ANÁLISE SIMBÓLICA DE INTERVALOS DE PONTEIROS

VITOR MENDES PAISANTE

ANÁLISE SIMBÓLICA DE INTERVALOS DE PONTEIROS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Fernando Magno Quintão Pereira
Coorientador: Leonardo Barbosa e Oliveria

Belo Horizonte
Julho de 2016

VITOR MENDES PAISANTE

SYMBOLIC RANGE ANALYSIS OF POINTERS

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Fernando Magno Quintão Pereira
Co-Advisor: Leonardo Barbosa e Oliveria

Belo Horizonte

July 2016

Mendes Paisante, Vitor

??? Symbolic Range Analysis of Pointers / Vitor Mendes Paisante. — Belo Horizonte, 2016
xxiv, 46 f. : il. ; 29cm

Dissertação (mestrado) — Federal University of Minas Gerais
Orientador: Fernando Magno Quintão Pereira

1. Computação. 2. Compiladores. 3. Análise Estática I. Orientador. II. Título.

CDU ???

[Folha de Aprovação]

Quando a secretaria do Curso fornecer esta folha,
ela deve ser digitalizada e armazenada no disco em formato gráfico.

Se você estiver usando o `pdflatex`,
armazene o arquivo preferencialmente em formato PNG
(o formato JPEG é pior neste caso).

Se você estiver usando o `latex` (não o `pdflatex`),
terá que converter o arquivo gráfico para o formato EPS.

Em seguida, acrescente a opção `approval={`*nome do arquivo*`}`
ao comando `\ppgccufmg`.

Se a imagem da folha de aprovação precisar ser ajustada, use:
`approval=[`*ajuste*`][`*escala*`]{`*nome do arquivo*`}`
onde *ajuste* é uma distância para deslocar a imagem para baixo
e *escala* é um fator de escala para a imagem. Por exemplo:
`approval=[-2cm][0.9]{`*nome do arquivo*`}`
desloca a imagem 2cm para cima e a escala em 90%.

I dedicate this work to everyone that helped in any capacity, to my family and to my friends.

Acknowledgments

This project was supported by the Brazilian Ministry of Science and Technology through CNPq, the Intel Corporation (the ISRA eCoSoC project) and the INRIA-FAPEMIG cooperation grant (The Prospiel project).

We'd like to thank Maroua Maalej and Laure Gonnord for their collaboration in our CGO16 article and this work. They handled the theoretical proofs of our analyses. And we also thank all other members of our research project.

Our acknowledgments also go to the Federal University of Minas Gerais Computer Sciences Department and the Laboratoire de l'Informatique du Parallelisme (LIP) for providing the help and infrastructure necessary for this research.

"'Are we nearly there?' Alice managed to pant out at last.
'Nearly there!' the Queen repeated. 'Why, we passed it ten minutes ago! Faster!"'
(Lewis Carroll)

Resumo

Análise de ponteiros é uma das técnicas mais fundamentais que compiladores utilizam para otimizar linguagens imperativas, especialmente linguagens orientadas a objetos. No entanto, mesmo com toda a atenção que este tópico já recebeu, as propostas do estado da arte atual presentes em compiladores ainda lidam com desafios em relação à precisão e velocidade. Em particular, aritmética de ponteiros, um fator chave de linguagens como C e C++, ainda precisa de soluções satisfatórias neste campo. Este trabalho apresenta um novo algoritmo para análise de ponteiros para resolver esse problema. O ponto chave da nossa proposta é combinar análise de ponteiros com análise simbólica de largura de inteiros. Tal combinação nos permite desambiguar campos dentro de vetores e estruturas de dados, de maneira efetiva obtendo maior precisão do que algoritmos tradicionais. Para validar nossa técnica, implementamos nosso algoritmo no compilador LLVM. Testes em uma vasta gama de benchmarks nos mostraram que podemos desambiguar vários tipos de estruturas em C que as análises atuais do estado da arte não conseguem lidar. Em particular, podemos desambiguar 1.35x mais comparações do que as análises de ponteiros presentes no LLVM. Além disso, nossa análise é rápida: podemos lidar com um milhão de instruções em assembly em 10 segundos.

Palavras-chave: Análise de Ponteiros, Análise Estática, Compiladores.

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 work 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.

Keywords: Alias Analysis, Static Analysis, Compilers.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Context

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 [Hind, 2001]. This research has contributed to make the present algorithms more precise [Hardekopf and Lin, 2007; Zhang et al., 2014], and faster [Hardekopf and Lin, 2011; Shang et al., 2012]. 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.

1.2 Problem

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 [Yong and Horwitz, 2004]. Figure 1.1 shows an example that state-of-the-art pointer analyses tend not to deal with satisfactorily. In the example, both stores on the r array posses different offsets and they do not alias. To figure out that such offsets are disjoint it is necessary to verify their ranges and add a new layer of analysis to the current pointer analyses available.

Field-sensitive pointer analysis, provide a partial solution to this problem. These

```
1:  char* duplicate (int size, char* v) {
2:    if(size > 0) {
3:      char* r = malloc(size*2);
4:      int i;
5:      for(i = 0; i < size; i++) {
6:        r[i] = v[i];
7:        r[i+size] = v[i];
8:      }
9:    }
10:   else return NULL;
11: }
```

Figure 1.1. Example that state-of-the-art pointer analyses handle unsatisfactorily.

analyses can distinguish different fields within a record, such as a struct in C [Pearce et al., 2004], or a class in Java [Yan et al., 2011]. However, they rely on syntax that is usually absent in the low level program representations adopted by compilers. Shape analyses [Jones and Muchnick, 1982; Sagiv et al., 1998] 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 [Wolfe, 1996, 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.

1.3 Solution

This work describes such an analysis. We introduce an abstract domain that associates pointers with symbolic ranges. In other words, for each pointer p we conservatively estimate the range of memory slots that can be addressed as an offset of p. We let $\mathrm{GR}(p)$ be the global abstract address set associated with pointer p, such that if $\mathsf{loc}_i + [l, u] \in \mathrm{GR}(p)$, 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 [Andersen, 1994] to the more recent technique of Zhang *et al.* [Zhang et al., 2014], 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 our research 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's symbols, a symbol being either a constant or the name of a variable. There exist many approaches to symbolic range analyses in the literature [Blume and Eigenmann, 1994; Nazaré et al., 2014; Rugina and Rinard, 2005]. The algorithms that we present in this work 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 [Blume and Eigenmann, 1994].

1.4 Summary of experimental results

To validate our ideas, we have implemented them in the LLVM compilation infrastructure [Lattner and Adve, 2004]. We have tested our pointer analysis onto three different benchmarks used in previous work related to pointer disambiguation: Prolangs [Ryder et al., 2001], PtrDist [Zhao et al., 2005] and MallocBench [Grunwald et al., 1993]. As we show in Chapter 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.

1.5 Summary of publications

The analysis described here was published on the International Symposium on Code Generation and Optimization (CGO) of 2016 held in Barcelona, Spain [Paisante et al., 2016].

The technology behind it is also present on two other publications [Paisante et al., 2014; Saggioro et al., 2015]. In these papers, the algorithm presented here was used to infer the layout and the content of buffers transfered through the network. This was useful for verifying, in a safe communication line, if the information transfered between two programs through a network should be considered potentially dangerous or not. If proven not dangerous, guards for checking integer overflows may not be necessary. These articles proposed methods for such verification to be used on the internet of things (IoT), where simple devices could run significantly faster with a

reduced number of integer overflow checks. The layout and content inference analysis used in these papers differed from our current approach by using a numerical range analysis, since a symbolic approach would not be of relevance for such application.

1.6 Next Chapters

This work is divided in the following structure:

- Chapter 1 presented an introduction describing the context of points-to analysis, the problem faced by this work, our solution of adding a layer of symbolic range analysis to points-to analysis, and our publications.
- Chapter 2 presents the background behind our work, the concepts and other related works. It also does a literature review of other related publications. The topics and concepts it covers are the SSA form, the LLVM compiler framework, points-to analysis, range analysis, the use of pointer disambiguation for automatic parallelization of code and the merging of points-to analysis with range analysis.
- Chapter 3 describes our points-to analysis. It gives an overview of our algorithm, shows how we combine the symbolic range analysis with pointer analysis, describes the local and global parts of our work, discusses about the algorithm's complexity and end with an example.
- Chapter 4 shows our experiments and results.
- Chapter 5 concludes this dissertation. It shows the limitations of our work, discusses future work to be done and issues final conclusions.

Chapter 2

Background

2.1 Lattice

A lattice is an algebraic structure that consists of a partially ordered set in which every two elements have a unique least upper bound (or join) and a unique greatest lower bound (or meet) [Nation, 2016]. An example is given by the natural numbers, partially ordered by divisibility, for which the unique least upper bound is the least common multiple and the unique greatest lower bound is the greatest common divisor. An example can be seen in the figure below.

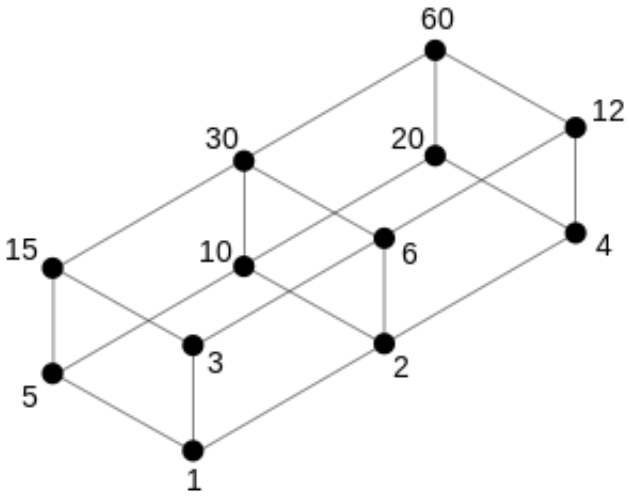


Figure 2.1. Example of a lattice of integer divisors of 60.

If $L = (S, \leq, \vee, \wedge, \bot, \top)$ is a complete lattice, and $e_1 \in S$ and $e_2 \in S$, then we let $e_{lub} = (e_1 \vee e_1) \in S$ be the least upper bound of the set $\{e_1, e_2\}$. The least upper bound e_{lub} has the following properties:

- $e_1 \leq e_{lub}$ and $e_2 \leq e_{lub}$
- For any element $e' \in S$, if $e_1 \leq e'$ and $e_2 \leq e'$, then $e_{lub} \leq e'$

In the same way, we let $e_{glb} = (e_1 \wedge e_1) \in S$ be the greatest lower bound of the set $\{e_1, e_2\}$. The greatest lower bound e_{glb} has the following properties:

- $e_{glb} \leq e_1$ and $e_{glb} \leq e_2$
- For any element $e' \in S$, if $e' \leq e_1$ and $e' \leq e_2$, then $e' \leq e_{glb}$

While the symbols $\vee$ and $\wedge$ correspond to the join and meet operators respectively. $\perp$ and $\top$ correspond to bottom and top respectively. Bottom ($\perp$) is the multiplicative one for the join operator and the multiplicative zero for the meet operator. Top ($\top$) is the multiplicative one for the meet operator and the multiplicative zero for the join operator:

- $\forall x \in S,\ (\perp \vee x) = x$
- $\forall x \in S,\ (\top \vee x) = \top$
- $\forall x \in S,\ (\top \wedge x) = x$
- $\forall x \in S,\ (\perp \wedge x) = \perp$

If a lattice is well-defined for only one of these operators, it is called a semilattice. Most dataflow analyses require only the semilattice structure to work. The meet operator (or the join operator) in a semilattice defines a partial order. The partial order is reflexive, antisymmetric and transitive. An example of partial order is the natural order on $\mathbb{N}$, where $0 \leq 1 \leq 2 \leq 3 \leq 4$ etc.

2.2 SSA form

Data structure choices directly influence in the power and efficiency of program optimizations in our current optimizing compilers. Choosing poorly which data structures to use can inhibit program optimization to a point where advanced techniques can become undesirable. The static single assignment form (SSA) makes a useful class of optimizations more efficient and powerful. It was initially proposed in the 80s to, among other uses, verify the equality of variables in a program [Alpern et al., 1988].

Consider the program slice in figure 2.2. Could it be said that I and J are equivalent? Such answer will depend on which program point the execution is currently.

```
1: if Q then
2:     I ← 5
3:     J ← 5
4:     ...
5: else
6:     I ← 6
7:     J ← 7
8:     ...
9: end if
10: ...
```

Figure 2.2. Example where the SSA form can improve on the value numbering optimization.

In the end of the **if-then-else** the answer will depend, dynamically, on the value of Q, or no answer can be defined in a conservative static approach. However, in program point 4, these variables are equal even on the static view. To allow the detection of such equivalences without taking into account different points in the program, it's necessary to add several new variables for each variable in the original program. So, in the example, it is necessary to break I into three distinct variables. One in the **then** clause, one in the **else** clause and a last one in the end of the **if-then-else**. The same is done for J and now it's possible to verify, statically, that the I and J from inside the **then** clause are equivalent.

The process of transforming a program into the SSA form requires two steps [Cytron et al., 1989]. The first step consists in inserting ϕ-functions into certain points on the program. Such functions are special assignment states in the form $U \leftarrow \phi(V, W, ...)$, where $U, V, W, ...$ are variables and the number of operands $V, W, ...$ is the number of control flow predecessors of the program point where the ϕ-function is inserted. Such control flow predecessors are listed in order where the jth operand is associated with the jth predecessor. If the program execution reaches the ϕ-function from its jth predecessor, then U is assigned the value of the jth operand. Each execution of a ϕ statement uses only one operand and which operand to be used depends on the flow of execution. A ϕ-function can be trivially placed for each variable V int the form $V \leftarrow \phi(V, V, ...)$ at the entrance to any CFG node in the program without changing semantics.

The second step consists in giving new names for each variable V in the form V_i for various integers i. Each mention of V in the program is replaced by a mention of ones of these new names of V. Each new name V_i for V is the target of exactly one assignment statement in the program text. And along any control flow path, if you

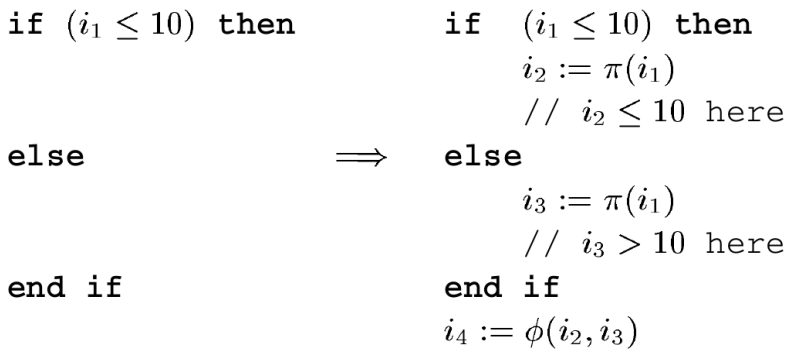


Figure 2.3. Example showing the Extended Static Single Assignment form transformation by inserting σ-functions.

consider any use of a new name V_i for V and the corresponding use of V in the original program, V and V_i will have the same value.

A program is in *minimalSSA* form if it is in SSA form and if the number of ϕ-functions inserted is as small as possible [Rosen et al., 1988]. Extra ϕ-functions (more than the *minimalSSA* form) might inhibit optimizations by hiding useful facts. They also add unnecessary overhead to the process of optimization. Thus it is very important to place ϕ-functions only where they are required to maintain the SSA form.

After a program has been transformed into the static single assignment form (SSA), it has two useful properties. First, each use of a variable is reached by exactly one assignment to that variable. Secondly, the programs contains ϕ-functions that distinguish values of variables obtained through distinct incoming control flow paths.

The Extended Static Single Assignment form is a cheaply computable extension of SSA [Bodik et al., 2000]. This extension differs from the *minimalSSA* form by introducing σ-functions that redefine the program's variables in points of interest, such as branches and switches. The main advantage of this σ-functions are inserted, for each variable appearing in the conditional expression, into the CFP out-edges of the conditional branch. Because of these assignments, each outcome of the conditional is associated with a distinct variable name, which allows for splitting the ranges of the conditional variables in range analysis, for example, with each of its σ-assignments taking distinct pieces of their range. The σ-functions are usually placed at the beginning of the basic blocks targeted by the branch.

2.3 LLVM

LLVM (Low-Level Virtual Machine) [Lattner and Adve, 2004] is a compiler framework that aims to make program analyses and transformations available for any piece of software in a transparent way for the programmers. It has two main characteristics are a code representation with several features that serve as common a representation for analyses and transformations and machine code translation, and a compiler design that exploits its representation to provide a multitude of capabilities.

The LLVM code representation uses a RISC-like instruction set allied to higher level information to describe programs in a useful way for analyses and transformations. This higher level information includes type information, explicit control flow graphs and an explicit data-flow representation using an infinite, typed register set in the SSA form. This code representation has several features, such as a low-level language independent type system that can be used to implement data-types and operations from higher-level languages, instructions for performing type conversions and address arithmetic, and two low-level exceptions-handling instructions for implementing exception semantics from languages who have it. The LLVM representation is independent from the source code language and it utilizes a low-level instruction set and memory model that are only slightly richer than common assembly languages. Its type system does not prevent representing code with very little type information and it does not impose any particular runtime requirement on programs.

The LLVM compiler framework exploits its code representation to provide a combination of five capabilities: persistent program information, offline code generation, user-based profiling and optimization, transparent runtime model and uniform, whole-program compilation. Over the last ten years, LLVM has somewhat altered this landscape and it is now used as a common infrastructure to implement a broad variety of statically and runtime compiled languages. We've used it to implement the analysis present in this work.

2.4 Points-to Analysis

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 [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].

Another analysis technique, originally investigated by Reynolds [Reynolds, 1968] for a Lisp-like language, is shape-analysis and it aims to give, for each program point a finite characterization of the possible "shapes" that the program's heap-allocated data structures can have at that point. Shapes characterize data structures. A shape descriptor could indicate whether the heap contains a singly linked list, potentially with a cycle, a doubly linked list, a binary tree, and so on. Shape Analysis is a static code analysis that discovers and verifies properties of linked, dynamically allocated data structures in computer programs. It is a form of pointer analysis, although it is more precise than typical pointer analysis. It has been applied to a variety of problems, such as memory safety and finding memory leaks, dereferences of dangling pointers, and discovering cases where a block of memory is freed more than once, finding array out-of-bounds errors, checking type-state properties, and verifying that a sort method is correct.

2.5 Range Analysis

Range Analysis is a compiler technique that aims to to find, statically, minimum and maximum values that each program variable might assume during the program execution. This analysis is important for enabling several compiler optimizations, such as dead and redundant code elimination. These are the removal of array bounds checks [Logozzo and Fahndrich, 2008] and overflow [Souza et al., 2011] checks for example. It can also be used for bitwidth aware register allocation [Barik et al., 2006], branch prediction [Patterson, 1995] and synthesis of hardware for specific applications [Cong et al., 2005].

The approach of simply having integer values as minimums and maximums may enable several compiler optimizations, but it is not effective do validate memory accesses or handle pointers. We adopt in this work a symbolic range analysis [Nazaré et al., 2014]. The analysis we use adopts symbolic ranges and a symbolic kernel, which is a set formed by either constants known statically or variables defined as input values, such as formal parameters of functions, reads and loads. An abstract interpretation framework generates invariants as symbolic ranges over the program's symbolic kernel, extracts a set of interval constraints from the program and performs a fix-point

algorithm until convergence.

The symbolic range analysis uses symbolic expressions for the lower and upper bounds of ranges. E is a symbolic expression, if and only if, E is defined by the grammar below. In this grammar s is a symbol and $n \in \mathbb{N}$.

$E ::= n \mid s \mid min(E, E) \mid max(E, E) \mid E - E \mid E + E \mid E/E \mid EmodE \mid E * E$

Even though there is a natural lattice between integer values coupled with both positive and negative infinities, there is no ordering between two distinct elements of the symbolic kernel of a program. The analysis provides a way to define such ordering with a valuation of symbolic expressions.

2.6 Pointer Disambiguation for Parallelization

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 [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.

We believe that the state-of-the-art approach in the field today is Rugina and Rinard [Rugina and Rinard, 2005]. Their paper presents a novel framework for the symbolic bounds analysis of pointers, array indexes, and accessed memory regions. Instead of using traditional fixed-point algorithms, it formulates each analysis problem as a system of inequality constraints between symbolic bound polynomials. It then reduces the constraint system to a linear program. The solution to the linear program provides symbolic lower and upper bounds for the values of pointer and array index variables and for the regions of memory that each statement and procedure accesses. Their technique was applied to divide and conquer programs that access disjoint regions of dynamically allocated arrays. Experimental results showed that their algorithm can verify the absence of data races in benchmark parallel programs, detect the available parallelism in benchmark serial programs, and verify that both sets of benchmark programs do not violate their array bounds.

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.

2.7 Points-to with Range Analyses

There exists 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 [Cousot and Cousot, 1977], whereas we use symbolic intervals.

The inspiration for much of this work springs from Balakrishnan and Reps notion of *Value Set* Analysis [Balakrishnan and Reps, 2004]. It is a static analysis for tracking the values of data objects. *Value Set* Analysis uses an abstract domain for representing the set of values that each data object can hold at each program point. The algorithm tracks memory addresses and integers values simultaneously. So it determines an over-approximation of the possible values that each variable of each type can hold at each program point. It tracks these values using a set of abstract data objects called abstract locations. The idea behind such abstraction is to exploit the fact that accesses on the variables of a program written in a high-level language appear as either static addresses (for globals) or static stack-frame offsets (for locals). The data object in the original source-code program that corresponds to a given abstract location can be one or more scalar, struct or array variables, but can also consist of just a segment of a scalar, struct or array variable.

Integer intervals have also being used by Yong *et al.* [Yong and Horwitz, 2004] and, more recently, by Oh *et al.* [Oh et al., 2014]. 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 [Bodik et al., 2000] 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.

Chapter 3

A New Points-to Analysis

The goal of this chapter is twofold. First, we want to illustrate our novel points-to analysis with an example. Section 3.1 presents this example, and introduces the reader to a few insights that came out of this work. After that, we discuss the technical details of our contribution. To this end, Section 3.2 introduces the SymbRanges lattice, and explains how to build this structure. This lattice will gives us, later, a way to answer the question: "do pointers p_1 and p_2 alias each other?"

3.1 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 3.1 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.

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

```
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 3.1. 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.

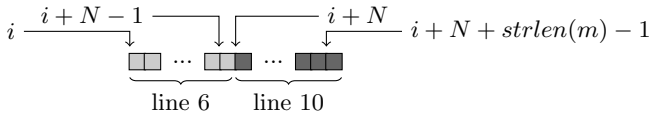


Figure 3.2. Array p in the routine `prepare` seen in Fig.3.1. Lines 6 and 10 represent the different stores in the figure.

of the lowest (l) and highest (u) values that i can assume throughout the execution of the program. In this work, we propose an alias analysis that solves this problem. To achieve this goal, we couple this alias analysis with range analysis on symbolic intervals [Blume and Eigenmann, 1994]. 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 $\mathtt{p} + N + \mathtt{strlen(m)} - \mathtt{1}$. 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 3.2, 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 3.1, and we represent them abstractly as loc_{17} and loc_{18}. These names

```
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 3.3. 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 3.4. Program from Figure 3.3, after pointer is renamed within loop.

are unique across the entire program. After running our analysis, we find out that the abstract state (GR) of `i` at line 6 is $\mathrm{GR}(\mathtt{i}_{\ell n.6}) = \{\mathsf{loc}_{17} + [0, \mathtt{N} - 1]$, and that the abstract state of `i` at line 10 is $\mathrm{GR}(\mathtt{i}_{\ell n.10}) = \{\mathsf{loc}_{17} + [\mathtt{N}, \mathtt{N} + \mathtt{strlen}(m) - 1]\}$. 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 3.3 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 $\mathrm{GR}(tmp_0) = \{\mathsf{loc}_0 + [0, N + 1]\}$ and that $\mathrm{GR}(tmp_1) = \{\mathsf{loc}_0 + [1, N + 2]\}$, 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 3.1 does not work in Figure 3.3. 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 $\mathrm{LR}(new_p) = \mathsf{loc}_{new} + [0, 0]$, regardless of

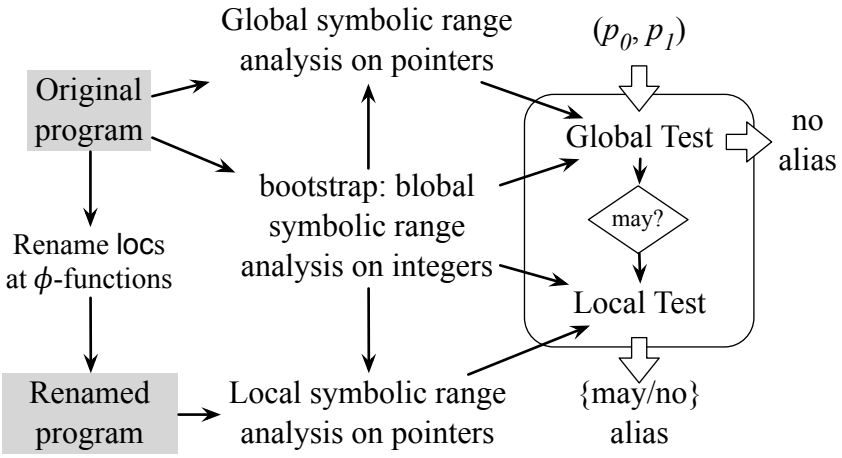


Figure 3.5. Overview of our pointer analysis.

the old ranges assigned to the original pointer p. In Figure 3.4 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.

3.2 Combining Range and Pointer Analyses

We perform our pointer analysis in several steps. Figure 3.5 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 [Blume and Eigenmann, 1994], 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.

3.2.1 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 3.1; henceforth, we shall call this syntax our *core language*. We shall be working on programs in Extended Static Single Assignment (e-SSA) form [Bodik et al., 2000]. E-SSA form is a flavor of Static Single Assignment (SSA) [Cytron et al., 1991] form,

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	|	$\text{jump}(\ell)$

Table 3.1. The syntax of our language of pointers.

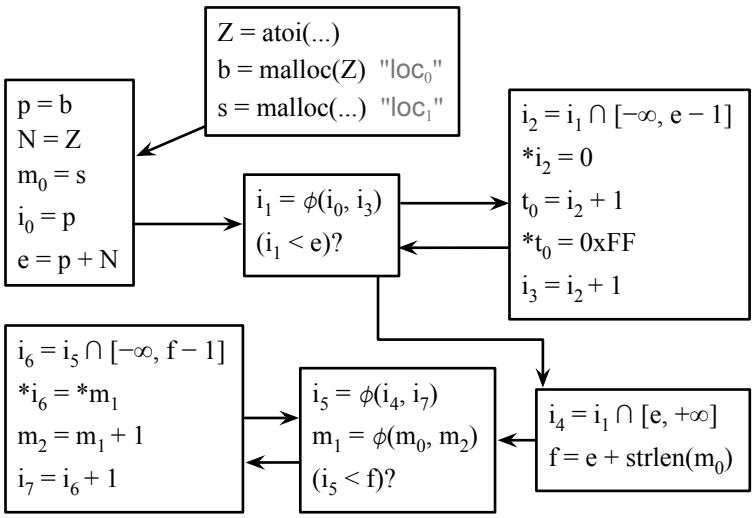


Figure 3.6. Interprocedural control flow graph of program seen in Figure 3.1

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 3.6 shows the interprocedural control flow graph of the program seen in Figure 3.1. It is equivalent to a normal control flow graph with its functions inlined. The implementation of the analysis that we shall present in this work is interprocedural, albeit not context-sensitive. To achieve interprocedurality, we associate actual parameters with formal parameters of functions. In the example of Figure 3.1, pointer b – an actual parameter – is linked with p – a formal parameter – through a ϕ-function.

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.* [Choi et al., 1991], 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 3.2.4, our abstract interpreter associates with p_0 a new abstract state which indicates that p_0 is not a valid reference to any location.

3.2.2 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 3.1, new locations are created by *malloc* operations.

Example 1 *Figure 3.6 shows the interprocedural control flow graph of the program seen in Figure 3.1. The two allocations at lines 17 and 18 are associated respectively with* loc_0 *and* loc_1*.*

3.2.3 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.* [Nazaré et al., 2014]. 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.

$$\begin{aligned} E \;::=\; & n \mid s \mid \min(E,E) \mid \max(E,E) \mid E-E \\ & \mid E+E \mid E/E \mid E \bmod E \mid E \times E \end{aligned}$$

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:

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

From the previous definitions, 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 work, 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 [Nazaré et al., 2014], if applied onto the program seen in Figure 3.3, will gives us that* $R(i_{\langle line\ 3\rangle}) = [0,0], R(i_{\ell n.5}) = [0, N-1]$, $R(i_{\ell n.7}) = [0, N+1]$.

3.2.4 Global Range Analysis of Pointers

As we have mentioned in Section 3.1 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 3.2.6. 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 $\mathrm{GR}(p) = (p_0, \ldots, p_{n-1})$ 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,* $S = \{s \mid \exists i \in \mathbb{N}, d \in \mathbb{Z}, s = @(\mathit{loc}_i) + d\}$ *is abstracted by* $\alpha(S) = (p_0, p_1 \ldots, p_{n-1})$ *where :*

- $p_i = \bot$ *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 $\mathrm{GR}(p)$ 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 $\mathrm{GR}(p) = (\bot, [l_1, u_1], \bot, [l_3, u_3], \bot)$, by the set $\mathrm{GR}(p) = \{\mathsf{loc}_1 + [l_1, u_1], \mathsf{loc}_3 + [l_3, u_3]\}$. 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 $u_3 = 8$. $\mathrm{GR}(p) = \{\mathsf{loc}_1 + [3, 5], \mathsf{loc}_3 + [3, 8]\}$ is then depicted in Figure 3.7.

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 $\mathrm{GR}(p^1) = (p_0^1, \ldots p_{n-1}^1)$ and $\mathrm{GR}(p^2) = (p_0^2, \ldots p_{n-1}^2)$, the union $\mathrm{GR}(p^1) \sqcup \mathrm{GR}(p^2)$ 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 $\mathrm{GR}(p^1) \sqsubseteq \mathrm{GR}(p^2)$ 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 3.6 where we only have two malloc sites denoted by* loc_0 *and* loc_1*, we will obtain the following results:* $GR(p) = GR(b) = \{\mathsf{loc}_0 + [0, 0]\}$, $GR(m_0) = GR(s) = \{\mathsf{loc}_1 + [0, 0]\}$, $GR(e) = \mathsf{loc}_0 + [N, N]$, $GR(m_1) = \mathsf{loc}_1 + [1, +\infty]$, $GR(i_7) = \mathsf{loc}_0 + [N + strlen(m_0), N + strlen(m_0) + 1]$. *How this mapping is found is discussed in the rest of this section.*

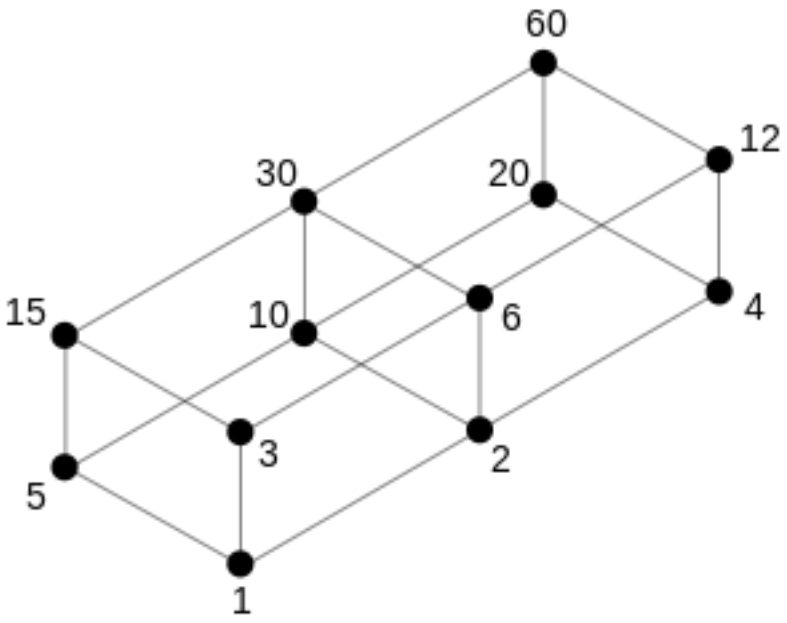


Figure 3.7. The concrete semantics of $\mathrm{GR}(p) = \{\mathsf{loc}_1 + [3,5], \mathsf{loc}_3 + [3,8]\}$. Dark grey cells denote possible (concrete) values of p.

Abstract semantics for GR, and concretisation. The abstract semantics of each instruction in our core language is given by Figure 3.8. Figure 3.8 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 3.8, this notation surfaces in the semantics of intersections. The abstract interpretation of the pointer-related instructions in Figure 3.6 yields the results discussed in Example 3.

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:*

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

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].

[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 $S(p_{i\uparrow})$.

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

Figure 3.8. Constraint generation for GR with $\mathrm{GR}(p) = (p_0, \ldots, p_{n-1})$ given p in the right hand side of rules

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

Proof 1 *Immediate since $(\alpha_{\mathsf{SymbRanges}}, \gamma_{\mathsf{SymbRanges}})$ 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 3.2.9.

The abstract interpretation of loads and stores. In Figure 3.8, 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 Chapter 4, a typical compilation infrastructure 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.

3.2.5 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:

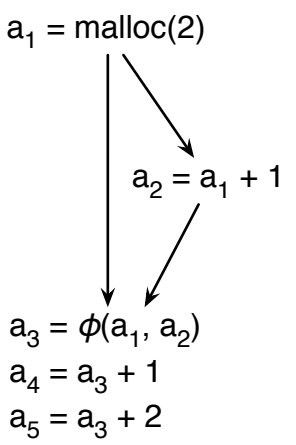
Proposition 2 (Correctness) *Let* p *and* p' *be two pointers in a given program then :*

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

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 $Q_{\mathrm{GR}}(p,p')$ as :

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



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

3.2.6 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 3.9 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 3.2.5 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 3.10 gives the abstract semantics of each instruction. The abstract value LR(p) exists in $(\mathcal{L}oc \cup \mathcal{N}ewLocs) \times \mathsf{SymbRanges}$ 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 $\mathsf{loc} + R$ instead of (loc, R). Similarly to γ_{GR}, γ_{LR} denotes the set of abstract addresses from $@(\mathsf{loc}) + R_{\downarrow}$ to $@(\mathsf{loc}) + R_{\uparrow}$.

To find a solution to the local analysis, we solve the system provided by the abstract rules seen in Figure 3.10. 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 3.9 (Right) shows the result of the local analysis. Contrary

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

Figure 3.10. Constraint generation for LR

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.

3.2.7 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 $LR(p) = \mathsf{loc} + R$ and $LR(p') = \mathsf{loc}' + R'$, then if $\mathsf{loc} = \mathsf{loc}'$ and $R \sqcap R' = \emptyset$ then $\gamma(LR(p)) \cap \gamma(LR(p') = \emptyset$. In other words, p and p' never alias.*

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

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

3.2.8 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.* [Tavares et al., 2014], this property – single information – is sufficient to enable sparse implementation of non-relational static analyses [Tavares et al., 2014]. 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.

3.2.9 A wrap-up Example

Example 4 shows how our analysis works on the program seen in Figure 3.1.

Example 4 *Figure 3.6 shows the interprocedural control flow graph (CFG) of the program in Figure 3.1. Our graph is in e-SSA form [Bodik et al., 2000]. Figure 3.11 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 3.1 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 3.1, 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 $\mathtt{i}_7$. 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

	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 3.11. Abstract interpretation of interprocedural CFG seen in Figure 3.6 (program in Figure 3.1). 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).

Abstract Interpreter depicted in Figure 3.8, 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.

Chapter 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[1]:

- 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 work, namely the ability to distinguish fields and indices within aggregate types. In this case, such disambiguation is only possible when

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

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

Table 4.1. 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 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 $\texttt{i} = \texttt{B} + \mathit{iter} \times S, \texttt{i} \leq \texttt{N}$. 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 4.1 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 [Ryder et al., 2001], PtrDist [Zhao et al., 2005] and MallocBench [Grunwald et al., 1993]. 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 4.2 shows the proportion of queries that we have been able to disambiguate with the global test of Section 3.2.4. The two columns **noalias** of Figure 4.2 correspond to the percentage in column **%rbaa** applied on the column **#Queries** of Figure 4.1. 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 during the

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

Table 4.2. Number of queries solved with the global test of Section 3.2.4. 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.

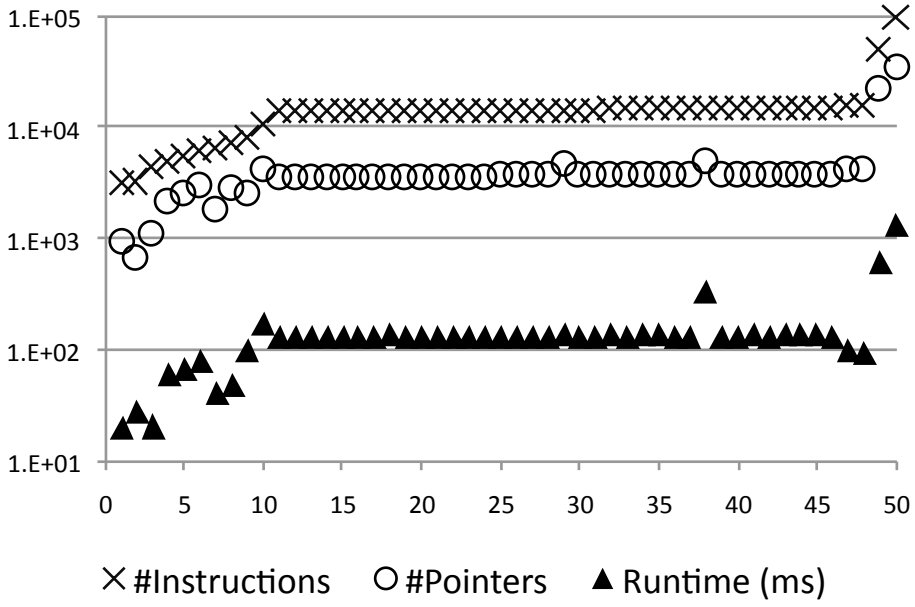


Figure 4.1. Runtime of our analysis for the 50 largest benchmarks in the LLVM test suite. Each point on the X-axis represents a different benchmark. Benchmarks are ordered by size. This experiment took less than 10 seconds.

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 4.1 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 3.2.3, 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 4.1 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 4.1 is 0.982, and the correlation between time and number of pointers is 0.975.

Chapter 5

Final Thoughts

5.1 Limitations

Using the symbolic range analysis for comparing offsets is inherently imprecise. Issues lie on comparing two symbolic expressions and its difficulties, especially on comparing lower bounds with upper bounds from two different variables, and on local relationships of variables with overlapping ranges.

Figure 5.1 shows the first issue. Analyzing this algorithm, the symbolic range analysis returns the following ranges: $R(\sigma_a) = [a, b-1]$, $R(\sigma_b) = [a+1, b]$. This hinders the disambiguation of the two array accesses, $V[\sigma_a]$ and $V[\sigma_b]$, since it is impossible to prove that the ranges of σ_a and σ_b do not overlap. To do so, it would be necessary for the valuation of symbolic expressions to be able to say that $b-1 < a+1$ or that $b < a$, which it cannot do. So in this example, even though it is obvious that the two array accesses cannot alias to the same location since their offsets are obligatorily different by the **if** condition, we cannot disambiguate them.

Figure 5.2 shows the second issue. It again shows two array accesses that are obviously disjoint that our analysis cannot disambiguate. Analyzing this algorithm, the symbolic range analysis returns the following ranges: $R(i) = [0, 9]$, $R(j) = [1, 10]$. It's clear that these ranges overlap and that our analysis could not say that they are disjoint even though, at the array accesses, j is aways different and greater than i.

These examples show two very interesting limitations of our analysis. They both can disable optimizations such as automatic parallelization of code and loop invariant code motion on some cases in which these optimizations might be desirable. It is clear that simply verifying if two offset ranges are disjoint is not enough to achieve ideal precision, since they can hide relationships between variables that are aways true when the two variables are alive and that can help in disambiguating two pointers, even if

```
1: ...
2: if a < b then
3:     σ_a = σ(a)
4:     σ_b = σ(b)
5:     V[σ_a] ← •
6:     V[σ_b] ← •
7:     ...
8: end if
9: ...
```

Figure 5.1. Example of our first described limitation. Even though we cannot prove that the symbolic ranges of σ_a and σ_b do not overlap, it is obvious that σ_a is aways different from σ_b because of the **if** condition.

```
1: ...
2: i = 0
3: while i < 10 do
4:     j = i + 1
5:     V[i] ← •
6:     V[j] ← •
7:     ...
8:     i++
9: end while
10: ...
```

Figure 5.2. Example of our second described limitation. Even though we cannot prove that the ranges of i and j do not overlap, it is obvious that j is aways different from i because it is aways greater.

there is a relationship between these pointers or between the offsets that constitute such pointers.

5.2 Future Work

The reason for the limitations exposed before lays on the fact that our analysis use range intervals to disambiguate pointers. In the examples of figure 5.1 and figure 5.2, the ranges of integer variables might either overlap or aren't comparable. But, in the examples, variables do have relationships between them that allow for disambiguation in the form of inequalities ($a < b$ and $i < j$). To take advantage of such relationships we intend to use a lattice similar to Pentagons.

Pentagons is an abstract domain invented by Logozzo and Fähndrich to infer symbolic bounds to the integer variables used in programs [Logozzo and Fahndrich,

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 ranges $[l, u]$ of numeric lower (l) and upper (u) bounds. The second lattice is the *strict upper bound*, which maps each variable v to a set $L_<$ of other variables, so that if $u \in L_<(v)$, at a given program point `p`, then $u < v$ at `p`.

Since their debut [Logozzo and Fahndrich, 2008], 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 [Logozzo and Fähndrich, 2010], 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 [Logozzo and Fahndrich, 2008].

Our initial and recent use of pentagons for such purpose has been successful. It is able to handle programs as large as SPEC's gcc in a few minutes and go through SPEC CPU 2006 [Henning, 2006] in able time. It has proven very useful in some cases, such as SPEC `470.lbm`. In future work, we plan to investigate better splitting strategies and other more expressive lattices to improve the global precision of our analyses.

5.3 Final Conclusions

In this work 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.

Apart from this contribution, there is plenty to study on the area of pointer analysis, and the area of pointer arithmetics still needs quite a bit of research. Their dire needs are very efficient static analyses that run fast on very big programs, and very lean dynamic analyses. Our focus has been on the static analysis side. Lazy

implementations, where main computations are made on the query moment, seem to be a very promising take on alias analyses and can expedite runtime. Focus on integrating new proposals to bigger compilation frameworks and existing optimizations should also be explored by the community on an effort of making new technology more usable across researchers and projects. There is still a lot of work to be done and we hope that our contribution can be only a building block of a much bigger effort from many more scientists.

Bibliography

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

Alpern, B., Wegman, M. N., and Zadeck, F. K. (1988). Detecting equality of variables in programs. In *In POPL*, pages 1--11. ACM.

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

Balakrishnan, G. and Reps, T. (2004). Analyzing memory accesses in x86 executables. In *In CC*, pages 5--23. Springer.

Barik, R., Grothoff, C., Gupta, R., Pandit, V., and Udupa, R. (2006). Optimal bitwise register allocation using integer linear programming. In *In LCPC*, volume 4382 of *Lecture Notes in Computer Science*, pages 267–282. Springer.

Blume, W. and Eigenmann, R. (1994). Symbolic range propagation. In *In IPPS*, pages 357--363.

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

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

Cong, J., Fan, Y., Han, G., Lin, Y., Xu, J., Zhang, Z., and Cheng, X. (18-21 Jan. 2005). Bitwidth-aware scheduling and binding in high-level synthesis. *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, 2:856–861.

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

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

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

Grunwald, D., Zorn, B., and Henderson, R. (1993). Improving the cache locality of memory allocation. In *In PLDI*, pages 177--186. ACM.

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

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

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

Hind, M. (2001). Pointer analysis: Haven't we solved this problem yet? In *In PASTE*, pages 54--61. ACM.

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

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

Logozzo, F. and Fahndrich, M. (2008). Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *In SAC*, pages 184--188. ACM.

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

Nation, J. B. (2016). Notes on lattice theory. University of Hawaii.

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

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

Paisante, V., Maleej, M., Gonnord, L., Barbosa, L., and Pereira, F. M. Q. (2016). Symbolic range analysis of pointer. In *CGO*. ACM.

Paisante, V. M., Rodrigues, R. E., Saggioro, L. F. Z., e Oliveria, L. B., and Pereira, F. M. Q. (2014). Prevencao de ataques em sistemas distribuidos via analise de intervalos. In *Anais do SBSeg*, page 209.

Patterson, J. R. C. (1995). Accurate static branch prediction by value range propagation. In *In PLDI*, pages 67--78. ACM.

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

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

Reynolds, J. C. (1968). Automatic computation of data set definitions. In *Proceedings of the IFIP Congress*. North-Holland.

Rosen, B. K., Zadeck, F. K., and Wegman, M. N. (1988). Global value numbers and redundant computations. In *In POPL*, pages 12–27. ACM Press.

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

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

Saggioro, L. F. Z., Paisante, V. M., Rodrigues, R. E., e Oliveria, L. B., and Pereira, F. M. Q. (2015). Cruzando dados distribuÃdos para detectar estouro de inteiro. volume 13. IEEE.

Sagiv, M., Reps, T., and Wilhelm, R. (1998). Solving shape-analysis problems in languages with destructive updating. *TOPLAS*, 20(1):1--50.

Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. (2012). Addresssanitizer: a fast address sanity checker. In *In USENIX ATC*, pages 28--28. USENIX Association.

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

Souza, M. R. S., Guillon, C., Pereira, F. M. Q., and da Silva Bigonha, M. A. (2011). Dynamic elimination of overflow tests in a trace compiler. In *In CC*, pages 2--21.

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

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

Wolfe, M. (1996). *High Performance Compilers for Parallel Computing*. Adison-Wesley, 1st edition.

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

Yong, S. H. and Horwitz, S. (2004). Pointer-range analysis. In *In SAS*, pages 133--148. Springer.

Zhang, Q., Xiao, X., Zhang, C., Yuan, H., and Su, Z. (2014). Efficient subcubic alias analysis for C. In *In OOPSLA*, pages 829--845. ACM.

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