Symbolic Range Analysis of Pointers

Vitor Paisante
Department of Computer Science
UFMG, Brazil
paisante@dcc.ufmg.br

Maroua Maalej
University of Lyon, France & LIP
(UMR CNRS/ENS Lyon/
UCB Lyon1/INRIA)
Maroua.Maalej@ens-lyon.fr

Leonardo Barbosa
Department of Computer Science
UFMG, Brazil
leob@dcc.ufmg.br

Laure Gonnord
Univ. Lyon1, France & LIP
(UMR CNRS/ENS Lyon/
UCB Lyon1/INRIA)
Laure.Gonnord@ens-lyon.fr

Fernando Magno Quintão Pereira
Department of Computer Science
UFMG, Brazil
fernando@dcc.ufmg.br

Abstract

Alias analysis is one of the most fundamental techniques that compilers use to optimize languages with pointers. However, in spite of all the attention that this topic has received, the current state-of-the-art approaches inside compilers still face challenges regarding precision and speed. In particular, pointer arithmetic, a key feature in C and C++, is yet to be handled satisfactorily. This paper presents a new alias analysis algorithm to solve this problem. The key insight of our approach is to combine alias analysis with symbolic range analysis. This combination lets us disambiguate fields within arrays and structs, effectively achieving more precision than traditional algorithms. To validate our technique, we have implemented it on top of the LLVM compiler. Tests on a vast suite of benchmarks show that we can disambiguate several kinds of C idioms that current state-of-the-art analyses cannot deal with. In particular, we can disambiguate 1.35x more queries than the alias analysis currently available in LLVM. Furthermore, our analysis is very fast: we can go over one million assembly instructions in 10 seconds.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors - Compilers

General Terms Languages, Experimentation

Keywords Alias analysis, range analysis, speed, precision

CGO'16, March 12–18, 2016, Barcelona, Spain
© 2016 ACM. 978-1-4503-3778-6/16/03...$15.00
http://dx.doi.org/10.1145/2854038.2854050

1. Introduction

Pointer analysis is one of the most fundamental compiler technologies. This analysis lets the compiler distinguish one memory location from others; hence, it provides the necessary information to transform code that manipulates memory. Given this importance, it comes as no surprise that pointer analysis has been one of the most researched topics within the field of compiler construction [12]. This research has contributed to make the present algorithms more precise [10, 27], and faster [11, 21]. Nevertheless, one particular feature of imperative programming languages remains to be handled satisfactorily by the current state-of-the-art approaches: the disambiguation of pointer intervals.

Mainstream compilers still struggle to distinguish intervals within the same array. In other words, state-of-the-art pointer analyses often fail to disambiguate regions addressed from a common base pointer via different offsets, as explained by Yong and Horwitz [26]. Field-sensitive pointer analysis, provide a partial solution to this problem. These analyses can distinguish different fields within a record, such as a struct in C [17], or a class in Java [25]. However, they rely on syntax that is usually absent in the low level program representations adopted by compilers. Shape analyses [13] can disambiguate subparts of data-structures such as arrays, yet their scalability remains an issue to be solved. Consequently, many compiler optimizations, such as loop transformations, tiling, fission, skewing and interchanging [24, Ch.09], are very limited in practice. Therefore, we claim that, to reach their full potential, compilers need to be provided with more effective alias analyses.

This paper describes such an analysis. We introduce an abstract domain that associates pointers with symbolic ranges. In other words, for each pointer p we conserva-

tively estimate the range of memory slots that can be addressed as an offset of p. We let $\text{GR}(p)$ be the global abstract address set set associated with pointer p, such that if $\mathsf{loc}_i + [l, u] \in \text{GR}(p)$, then p may dereference any address from $@(\mathsf{loc}_i) + l$ to $@(\mathsf{loc}_i) + u$, where loc_i is a program site that contains a memory allocation call, and $@(\mathsf{loc}_i)$ is the actual return address of the `malloc` at runtime. We let $\{l, u\}$ be two *symbols* defined within the program code. Like the vast majority of pointer analyses available in the compiler literature, from Andersen's work [2] to the more recent technique of Zhang *et al.* [27], our method is correct if the underlying program is also correct. In other words, our results are sound with respect to the semantics of the program if this program has no undefined behavior, such as out-of-bounds accesses.

The key insight of this paper is the combination of pointer analysis with range analysis on the symbolic interval lattice. In a symbolic range analysis, ranges are defined as expressions of the program symbols, a symbol being either a constant or the name of a variable. There exist many approaches to symbolic range analyses in the literature [4, 15, 19]. The algorithms that we present in this paper do not depend on any particular implementation. Nevertheless, the more precise the range analysis that we use, the more precise the analysis facts that we produce. In this work we have adopted the symbolic range analysis proposed in 1994 by William Blume and Rudolf Eigenmann [4].

To validate our ideas, we have implemented them in the LLVM compilation infra-structure [14]. We have tested our pointer analysis onto three different benchmarks used in previous work related to pointer disambiguation: Prolangs [20], PtrDist [28] and MallocBench [9]. As we show in Section 4, our analysis is linear on the size of programs. It can go over one-million assembly instructions in approximately 10 seconds. Furthermore, we can disambiguate 1.35x more queries than the alias analysis currently available in LLVM.

2. Overview

We have two different ways to answer the following question: "do pointers tmp_i and tmp_j alias?" These tests are called *global* and *local*. In this section, we will use two different examples to illustrate situations in which each query is more effective. These distinct strategies are complementary: one is not a superset of the other.

Global pointer disambiguation. Figure 1 illustrates our first approach to disambiguate pointers. The figure shows a pattern typically found in distributed systems implemented in C. Messages are represented as arrays of bytes. In this particular example, messages have two parts: an identifier, which is stored in the beginning of the array, and a payload, which is stored right after. The loops in lines 5-8 and 9-12 fill up each of these parts with data. If a compiler can prove that the stores at lines 6 and 10 are always independent, then it can perform optimizations that would not be possible otherwise. For instance, it can parallelize the loops, or switch them, or merge them into a single body.

```
1  #include <stdlib.h>
2
3  void prepare(char* p, int N, char* m) {
4    char *i, *e, *f;
5    for (i = p, e = p + N; i < e; i += 2) {
6      *i = 0;
7      *(i + 1) = 0xFF;
8    }
9    for (f = e + strlen(m); i < f; i++) {
10     *i = *m;
11     m++;
12   }
13 }
14
15 int main(int argc, char** argv) {
16   int Z = atoi(argv[1]);
17   char* b = (char*)malloc(Z);
18   char* s = (char*)malloc(strlen(argv[2]));
19   strcpy(s, argv[2]);
20   prepare(b, Z, s);
21   ...
22   return 0;
23 }
```

Figure 1. Example of program that builds up messages as sequences of serialized bytes. We are interested in disambiguating the locations accessed at lines 6 and 10.

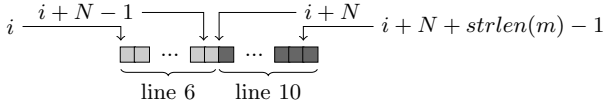


Figure 2. Array p in the routine `prepare` seen in Fig.1. Lines 6 and 10 represent the different stores in the figure.

No alias analysis currently available in either gcc or LLVM is able to disambiguate the stores at lines 6 and 10. These analyses are limited because they do not contain *range information*. The range interval $[l, u]$ associated with a variable i is an estimate of the lowest (l) and highest (u) values that i can assume throughout the execution of the program. In this paper, we propose an alias analysis that solves this problem. To achieve this goal, we couple this alias analysis with range analysis on symbolic intervals [4]. Thus, we will say that the store at line 6 might modify any address from $\mathtt{p} + 0$ to $\mathtt{p} + \mathtt{N} - 1$, and that the store at line 10 might write on any address from $\mathtt{p} + N$ to $\mathtt{p} + N + \mathtt{strlen(m)} - 1$. For this purpose, we will use an *abstract address* that encodes the actual value(s) of p inside the `prepare` function. These memory addresses are depicted in Figure 2, where each $\square$ represents a memory slot.

Whole program analysis reveals that there are two candidate locations that any pointer in the program may refer to. These locations have been created at lines 17 and 18 of Figure 1, and we represent them abstractly as loc_{17} and

```
1 void accelerate
2 (float* p, float X, float Y, int N) {
3   int i = 0;
4   while (i < N) {
5     p[i] += X;     // float* tmp0 = p + i; *tmp0 = ...;
6     p[i + 1] += Y;      // float* tmp1 = p + i + 1;
7     i += 2;             // *tmp1 = ...;
8   }
9 }
```

Figure 3. Program that shows the need to assign common names to addresses that spring from the same base pointer.

```
1 void accelerate
2 (float* p, float X, float Y, int N) {
3   int i = 0;
4   while (i < N) {
5     float* newp = p+i;    // LR(newp) = loc_new + [0, 0]
6     newp[0] += X;  // float* tmp2 = newp; *tmp2 = ...;
7     newp[1] += Y;  // float* tmp3 = newp + 1; *tmp3 = ...;
8     i += 2;
9   }
10 }
```

Figure 4. Program from Figure 3, after pointer is renamed within loop.

loc_{18}. These names are unique across the entire program. After running our analysis, we find out that the abstract state (GR) of `i` at line 6 is $\text{GR}(\mathtt{i}_{\ell n.6}) = \{\mathsf{loc}_{17} + [0, \mathtt{N} - 1]$, and that the abstract state of `i` at line 10 is $\text{GR}(\mathtt{i}_{\ell n.10}) = \{\mathsf{loc}_{17} + [\mathtt{N}, \mathtt{N} + \mathtt{strlen}(m) - 1]\}$. Given that these two abstract ranges do not intersect, we know that the two stores update always different locations. We call this check the *global disambiguation criterion*.

Local pointer disambiguation. Figure 3 shows a program in which the simple intersection of ranges would not let us disambiguate pointers tmp_0 and tmp_1. After solving global range analysis for that program, we have that $\text{GR}(tmp_0) = \{\mathsf{loc}_0 + [0, N + 1]\}$ and that $\text{GR}(tmp_1) = \{\mathsf{loc}_0 + [1, N + 2]\}$, where loc_0 defines the abstract address of the function parameter p. The intersection of these ranges is non-empty for $N \geq 1$. Thus, the global check that we have used to disambiguate locations in Figure 1 does not work in Figure 3. Notwithstanding this fact, we know that tmp_0 and tmp_1 will never point to a common location. In fact, these pointers constitute different offsets from the same base address. To deal with this imprecision of the global check, we will be also discussing a *local disambiguation criterion*. In this case, we rename every pointer p that is alive at the beginning of a single entry region to a fresh name new_p. Whereas we use the global test for pointers in different regions, the local test is applied onto pointers within the same single entry region. After renaming, we update the table of pointer pairs, so that $\text{LR}(new_p) = \mathsf{loc}_{new} + [0, 0]$, regardless of the old ranges assigned to the original pointer p. In Figure 4 we would have that $\text{LR}(tmp_2) = \mathsf{loc}_{new} + [0, 0]$ and $\text{LR}(tmp_3) = \mathsf{loc}_{new} + [1, 1]$, where tmp_2 is the name of the address $new_p[0]$, and tmp_3 is the name of the address $new_p[1]$. This new binding of intervals to pointers gives us empty intersections between similar locations in $\text{LR}(tmp_2)$ and $\text{LR}(tmp_3)$. Consequently, the local check is able to distinguish addresses referenced by tmp_2 and tmp_3.

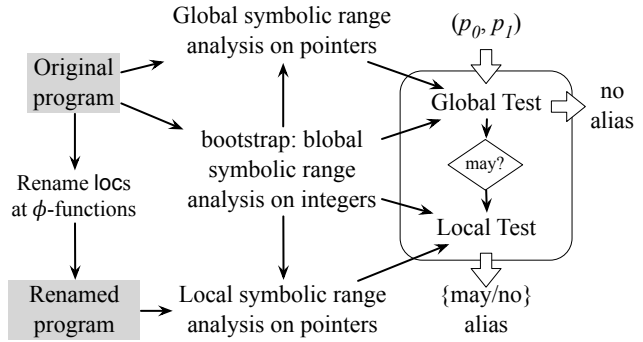


Figure 5. Overview of our pointer analysis.

3. Combining Range and Pointer Analyses

We perform our pointer analysis in several steps. Figure 5 shows how each of these phases relates to each other. Our final product is a function that, given two pointers, p_0 and p_1, tells if they may point to overlapping areas or not. An invocation of this function is called a *query*. We use an off-the-shelf symbolic range analysis, e.g., à la Blume [4], to bootstrap our pointer analysis. By inferring the symbolic ranges of pointers, we have two alias tests: the global and the local approach. In the rest of this section we describe each one of these contributions.

3.1 A Core Language

We solve range analysis through abstract interpretation. To explain how we abstract each instruction in our intermediate representation, we shall use the language seen in Figure 6; henceforth, we shall call this syntax our *core language*. We shall be working on programs in Extended Static Single Assignment (e-SSA) form [5]. E-SSA form is a flavor of Static Single Assignment (SSA) [8] form, with variable renaming after inequalities. Thus, our core language contains ϕ-functions to ensure the single definition (SSA) property, and intersections to rename variables after conditionals. We assume that ϕ-functions have only two arguments. Generalizing this notation to n-ary functions is immediate.

Figure 7 shows the control flow graph of the program seen in Figure 1. The implementation of the analysis that we shall present in this paper is interprocedural, albeit not context-sensitive. To achieve interprocedurality, we associate actual parameters with formal parameters of functions. In the example of Figure 1, pointer b – an actual parameter – is linked with p – a formal parameter – through a ϕ-function.

Integer constants	::=	$\{c_1, c_2, \ldots\}$
Integer variables	::=	$\{i_1, i_2, \ldots\}$
Pointer variables	::=	$\{p_1, p_2, \ldots\}$
Instructions (I)	::=	
– Allocate memory	|	$p_0 = \text{malloc}(i_0)$
– Free memory	|	$p_0 = \text{free}(p_1)$
– Pointer plus int	|	$p_0 = p_1 + i_0$
– Pointer plus const	|	$p_0 = p_1 + c_0$
– Bound intersection	|	$p_0 = p_1 \cap [l, u]$
– Load into pointer	|	$p_0 = *p_1$
– Store from pointer	|	$*p_0 = p_1$
– ϕ-function	|	$p_0 = \phi(p_1 : \ell_1, p_2 : \ell_2)$
– Branch if not zero	|	$\text{bnz}(v, \ell)$
– Unconditional jump	|	$\text{jump}(\ell)$

Figure 6. The syntax of our language of pointers.

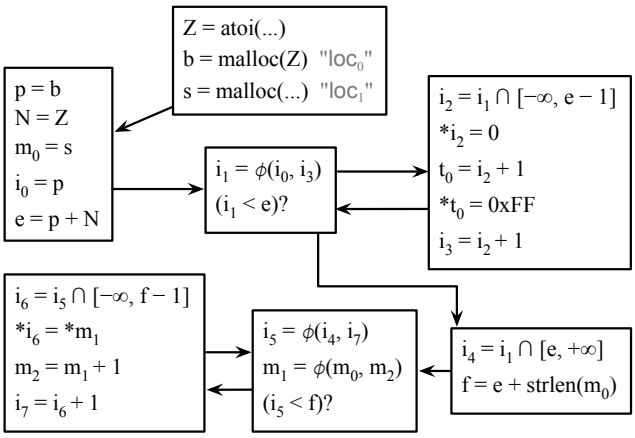


Figure 7. Control flow graph of program seen in Figure 1

The e-SSA format lets us implement our analysis sparsely, e.g,. we can assign information directly to variables, instead of pairs of variables and program points. As demonstrated by Choi *et al.* [6], the main advantage of a sparse analysis is efficiency: the product of the analysis - the information that is bound to each variable - requires $O(N)$ space, where N is the number of variable names in the program. Furthermore, as we shall explain in the rest of this section, our analysis can be computed in $O(N)$ time.

Key to these good properties is the fact that we create new variable names at each program point where our analysis can infer new information. This knowledge appears due to memory allocation (malloc), deallocation (free), pointer arithmetic, intersections and ϕ-functions. Each of these instructions defines new variables, whose names are associated with information. For instance, the instruction $p_0 = \text{free}(p_1)$ copies p_1 to p_0, and binds p_0 to a memory chunk of size 0. As we will show in Section 3.4, our abstract interpreter associates with p_0 a new abstract state which indicates that p_0 is not a valid reference to any location.

3.2 Program Locations

Our analysis binds variable names to sets of *locations* and *ranges*. We denote the set of locations in a program by $\mathcal{L}oc = \{\mathsf{loc}_0, \mathsf{loc}_1 \ldots, \mathsf{loc}_{n-1}\}$ where n is the number of allocation sites. In our representation, i.e., Figure 6, new locations are created by *malloc* operations.

EXAMPLE 1. *Figure 7 shows the control flow graph of the program seen in Figure 1. The two allocations at lines 17 and 18 are associated respectively with* loc_0 *and* loc_1.

3.3 Symbolic Range Analysis

We start our pointer analysis by running an off-the-shelf *range analysis* parameterized on *symbols*. For the sake of completeness, we shall revisit the main notions associated with range analysis, which we borrow from Nazaré *et al.* [15]. We say that E is a symbolic *expression*, if and only if, E is defined by the grammar below. In this definition, s is a symbol and $n \in \mathbb{N}$. The set of symbols s in a program form its *symbolic kernel*. The symbolic kernel is formed by names that cannot be represented as function of other names in the program text. Concretely, this set contains the names of global variables and variables assigned with values returned from library functions.

$$E \quad ::= \quad n \mid s \mid \min(E, E) \mid \max(E, E) \mid E - E \\ \mid E + E \mid E/E \mid E \bmod E \mid E \times E$$

We shall be performing arithmetic operations over the partially ordered set $S = S_E \cup \{-\infty, +\infty\}$, where S_E is the set of symbolic expressions. The partial ordering is given by $-\infty < \ldots < -2 < -1 < 0 < 1 < 2 < \ldots + \infty$. There exists no ordering between two distinct elements of the symbolic kernel of a program. For instance, $N < N+1$ but there is no relationship between an expression containing N and another expression containing M.

A *symbolic interval* is a pair $R = [l, u]$, where l and u are symbolic expressions. We denote by $R_\downarrow$ the lower bound l and $R_\uparrow$ the upper bound u. We define the partially ordered set of (symbolic) intervals $S^2 = (S \times S, \sqsubseteq)$, where the ordering operator is defined as:

$$[l_0, u_0] \sqsubseteq [l_1, u_1], \text{ if } l_1 \leq l_0 \wedge u_1 \geq u_0$$

We define the semi-lattice SymbRanges of symbolic intervals as $(S^2, \sqsubseteq, \sqcup, \emptyset, [-\infty, +\infty])$, where the join operator "$\sqcup$" is defined as:

$$[a_1, a_2] \sqcup [b_1, b_2] = [\min(a_1, b_1), \max(a_2, b_2)]$$

Our lattice has a least element $\emptyset$, such that:

$$\emptyset \sqcup [l, u] = [l, u] \sqcup \emptyset = [l, u]$$

and a greatest element $[-\infty, +\infty]$, such that:

$$[-\infty, +\infty] \sqcup [l, u] = [l, u] \sqcup [-\infty, +\infty] = [-\infty, +\infty]$$

For sake of clarity, we also define the intersection operator "$\sqcap$":

$$[a_1, a_2] \sqcap [b_1, b_2] = \begin{cases} \emptyset, & \text{if } a_2 < b_1 \text{ or } b_2 < a_1 \\ [\max(a_1, b_1), \min(a_2, b_2)], & \text{otherwise} \end{cases}$$

$[-\infty, +\infty]$ is absorbant and $\emptyset$ is neutral for $\sqcap$.

The result of range analysis is a function $R : V \mapsto S^2$, that maps each integer variable i in a program to an interval $[l, u], l \leq u$, e.g., $R(i) = [l, u]$. The more precise the technique we use to produce this result, the more precise our results will be. Nevertheless, the exact implementation of the range analysis is immaterial for the formalization that follows. In this paper, we are using the following widening operator on SymbRanges :

$$[l, u] \nabla [l', u'] = \begin{cases} [l, u] & \text{if } l = l' \text{ and } u = u' \\ [l, +\infty] & \text{if } l = l' \text{ and } u' > u \\ [-\infty, u] & \text{if } l' < l \text{ and } u' = u \\ [-\infty, +\infty] & \text{if } l' < l \text{ and } u' > u \end{cases}$$

The only requirement that we impose on the implementation of range analysis is that it exists over SymbRanges , our lattice of symbolic intervals.
We denote by $(\alpha_{\mathsf{SymbRanges}}, \gamma_{\mathsf{SymbRanges}})$ the underlying galois connection.

EXAMPLE 2. *A range analysis such as Nazaré* et al.*'s [15], if applied onto the program seen in Figure 3, will gives us that* $R(i_{\langle line\ 3 \rangle}) = [0, 0]$, $R(i_{\ell n.5}) = [0, N-1]$, $R(i_{\ell n.7}) = [0, N+1]$.

3.4 Global Range Analysis of Pointers

As we have mentioned in Section 2 we use two different strategies to disambiguate pointers: the global and the local test. Our global pointer analysis goes over the entire code of the program, associating variables that have the pointer type with elements of an abstract domain that we will define soon. The local analysis, on the other hand, works only for small regions of the program text. We shall discuss the local test in Section 3.6. In this section, we focus on the global test, which is an abstract-interpretation based algorithm.

An Abstract Domain of Pointer Locations. We associate pointers with tuples of size n: $(\mathsf{SymbRanges} \cup \bot)^n$; n being the number of program sites where memory is allocated (the cardinal of $\mathcal{L}oc$) and $\cup$ is the disjoint union.

Let $@(\mathsf{loc}_i)$ denotes the actual address value returned by the i^{th} malloc of the program. By construction, all actual addresses are supposed to be offsets of a given $@(\mathsf{loc}_i)$. The abstract value $\text{GR}(p) = (p_0, \ldots, p_{n-1})$ represents (an abstract version) of the set of memory locations that pointer variable p can address throughout the execution of a program:

DEFINITION 1 (Abstraction). *A set of actual addresses,* $S = \{s \mid \exists i \in \mathbb{N}, d \in \mathbb{Z}, s = @(\mathsf{loc}_i) + d\}$ *is abstracted by* $\alpha(S) = (p_0, p_1 \ldots, p_{n-1})$ *where :*

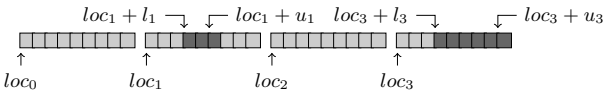


Figure 8. The concrete semantics of $\text{GR}(p) = \{\mathsf{loc}_1 + [3, 5], \mathsf{loc}_3 + [3, 8]\}$. Dark grey cells denote possible (concrete) values of p.

- $p_i = \bot$ *if there is no adress in* S *which is an offset of* $@(\mathsf{loc}_i)$
- $p_i = \alpha_{\mathsf{SymbRanges}}(\{d \in \mathbb{Z} \mid s = @(\mathsf{loc}_i) + d \in S\})$, *otherwise. The offsets from a given pointer are abstracted alltogether in the* SymbRanges *lattice.*

The goal of our GR analysis is to compute such an abstract value for each pointer of the program. Some elements in a tuple $\text{GR}(p)$ are bound to the undefined location, e.g., $\bot$. These elements are not interesting to us, as they do not encode any useful information. Thus, to avoid keeping track of them, we rely on the concept of *support*, which we state in Definition 2.

DEFINITION 2 (Support). *We denote by* $supp_{GR}(p)$ *the set of indexes for which* p_i *is not* $\bot$ *:*

$$supp_{GR}(p) = \{i \mid p_i \neq \bot\}.$$

For sake of readability, let us denote for instance $\text{GR}(p) = (\bot, [l_1, u_1], \bot, [l_3, u_3], \bot)$, by the set $\text{GR}(p) = \{\mathsf{loc}_1 + [l_1, u_1], \mathsf{loc}_3 + [l_3, u_3]\}$. In the concrete world, this notation will mean that pointer p can address any memory location from $@(\mathsf{loc}_1) + l_1$ to $@(\mathsf{loc}_1) + u_1$, and from $@(\mathsf{loc}_3) + l_3$ to $@(\mathsf{loc}_3) + u_3$.

For instance, consider that $l_1 = 3$, $u_1 = 5$, $l_3 = 3$ and $u_3 = 8$. $\text{GR}(p) = \{\mathsf{loc}_1 + [3, 5], \mathsf{loc}_3 + [3, 8]\}$ is then depicted in Figure 8.

Now for the abstract operations: $(\bot, \ldots, \bot)$ is the least element of our lattice, and $([-\infty, \infty], \ldots, [-\infty, \infty])$ the greatest one.

Given the two abstract values $\text{GR}(p^1) = (p_0^1, \ldots p_{n-1}^1)$ and $\text{GR}(p^2) = (p_0^2, \ldots p_{n-1}^2)$, the union $\text{GR}(p^1) \sqcup \text{GR}(p^2)$ is the tuple $(q_0, \ldots, q_{n-1})$ where:

$$q_i = \begin{cases} \bot & \text{if } p_i^1 = p_i^2 = \bot \\ p_1 \sqcup p_2 & \text{else} \end{cases}$$

and $\text{GR}(p^1) \sqsubseteq \text{GR}(p^2)$ if an only if all involved (symbolic) intervals of p^1 are included in the ones of p^2: $\forall i \in [0..(n-1)], p_i^1 \sqsubseteq p_i^2$ (considering $\bot \sqsubseteq R$ and $\bot \sqcup R = R$ for all non-empty intervals R). We call this structure, formed by $(\mathsf{SymbRanges} \cup \bot)^n$ plus its partial ordering the lattice MemLocs .

EXAMPLE 3. *For the example depicted in Figure 7 where we only have two malloc sites denoted by* loc_0 *and* loc_1*, we will obtain the following results:* $GR(p) = GR(b) = \{\mathsf{loc}_0 +$

$[0,0]\}$, $GR(m_0) = GR(s) = \{loc_1 + [0,0]\}$, $GR(e) = loc_0 + [N,N]$, $GR(m_1) = loc_1 + [1,+\infty]$, $GR(i_7) = loc_0 + [N + strlen(m_0), N + strlen(m_0) + 1]$. *How this mapping is found is discussed in the rest of this section.*

Abstract semantics for GR, and concretisation. The abstract semantics of each instruction in our core language is given by Figure 9. Figure 9 defines a system of equations whose fixed point gives us an approximation on the locations that each pointer may dereference. We remind the reader of our notation: $[l,u]_\downarrow = l$, and $[l,u]_\uparrow = u$. In Figure 9, this notation surfaces in the semantics of intersections. The abstract interpretation of the pointer-related instructions in Figure 7 yields the results discussed in Example 3.

$$
\begin{array}{rcl}
j : p = \text{ malloc } (v) \text{ with } v \text{ scalar} & \Rightarrow & \mathrm{GR}(p) = (\bot, \ldots, \underbrace{[0,0]}_{j^{th} component}, \ldots) \\
p = \text{ free } (v) \text{ with } v \text{ scalar} & \Rightarrow & \mathrm{GR}(p) = (\bot, \ldots, \bot) \\
v = v_1 & \Rightarrow & \mathrm{GR}(v) = \mathrm{GR}(v_1) \\
q = p + c \text{ with } c \text{ scalar variable} & \Rightarrow & \begin{cases} \mathrm{GR}(q) = (q_0, \ldots, q_{n-1}) \text{ with} \\ q_i = \begin{cases} \bot \text{ if } p_i = \bot \\ p_i + R(c) \text{ else} \end{cases} \end{cases} \\
q = \phi(p^1, p^2) & \Rightarrow & \mathrm{GR}(q) = \mathrm{GR}(p^1) \sqcup \mathrm{GR}(p^2) \\
q = p^1 \cap [-\infty, p^2] & \Rightarrow & \begin{cases} \mathrm{GR}(q) = (q_0, \ldots, q_{n-1}) \text{ with} \\ q_i = \begin{cases} \bot \text{ if } (p_i^1 = \bot \text{ or } p_i^2 = \bot) \\ p_i^1 \sqcap [-\infty, {p_i^2}_\uparrow] \text{ else} \end{cases} \end{cases} \\
q = p^1 \cap [p^2, +\infty] & \Rightarrow & \begin{cases} \mathrm{GR}(q) = (q_0, \ldots, q_{n-1}) \text{ with} \\ q_i = \begin{cases} \bot \text{ if } (p_i^1 = \bot \text{ or } p_i^2 = \bot) \\ p_i^1 \sqcap [{p_i^2}_\downarrow, +\infty] \text{ else} \end{cases} \end{cases} \\
q = *p & \Rightarrow & \mathrm{GR}(q) = ([-\infty, \infty], \ldots, [-\infty, \infty]) \\
*q = p & \Rightarrow & \text{Nothing}
\end{array}
$$

Figure 9. Constraint generation for GR with $\mathrm{GR}(p) = (p_0, \ldots, p_{n-1})$ given p in the right hand side of rules

There remains to define how the abstract states will be concretised ($@(\mathsf{loc}_i)$ is the actual address returned by the i^{th} malloc):

DEFINITION 3 (Concretisation). *Given $GR(p)$ an abstract value (a set of "abstract addresses for p"), denoted by $GR(p) = (p_0, \ldots, p_{n-1})$ then we define its concretisation as follows:*

$$\gamma(GR(p)) = \bigcup_{i \in supp_{GR}(p)} \{@(loc_i) + o, o \in p_i\}$$

The concretisation function of this abstract value is thus a set of (concrete) addresses, obtained by shifting a set of base addresses by a certain value in SymbRanges [1].

PROPOSITION 1. *(α, γ) is a Galois connection.*

PROOF 1. *Immediate since $(\alpha_{SymbRanges}, \gamma_{SymbRanges})$ is a galois connection.*

Solving the abstract system of contraints Following the abstract interpretation framework, we solve our system of constraints by computing for each pointer a growing set of abstract values until convergence.

However, as the underlying lattice SymbRanges has infinite height, widening is necessary to ensure that these sequence of iterations actually terminate. Our widening operation on pointers generalizes the widening operation on ranges. It is defined as follows:

DEFINITION 4. *Given $GR(p)$ and $GR(p')$ with $GR(p) \sqsubseteq GR(p')$, we define the widening operator:*

$$GR(p) \nabla GR(p') = (p_0 \nabla p'_0, \ldots, p_{n-1} \nabla p'_{n-1}),$$

where ∇ denotes the widening on SymbRanges, extended with $\bot \nabla \bot = \bot$ and $\bot \nabla [l,u] = [l,u]$.

As usual, we only apply the widening operator on a cut set of the control flow graphs (here, only on ϕ functions).

Widening may lead our interpreter to produce very imprecise results. To recover part of this imprecision, we use a descending sequence of finite size: after convergence, we redo a step of symbolic evaluation of the program, starting from the value obtained after convergence. One example of analysis will be detailed later, in Section 3.9.

The abstract interpretation of loads and stores. In Figure 9, we chose not to track precisely the intervals associated with pointers stored in memory. In other words, when interpreting stores, e.g., $q = *p$, we assign the top value of our lattice to q. This decision is pragmatic. As we shall explain in Section 4, a typical compilation infra-structure already contains analyses that are able to track the propagation of pointer information throughout memory. Our goal is not to solve this problem. We want to deliver a fast analysis that is precise enough to handle C-style pointer arithmetic.

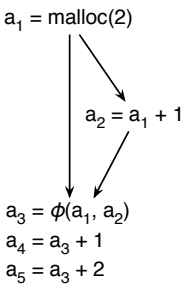
3.5 Answering GR Queries

Our queries are based on the following result, that is an immediate consequence of the fact our analysis is an abstract interpretation:

PROPOSITION 2 (Correctness). *Let p and p' be two pointers in a given program then :*

$$\begin{array}{ll} \textit{if} & supp_{GR}(p) \cap supp_{GR}(p') = \emptyset \\ \textit{or} & \forall i \in supp_{GR}(p) \cap supp_{GR}(p'), p_i \sqcap p'_i = \emptyset \end{array}$$

[1] While speaking about symbolic ranges, we also have to concretize the values involved in the bounds of p_i, that is we shall use the actual values between $S(p_{i\downarrow})$ and $S(p_{i\uparrow})$.



	Global Analysis	Local Analsys
a_1 = malloc(2)	GR(a_1) = {loc$_1$ + [0, 0]}	LR(a_1) = loc$_1$ + [0, 0]
$a_2 = a_1 + 1$	GR(a_2) = {loc$_1$ + [1, 1]}	LR(a_2) = loc$_1$ + [1, 1]
$a_3 = \phi(a_1, a_2)$	GR(a_3) = {loc$_1$ + [0, 1]}	LR(a_3) = loc$_2$ + [0, 0]
$a_4 = a_3 + 1$	GR(a_4) = {loc$_1$ + [1, 2]}	LR(a_4) = loc$_2$ + [1, 1]
$a_5 = a_3 + 2$	GR(a_5) = {loc$_1$ + [2, 3]}	LR(a_5) = loc$_2$ + [2, 2]

Figure 10. Example that illustrates imprecision of global analysis due to lack of path-sensitiveness.

then $\gamma(GR(p)) \cap \gamma(GR(p')) = \emptyset$.

In other words, if the abstract values of two different pointers of the program have a null intersection, then the two *concrete pointers* do not alias. This result is directly implied by the abstract interpretation framework. Thanks to this result, we implement the query $Q_{\text{GR}}(p, p')$ as :

- If GR(p) and GR(p') have an empty intersection, then "they do not alias".
- Else "they may alias".

3.6 Local Range Analysis of Pointers

The global pointer analysis is not path sensitive. As a consequence, this analysis cannot, for instance, distinguish the effects of different iterations of a loop upon the actual value of a pointer, or the effects of different branches of a conditional test on that very pointer. The program in Figure 10 illustrates this issue. Pointers a_4 and a_5 clearly must not alias. Yet, their abstract states have non-empty intersections for loc_1. Therefore, the query mechanism of Section 3.5 would return a "may-alias" result in this case.

To solve this problem, we have developed a *local* version of our pointer analysis. We call it local because it creates new locations for every ϕ-function. Our local range analysis is simpler than its global counterpart. We solve it in a single iteration of abstract interpretation applied on the instructions of our core language. Instructions are evaluated abstractly in the order given by the program's dominance tree. Figure 11 gives the abstract semantics of each instruction. The abstract value LR(p) exists in $(\mathcal{L}oc \cup \mathcal{N}ewLocs) \times$ SymbRanges where $\mathcal{N}ewlocs$ denotes a set of "fresh location variables", that are computed by invocation of the function $\mathcal{N}ewLocs()$. As before, we write loc $+ R$ instead of (loc, R). Similarly to γ_{GR}, γ_{LR} denotes the set of abstract addresses from $@(\text{loc}) + R_{\downarrow}$ to $@(\text{loc}) + R_{\uparrow}$.

To find a solution to the local analysis, we solve the system provided by the abstract rules seen in Figure 11. This resolution process involves computing an increasing sequence of abstract values for each pointer p of the program. Contrary to the global analysis, this analysis is based on a finite lattice, we do not need any widening operator. Figure 10 (Right) shows the result of the local analysis. Contrary to the global analysis, we have a new location bound to variable a_3, which is defined by a ϕ operator. The range of this new location is $[0, 0]$. The other variables that are functions of a_3, e.g., a_4 and a_5, have now non-intersection ranges associated with this new memory name.

$$
\begin{array}{rcl}
p = \text{ malloc } (v) \text{ with } v \text{ scalar} & \Rightarrow & \text{LR}(p) = \mathcal{N}ewLocs() + [0,0] \\
v = v_1 & \Rightarrow & \text{LR}(v) = \text{LR}(v_1) \\
q = p + c \text{ with } c \text{ scalar variable and } \text{LR}(q) = \text{loc}_i + [l,u] & \Rightarrow & \text{LR}(q) = \text{loc}_i + ([l,u] + R(c)) \\
j : q = \phi(p_1, p_2) & \Rightarrow & \text{LR}(q) = \mathcal{N}ewLocs() + [0,0] \\
q = p_1 \cap [-\infty, p_2] \text{ or } q = p_1 \cap [p_2, +\infty] & \Rightarrow & \text{LR}(q) = \text{LR}(p_1) \\
q = *p_1 & \Rightarrow & \text{LR}(q) = \mathcal{N}ewLocs() + [0,0] \\
*q = p_1 & \Rightarrow & \text{Nothing}
\end{array}
$$

Figure 11. Constraint generation for LR

3.7 Answering LR Queries

The correction for the local analysis is stated by the following proposition:

PROPOSITION 3 (Correctness). *Let p and p' be two pointers in a given program, and γ_{LR} be the concretization of the abstract map LR, which we state like in Definition 3. If LR(p) = loc $+ R$ and LR(p') = loc$'$ $+ R'$, then if loc = loc$'$ and $R \sqcap R' = \emptyset$ then γ(LR$(p)) \cap \gamma$(LR$(p') = \emptyset$. In other words, p and p' never alias.*

Thanks to this result, we implement the query $Q_{\text{LR}}(p, p')$:

- If LR(p) and LR(p') have a common base pointer with ranges that do not intersect, then "they do not alias".
- Else "they may alias".

3.8 Complexity

The e-SSA representation ensures that we can implement our analysis sparsely. Sparsity is possible because the e-SSA form renames variables at each program point where new abstract information, e.g., ranges of integers and pointers, arises. According to Tavares *et al.* [23], this property – single information – is sufficient to enable sparse implementation of non-relational static analyses [23]. Therefore, the abstract state of each variable is invariant along the entire live range of that variable. Consequently, the space complexity of our static analysis is $O(|V| \times I)$, where V is the set of names of variables in the program in e-SSA form, and I is a measure of the size of the information that can be bound to each variable.

We apply widening after one iteration of abstract interpretation. Thus, we let the state of a variable to change first

from $[\bot, \bot]$ to $[s_l, s_u]$, where $s_l \neq -\infty$, and $s_u \neq +\infty$. From there, we can reach either $[-\infty, s_u]$ or $[s_l, +\infty]$. And, finally, this abstract state can jump to $[-\infty, +\infty]$. Hence, our time complexity is $O(3 \times |V|) = O(|V|)$. This observation also prevents our algorithm from generating expressions with very long chains of "min" and "max" expressions. Therefore, I, the amount of information associate with a variable, can be represented with $O(1)$ space. As a consequence of this frugality, our static analysis runs in $O(|V|)$ time, and requires $O(|V|)$ space.

3.9 A Complete Example

Example 4 shows how our analysis works on the program seen in Figure 1.

EXAMPLE 4. *Figure 7 shows the control flow graph (CFG) of the program in Figure 1. Our graph is in e-SSA form [5]. Figure 12 shows the result of widening ranges after one round of abstract interpretation (stabilization achieved), and a descending sequence of size two. Our system stabilizes after each instruction is visited four times. The first visit does initialization, the second widening (and stabilization check), and the last two build the descending sequence.*

This example illustrates the need of widening to ensure termination. Our program has a cycle of dependencies between pointers i_1, i_2 and i_3. If not for widening, pointer i_3, incremented in line 5 of Figure 1 would grow forever. Thus, as in Abstract Interpretation, we must break the cyclic dependences between our pointers under analysis, by means of insertion of widening points (identify points in the CFG where to apply widening to insure convergence).

Returning to our example of Figure 1, we are interested in knowing, for instance, that the memory access at line 6 is independent on the accesses that happen at line 10. To achieve this goal, we must bound the memory regions covered by pointers i_3 and i_7. A cyclic dependence happens at the operation `i++`, because in this case, we have a pointer being used as both, source and destination of the update. Thus, we should have inserted a widening point at stores and load instructions. However, in the Abstract Interpreter depicted in Figure 9, it was sufficient to insert widening points at ϕ functions (as we already said before) because :

- heads of loops are ϕ functions (thus dependencies between variables of different iteration of loops are broken).
- we are working on (e-)SSA form programs; thus, the only inter-loop dependencies are successive stores to the same variable : `*q=...`, `*q=...`. The value GR(q) is the union of all information gathered inside the loop. (In essence, memory addresses *are not* in static single assignment form, i.e., we could have the same address being used as the target of a store multiple times). This information might grow forever; hence, we would have inserted a widening point on the last write. In our case, the information we store is already the top of our lattice; hence, there is no need for widening.

	Var	GR	LR
Starting state	b, p, i_0	$([0,0], \bot)$	$\mathsf{loc}_0 + [0,0]$
	m_0, s	$(\bot, [0,0])$	$\mathsf{loc}_1 + [0,0]$
	i_1	$([0,0], \bot)$	$\mathsf{loc}_2 + [0,0]$
	i_2	$([0,0], \bot)$	$\mathsf{loc}_2 + [0,0]$
	t_0	$([1,1], \bot)$	$\mathsf{loc}_2 + [1,1]$
	e	$([N,N], \bot)$	$\mathsf{loc}_0 + [N,N]$
	i_3	$([1,1], \bot)$	$\mathsf{loc}_2 + [1,1]$
	i_4	$(\bot, \bot)$	$\mathsf{loc}_2 + [0,0]$
	f	$([k,k], \bot)$	$\mathsf{loc}_0 + [k,k]$
	m_1	$(\bot, [0,0])$	$\mathsf{loc}_3 + [0,0]$
	m_2	$(\bot, [1,1])$	$\mathsf{loc}_3 + [1,1]$
	i_5	$(\bot, \bot)$	$\mathsf{loc}_4 + [0,0]$
	i_6	$(\bot, \bot)$	$\mathsf{loc}_4 + [0,0]$
	i_7	$(\bot, \bot)$	$\mathsf{loc}_4 + [1,1]$
Growing iterations + widening	i_1	$([0,+\infty], \bot)$	
	i_2	$([0,+\infty], \bot)$	
	t_0	$([1,+\infty], \bot)$	
	i_3	$([1,+\infty], \bot)$	
	i_4	$([N,+\infty], \bot)$	
	m_1	$(\bot, [0,+\infty])$	
	m_2	$(\bot, [1,+\infty])$	
	i_5	$([N,+\infty], \bot)$	
	i_6	$([N,k-1], \bot)$	
	i_7	$([N+1,k], \bot)$	
After one descending step	i_2	$([0,N-1], \bot)$	
	t_0	$([1,N], \bot)$	
	i_3	$([1,N], \bot)$	
	m_1	$(\bot, [0,+\infty])$	
	m_2	$(\bot, [1,+\infty])$	
After two descending steps	i_1	$([0,N], \bot)$	
	i_3	$([1,N], \bot)$	
	i_4	$([N,N], \bot)$	
	i_5	$([N,k], \bot)$	
	i_6	$([k-1,k], \bot)$	
	i_7	$([k,k+1], \bot)$	

Figure 12. Abstract interpretation of CFG seen in Figure 7 (program in Figure 1). For GR, we associate loc_0 with the `malloc` at line 17 and loc_1 with the `malloc` at line 18 (of the program). Only changes in GR and LR are rewritten after the growing and descending iterations. We let k = N+`strlen`(m_0).

4. Experiments

We have implemented our range analysis in the LLVM compiler, version 3.5. In this section, we show a few numbers that we have obtained with this implementation. All our experiments have been performed on an Intel i7-4770K, with 8GB of memory, running Ubuntu 14.04.2. Our goal with these experiments is to show: (i) that our alias analysis is more precise than other alternatives of practical runtime; and (ii) that it scales up to large programs.

On the Precision of our Analysis. In this section, we compare our analysis against the other pointer analyses that are available in LLVM 3.5, namely *basic* and *SCEV*. The first of them, although called "basic", 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[2]:

[2] This list has been taken from the LLVM documentation, available at `http://llvm.org/docs/AliasAnalysis.html` in September of 2015

- 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 differing subscripts cannot alias.
- Many common standard C library functions never access memory or only read memory.
- Function calls cannot reference stack allocations which never escape from the function that allocates them

As we see from the above list, the *basic* alias analysis has some of the capabilities of the technique that we present in this paper, namely the ability to distinguish fields and indices within aggregate types. In this case, such disambiguation is only possible when the aggregates are indexed with constants known at compilation time. For situations when these indices are symbols, LLVM relies on a second kind of analysis to perform the disambiguation: the "scalar-evolution-based" (SCEV) alias analysis. This analysis tries to infer closed-form expressions to the induction variables used in loops. For each loop such as:

```
for (i = B; i < N; i += S) { ... a[i] ... }
```

this analysis associates variable `i` with the expression $\texttt{i} = \texttt{B} + iter \times S, \texttt{i} \leq \texttt{N}$. The parameter *iter* represents the current iteration of the loop. With this information, SCEV can track the ranges of indices which dereference array `a` within the loop. Contrary to our analysis, SCEV is only effective to disambiguate pointers accessed within loops and indexed by variables in the expected closed-form.

Figure 13 shows how the three different analyses fare when applied on larger benchmarks. For this experiment we have chosen three benchmarks that have been used in previous work that compares pointer analyses: Prolangs [20], PtrDist [28] and MallocBench [9]. We first notice that in general all the pointer analyses in LLVM disambiguate a relatively low number of pointers. This happens because many pointers are passed as arguments of functions, and, not knowing if these functions will be called from outside the program, the analyses must, conservatively, assume that these parameters may alias. Second, we notice that our pointer analysis is one order of magnitude more precise than the scalar-evolution based implementation available in LLVM. Finally, we notice that we are able to disambiguate more queries than the *basic* analysis. Furthermore, our results complements it in non-trivial ways. In total, we tried to disambiguate 3.093 million pairs of pointers. Our analysis found out that 1.29 million pairs reference non-overlapping regions. The *basic* analysis has been able to distinguish 953 thousand pairs. By combining these two analyses, we extended this number to 1.439 thousand pairs of pointers. SCEV could not increase this number any further.

Figure 14 shows the proportion of queries that we have been able to disambiguate with the global test of Section 3.4.

Program	#Queries	%scev	%basic	%rbaa	%(r + b)
cfrac	89,255	0.87	9.70	16.65	21.03
espresso	787,223	2.39	12.62	28.16	33.04
gs	608,374	15.56	40.67	56.18	59.99
allroots	974	16.32	64.37	79.77	79.88
archie	159,051	0.98	20.57	16.44	28.04
assembler	35,474	2.16	40.31	47.86	55.61
bison	114,025	0.74	10.95	9.56	14.74
cdecl	301,817	13.74	24.80	49.72	50.73
compiler	9,515	0.49	67.27	67.27	69.20
fixoutput	3,778	0.11	88.30	83.17	90.37
football	495,119	3.58	59.20	60.08	65.08
gnugo	13,519	9.23	60.89	78.21	79.29
loader	13,782	2.32	29.55	36.47	46.09
plot2fig	27,372	2.90	24.09	46.45	49.54
simulator	25,591	3.56	46.32	41.25	52.27
unix-smail	61,246	1.22	37.36	42.92	48.95
unix-tbl	85,339	7.30	44.38	33.92	48.83
anagram	3,114	2.18	32.85	53.31	59.54
bc	198,674	14.14	30.95	47.86	50.01
ft	7,660	2.73	5.23	24.65	25.91
ks	14,377	0.61	22.98	21.60	27.70
yacr2	38,262	0.20	7.22	12.83	14.48
Total	3,093,541	6.97	30.83	41.73	46.53

Figure 13. Comparison between three different alias analyses. We let **r + b** be the combination of our technique and the *basic* alias analysis of LLVM. Numbers in **scev**, **basic**, **rbaa** and **r+b** show percentage of queries that answer "no-alias".

The two columns **noalias** of Figure 14 correspond to the percentage in column **%rbaa** applied on the column **#Queries** of Figure 13. Overall, the global test has given us 239,008, out of 1,290,457 "no-alias" answers. This corresponds to 18.52% of all the pairs of pointers that we have disambiguated. We did not show the local test in this table because these two tests are not directly comparable. The global test disambiguate pointers, and the local test disambiguate the addresses used in instructions such as loads and stores. These instructions can use pointers that might dereference overlapping regions; however, not at the same moment dur-

Prog	noalias	global	Prog	noalias	global
cfrac	14,865	1,102	gnugo	10,573	1,851
espresso	221,416	20,791	loader	5,026	433
gs	341,532	106,859	plot2fig	12,713	861
allroots	777	182	simulator	10,557	1,092
archie	26,142	2,034	unix-smail	26,289	771
assembler	16,977	905	unix-tbl	28,948	1,136
mybison	10,905	1,417	anagram	1,660	88
cdecl	150,050	43,619	bc	95,091	32,498
compiler	6,401	156	ft	1,888	452
fixoutput	3,142	4	ks	3,105	218
football	297,491	22,052	yacr2	4,909	487

Figure 14. Number of queries solved with the global test of Section 3.4. Column **noalias** gives the number of queries that we have been able to disambiguate, and column **global** shows how many queries were solved with the global test.

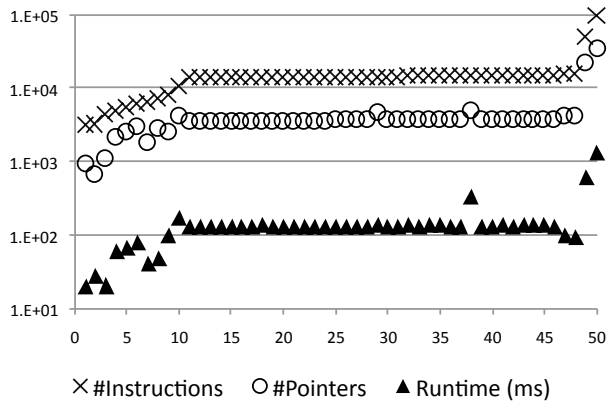


Figure 15. Runtime of our analysis for the 50 largest benchmarks in the LLVM test suite. Each point on the X-axis represents a different benchmark. Benchmarks are ordered by size. This experiment took less than 10 seconds.

ing the execution of the program. The local test has been able to disambiguate 6.55% of all the addresses used in our benchmarks. The rest of the disambiguation was obtained by comparing off-sets from different locations.

On the Scalability of our Analysis. The chart in Figure 15 shows how our analysis scales when applied on programs of different sizes. We have used the 50 largest programs in the LLVM benchmark suite. These programs gave us a total of 800,720 instructions in the LLVM intermediate representation, and a total of 241,658 different pointer variables. We analyzed all these 50 programs in 8.36 seconds. We can – effectively – analyze 100,000 instructions in about one second. In this case, we are counting only the time to map variables to values in SymbRanges . We do not count the time to query each pair of pointers, because usually compiler optimizations perform these queries selectively, for instance, only for pairs of pointers within a loop. Also, we do not count the time to run the out-of-the-box implementation of range analysis mentioned in Section 3.3, because our version of it is not implemented within LLVM. It runs only once, and we query it afterwards, never having to re-execute it.

The chart in Figure 15 provides strong visual indication of the linear behavior of our algorithm. We have found, indeed, cogent evidence pointing in this direction. The linear correlation coefficient (R) indicates how strong is a linear relationship between two variables. The closer to one, the more linear is the correlation. The linear correlation between time and number of instructions for the programs seen in Figure 15 is 0.982, and the correlation between time and number of pointers is 0.975.

5. Related Work

The contribution of this work is a new representation of pointers, based on the SymbRanges lattice, and an algorithm to reach a fixed point in this lattice, based on abstract interpretation. This contribution complements classic work on pointer analysis. In other words, our representation of pointers can be used to enhance the precision of algorithms such as Steensgard's [22], Andersen's [2], or even the state-of-the-art technique of Hardekopf and Lin [11]. 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 [10], or wave propagation [18].

There exist previous work that used similar lattices as ours, albeit different resolution algorithms. For instance, much of the work on automatic parallelization has some way to associate symbolic offsets, usually loop bounds, with pointers. Michael Wolfe [24, Ch.7] and Aho et al. [1, Ch.11] have entire chapters devoted to this issue. The key difference between our work and this line of research is the algorithm to solve pointer relations: they resort to integer linear programming (ILP) or the Greatest Common Divisor test to solve diophantine equations, whereas we do abstract interpretation. Even Rugina and Rinard [19], 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 work to pointers within loops, we can analyze programs with over one million assembly instructions in a few seconds.

There exist work that, like ours, also associates intervals with pointers, and solves static analysis via abstract interpretation techniques. However, to the best of our knowledge, these approaches have a fundamental difference to our work: they use integer intervals a là Cousot [7], whereas we use symbolic intervals. The inspiration for much of this work springs from Balakrishnan and Reps notion of *Value Set* Analysis [3]. Integer intervals have also being used by Yong *et al.* [26] and, more recently, by Oh *et al.* [16]. In the latter case, Oh *et al.* use pointer disambiguation incidentally, to demonstrate their ability to implement efficiently static analyses in a context-sensitive way. Even though integer ranges fit well the need of machine code, as demonstrated by Balakrishnan and Reps, we believe that further precision requires more expressive lattices. We have not implemented value set analysis, but we have tried a simple experiment: we counted the number of pointers that have integer ranges, and compared this number against the quantity of pointers that have symbolic ranges. We found out that 20.47% of the pointers in our three benchmark suites have exclusively symbolic ranges. Classic range analysis would not be able to distinguish them. Notice that numeric ranges are more common among pointer variables than among integer, because fields within `structs` – a very common construct in C – are indexed through integers. Finally, the fact that we use

Bodik's e-SSA form [5] distinguish our abstract interpretation algorithm from previous work. This representation lets us solve our analysis sparsely, whereas Balakrishnan's algorithm works on a dense representation that associates facts with pairs formed by variables and program points.

6. Conclusion

In this paper we have presented a new alias analysis technique that handles, within the same theoretical framework, the subtleties of pointer arithmetic and memory indexation. Our technique can disambiguate regions within arrays and C-like structs using the same abstract interpreter. We have achieved precision in our algorithm by combining alias analysis with classic range analysis on the symbolic domain. Our analysis is fast, and handles cases that the implementations of pointer analyses currently available in LLVM cannot deal with. In future work, we plan to investiguate better splitting strategies and other more expressive lattices to improve the global precision of our analyses.

Acknowledgment

This project is supported by CNPq, Intel, 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

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

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

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

[4] W. Blume and R. Eigenmann. Symbolic range propagation. In *In IPPS*, pages 357–363, 1994.

[5] R. Bodik, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *In PLDI*, pages 321–333. ACM, 2000.

[6] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *In POPL*, pages 55–66. ACM, 1991.

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

[8] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

[9] D. Grunwald, B. Zorn, and R. Henderson. Improving the cache locality of memory allocation. In *In PLDI*, pages 177–186. ACM, 1993.

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

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

[12] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *In PASTE*, pages 54–61. ACM, 2001.

[13] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *In POPL*, pages 66–74. ACM, 1982.

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

[15] H. Nazaré, I. Maffra, W. Santos, L. Barbosa, L. Gonnord, and F. M. Q. Pereira. Validation of memory accesses through symbolic analyses. In *In OOPSLA*, pages 791–809. ACM, 2014.

[16] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi. Selective context-sensitivity guided by impact pre-analysis. In *In PLDI*, pages 475–484. ACM, 2014.

[17] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis for C. In *In PASTE*, pages 37–42, 2004.

[18] F. M. Q. Pereira and D. Berlin. Wave propagation and deep propagation for pointer analysis. In *In CGO*, pages 126–135. IEEE, 2009.

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

[20] B. G. Ryder, W. A. Landi, P. A. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Trans. Program. Lang. Syst.*, 23(2):105–186, 2001.

[21] L. Shang, X. Xie, and J. Xue. On-demand dynamic symmary-based points-to analysis. In *In CGO*, pages 264–274. ACM, 2012.

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

[23] A. L. C. Tavares, B. Boissinot, F. M. Q. Pereira, and F. Rastello. Parameterized construction of program representations for sparse dataflow analyses. In *In Compiler Construction*, pages 2–21. Springer, 2014.

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

[25] D. Yan, G. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for java. In *In ISSTA*, pages 155–165. ACM, 2011.

[26] S. H. Yong and S. Horwitz. Pointer-range analysis. In *In SAS*, pages 133–148. Springer, 2004.

[27] Q. Zhang, X. Xiao, C. Zhang, H. Yuan, and Z. Su. Efficient subcubic alias analysis for C. In *In OOPSLA*, pages 829–845. ACM, 2014.

[28] Q. Zhao, R. Rabbah, and W.-F. Wong. Dynamic memory optimization using pool allocation and prefetching. *SIGARCH Comput. Archit. News*, 33(5):27–32, 2005.