, B., 2015. Giga-scale exhaustive points-to analysis for java in under a minute. In: OOPSLA. ACM, New York, NY, USA, pp. 535–551.

Fähndrich, M., Logozzo, F., 2010. Static contract checking with abstract interpretation. In: FoVeOOS. Springer, pp. 10–30.

Ferrante, J., Ottenstein, J., Warren, D., 1987. The program dependence graph and its use in optimization. TOPLAS 9 (3), 319–349.

Hardekopf, B., Lin, C., 2007. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In: PLDI. ACM, pp. 290–299.

Hardekopf, B., Lin, C., 2011. Flow-sensitive pointer analysis for millions of lines of code. In: CGO. pp. 265–280.

Henning, J. L., 2006. Spec cpu2006 benchmark descriptions. SIGARCH Comput. Archit. News 34 (4), 1–17.

Henry, J., Monniaux, D., Moy, M., 2012. Succinct representations for abstract interpretation: Combined analysis algorithms and experimental evaluation. In: SAS. Springer, pp. 283–299.

Hind, M., 2001. Pointer analysis: Haven't we solved this problem yet? In: In PASTE. ACM, pp. 54–61.

ISO, 9899:2011. Programming Languages – C. IEC.
URL www.open-std.org/jtc1/sc22/wg14/docs/n1539.pdf

Johnson, A., Waye, L., Moore, S., Chong, S., 2015. Exploring and enforcing security guarantees via program dependence graphs. In: PLDI. ACM, New York, NY, USA, pp. 291–302.

Lattner, C., Adve, V. S., 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO. IEEE, pp. 75–88.

Logozzo, F., Fähndrich, M., 2008. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In: SAC. ACM, pp. 184–188.

Logozzo, F., Fähndrich, M., 2010. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. Sci. Comput. Program. 75 (9), 796–807.

Maalej, M., Paisante, V., Ramos, P., Gonnord, L., Pereira, F. M. Q. a., 2017. Pointer disambiguation via strict inequalities. In: CGO. ACM, pp. 134–147.

Miné, A., 2006. The octagon abstract domain. Higher Order Symbol. Comput. 19, 31–100.

Nazaré, H., Maffra, I., Santos, W., Barbosa, L., Gonnord, L., Pereira, F. M. Q., 2014. Validation of memory accesses through symbolic analyses. In: OOPSLA. ACM, pp. 791–809.

Nielson, F., Nielson, H. R., Hankin, C., 2005. Principles of program analysis. Springer.

Paisante, V., Maalej, M., Barbosa, L., Gonnord, L., Quintão Pereira, F. M., 2016. Symbolic range analysis of pointers. In: CGO. ACM, pp. 171–181.

Pereira, F. M. Q., Berlin, D., 2009. Wave propagation and deep propagation for pointer analysis. In: CGO. IEEE, pp. 126–135.

Petrashko, D., Ureche, V., Lhoták, O., Odersky, M., 2016. Call graphs for languages with parametric polymorphism. In: OOPSLA. ACM, New York, NY, USA, pp. 394–409.

Rodrigues, R. E., Campos, V. H. S., Pereira, F. M. Q., 2013. A fast and low overhead technique to secure programs against integer overflows. In: CGO. ACM, pp. 1–13.

Rugina, R., Rinard, M. C., 2000. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In: PLDI. ACM, pp. 182–195.

Rugina, R., Rinard, M. C., 2005. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. TOPLAS 27 (2), 185–235.

Rus, S., Rauchwerger, L., Hoeflinger, J., 2002. Hybrid analysis: Static and dynamic memory reference analysis. In: ICS. IEEE, pp. 251–283.

Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D., 2012. Addresssanitizer: a fast address sanity checker. In: ATC. USENIX, pp. 28–28.

Singer, J., 2006. Static program analysis based on virtual register renaming. Ph.D. thesis, University of Cambridge.

Steensgaard, B., 1996. Points-to analysis in almost linear time. In: POPL. pp. 32–41.

Sui, Y., Fan, X., Zhou, H., Xue, J., 2016. Loop-oriented array- and field-sensitive pointer analysis for automatic SIMD vectorization. In: LCTES. ACM, pp. 41–51.

Surendran, R., Barik, R., Zhao, J., Sarkar, V., 2014. Inter-iteration scalar replacement using array SSA form. In: CC. Springer, pp. 40–60.

Tavares, A. L. C., Boissinot, B., Pereira, F. M. Q., Rastello, F., 2014. Parameterized construction of program representations for sparse dataflow analyses. In: CC. Springer, pp. 2–21.

van Engelen, R. A., Birch, J., Shou, Y., Walsh, B., Gallivan, K. A., 2004. A unified framework for nonlinear dependence testing and symbolic analysis. In: ICS. ACM, pp. 106–115.

Wilson, R. P., Lam, M. S., 1995. Efficient context-sensitive pointer analysis for c programs. In: PLDI. ACM, pp. 1–12.

Wolfe, M., 1996. High Performance Compilers for Parallel Computing, 1st Edition. Adison-Wesley.