Pointer Disambiguation via Strict Inequalities
Maroua Maalej, Vitor Paisante, Ramos Pedro, Laure Gonnord, Fernando Pereira

To cite this version:
Abstract

The design and implementation of static analyses that disambiguate pointers has been a focus of research since the early days of compiler construction. One of the challenges that arise in this context is the analysis of languages that support pointer arithmetics, such as C, C++ and assembly dialects. This paper contributes to solve this challenge. We start from an obvious, yet unexplored, observation: if a pointer is strictly less than another, they cannot alias. Motivated by this remark, we use abstract interpretation to build strict less-than relations between pointers. We construct a program representation that bestows the Static Single Information (SSI) property onto our dataflow analysis. SSI gives us a sparse algorithm, whose correctness is easy to ensure. We have implemented our static analysis in LLVM. It runs in time linear on the number of program variables, and, depending on the benchmark, it can be as much as six times more precise than the pointer disambiguation techniques already in place in that compiler.

Keywords Alias analysis, range analysis, speed, precision

1. Introduction

Pointer disambiguation consists in determining if two pointers, \( p_1 \) and \( p_2 \), can refer to the same memory location. If overlapping happens, then \( p_1 \) and \( p_2 \) are said to alias. Pointer disambiguation has been a focus of much research [3, 15, 17, 34], since the debut of the first alias analysis approaches [6, 53]. Today, state-of-the-art algorithms have acceptable speed [15, 28], good precision [40] or meet each other halfway [16, 36]. However, understanding the relationships between memory references in programming languages that support pointer arithmetics remains challenging.

Pointers arithmetic come from the ability to associate pointers with offsets. Much of the work on automatic parallelization consists in the design of techniques to distinguish offsets from the same base pointer. Michael Wolfe [42, Ch.7] and Aho et al. [1, Ch.11] have entire chapters devoted to this issue. State-of-the-art approaches perform such distinction by solving diophantine equations, be it via the greatest common divisor test, be it via integer linear programming, as Rugina and Rinard do [31]. Other techniques try to associate intervals, numeric or symbolic, with pointers [4, 26, 27, 44], presenting different ways to build Balakrishnan and Reps' notion of value sets. And yet, as expressive and powerful as such approaches are, they fail to disambiguate locations that are obviously different, as \( v[i] \) and \( v[j] \), in the loop:

\[
\text{for}(i = 0, j = N; i < j; i++, j--) \ v[i] = v[j];
\]

In this paper, we present a simple and efficient solution to this shortcoming. We say that \( v[i] \) and \( v[j] \) are obviously different locations because \( i < j \). There are techniques to compute \textit{less-than} relations between integer variables in programs [7, 20, 21, 23]. Nevertheless, so far, they have not been used to disambiguate pointer locations. The insight that such techniques are effective and useful to such purpose is the key contribution of this paper. However, we go beyond: we rely on recent advances on the construction of sparse dataflow analyses [58] to design an efficient way to solve less-than inequalities. The sparse implementation lets us view this problem as an instance of the abstract interpretation framework; hence, we get correctness for free. The end result of our tool is a less-than analysis that can be augmented to handle different program representations, and that can increase in non-trivial ways the ability of compilers to distinguish pointers.

To demonstrate this last statement, we have implemented our static analysis in the LLVM compiler [19]. We show...
2. Overview

To motivate the need for a new points-to analysis we show its application on the programs seen in Figure 1. The figure displays the C implementation of two sorting routines that make heavy use of pointers. In both cases, we know that memory positions v[i] and v[j] can never alias within the same loop iteration. However, traditional points-to analyses cannot prove this fact. Typical implementations of these analyses, built on top of the work of Andersen [3] or Steensgaard [34], can distinguish pointers that dereference different memory blocks; however, they do not say much about references ranging on the same array.

There are points-to analyses designed specifically to deal with pointer arithmetics [2] [4] [24] [27] [31] [32] [37] [39] [41]. Still, none of them works satisfactorily for the two examples seen in Figure 1. The reason for this ineffectiveness lays on the fact that these analyses use range intervals to disambiguate pointers. In our examples, the ranges of integer variables i and j overlap. Consequently, any conservative range analysis, à la Cousot [10], once applied on Figure 1(a), will conclude that i exists on the interval [0, N − 1], and that j exists on the interval [1, N − 1]. Because these two intervals have non-empty intersection, points-to analyses based on the interval lattice will not be able to disambiguate the memory accesses at lines 6-8 of Figure 1(a). The same holds true for the memory accesses in Figure 1(b).

The technique that we introduce in this paper can disambiguate every use of v[i] and v[j] in both examples. The key to this success is the observation that i < j at every program point where we have an access to v. We conclude that i < j by means of a “less-than check”. A less-than check is a relationship between two variables that is true whenever we can prove – statically – that one holds a value lesser than the value stored in the other. In Figure 1(a), we know that i < j because of the way that j is initialized, within the for statement at line 4. In Figure 1(b), we know that i < j due to the conditional check at line 7.

A more precise alias analysis brings many advantages to compilers. One of such benefits is optimizations: the extra precision gives compilers information to carry out more extensive transformations in programs. Notice that the pointer analysis that we propose in this paper provides weaker guarantees than more traditional approaches. Continuing with our example, the pointers v[i] and v[j] in Figure 1(a & b) do, indeed, alias across the entire loop nest, albeit not at the same time. As we shall discuss in Section 3.5 if we say that two pointers do not alias, then they will never alias at any program point where they are simultaneously alive. This property is strong enough to support most of the classic compiler optimizations: constant propagation, value numbering, subexpression elimination, scheduling, etc. However, our technique cannot be used with optimizations that modify the iteration space of programs, such as loop fission and fusion.

3. The Less-Than Check

This section introduces a dataflow analysis whose goal is to construct a “less-than” set for each variable x (pointer or numeric, as we will discuss in Section 3.6). We denote such an object by LT(x). As we prove in Section 3.5 the important invariant that this static analysis guarantees is that if x' ∈ LT(x), then x' < x at every program point where both variables are alive. Our ultimate goal is to use this invariant to disambiguate pointers, as we explain in Section 3.6.

3.1 The Core Language

We use a core language to formalize the developments that we present in this paper. Figure 2 shows the syntax of this language. Our core language contains only those instructions that are essential to describe our static analysis.

Figure 1. Two snippets of C code that challenge typical pointer disambiguation approaches.
The syntax of our language. Variables have scalar type, e.g., either integer or pointer.

Figure 3. Program written in our core language.

The Support of Range Analysis on Integer Intervals. The SSA representation ensures that a new name is created at each program point where a variable is defined. To meet the other two requirements, we split live ranges at subtractions and after conditionals. Going back to Figure 2, we see that our core language contains only syntax for arithmetic addition and after conditionals. Going back to Figure 2, we see that our core language contains only syntax for arithmetic addition and subtraction if \( x \) is used in a conditional, e.g., \( x = x + n, n < 0 \). In this case, we know that \( x < x' \).

3.2 Program Representation

We want to implement a sparse dataflow analysis. Sparsity is good for: (i) time and space, as it reduces from cubic to quadratic (on the number of variables) the amount of information that needs to be stored; and (ii) correctness, as it simplifies all the proofs of theorems. A dataflow analysis is said to be sparse if it runs on a program representation that ensures the Static Single Information (SSI) Property. To keep this paper self-contained, we quote Tavares et al.’s notion of single information property:

**Definition 3.2** (Static Single Information Property). A dataflow analysis bears the static single information property if it always associates a variable with the same abstract state at every program point where that variable is alive.

Following Tavares et al. [38], to ensure the SSI property, we split the live range of every variable \( x \) at each program point where new information about \( x \) can appear. The live range of a variable \( v \) is the collection of program points where \( v \) is alive. To split the live range of \( x \) at a point \( \ell \), we create a copy \( x' = x \) at \( \ell \), and rename uses of \( x \) at every program point dominated by \( \ell \). We shall write \( \ell \) dom \( \ell' \) to indicate that \( \ell \) dominates \( \ell' \), meaning that any path from the beginning of the control flow graph to \( \ell' \) must cross \( \ell \). There are three situations that create new less-than information about a variable \( x' \):

1. \( x \) is defined. For instance, if \( x = x' + 1 \), then we know that \( x' < x \);
2. \( x \) is used in a subtraction, e.g., \( x' = x + n, n < 0 \). In this case, we know that \( x' < x \);
3. \( x \) is used in a conditional, e.g., \( x < x' \). In this case, we know that \( x < x' \) at the true branch, and \( x' \leq x \) at the false branch.

The reader can augment it with other assembly instructions, to make it as expressive as any industrial-strength program representation. As a testimony to this fact, the implementation that we describe in Section 4 comprises the entire LLVM intermediate representation. Figure 2 describes programs in Static Single Assignment form [12]: therefore, it contains \( \phi \)-functions. Additionally, it contains arithmetic instructions and conditional branches. These two kinds of instructions feed our static analysis with new information.

**Example 3.1**. Figure 3 describes a program in our core language. This is an artificial example, whose semantics is immaterial. Figure 2 illustrates a few key properties of the strict SSA representation: (i) the definition point of a variable dominates all its uses, and (ii) if two variables interfere, one of them is alive at the definition point of the other. Such properties will be useful in Section 3.3.

**Figure 4**. The syntax of our intermediate language.
Our live range splitting strategy leads to the creation of a different program representation. Figure 4 shows the two syntactic transformations that convert a program written in the syntax of Figure 2 into a program written in the syntax of Figure 3. We let $x_0 = x_1 - n \parallel \langle x_2 = x_1 \rangle$ denote a composition of two statements, $x_0 = x_1 - n$ and $x_2 = x_1$. The second instruction splits the live range of $x_1$. Both statements happen in parallel. Thus, $x_0 = x_1 - n \parallel \langle x_2 = x_1 \rangle$ does not represent an actual assembly instruction; it is only used for notational convenience. Similarly, when transforming conditional tests, we let $\langle x_{1t} = x_1, x_{2t} = x_2 \rangle$ denote two copies that happen in parallel: $x_{1t} = x_1$, and $x_{2t} = x_2$. Whenever there is no risk of ambiguity, we write simply $\langle x_{1t}, x_{2t} \rangle$, as in Figure 2. Parallel copies and $\phi$-functions are removed before code generation, after the analyses that require them have already run. This step is typically called SSA-Elimination phase.

**Example 3.3.** Figure 6 shows the result of applying the rules seen in Figure 5 onto the program in Figure 3.

### 3.3 Constraint Generation

Once we have a suitable program representation, we use the rules in Figure 7 to generate constraints. These constraints determine the less-than set of variables. Constraint generation is $O(|\mathcal{V}|)$, where $\mathcal{V}$ is the set of variables in the target program. We have four kinds of constraints:

- **init:** Set the less-than set of a variable to empty: $\text{LT}(x) = \emptyset$.
- **union:** Set the less-than set of a variable to be the union of another less-than set and a single element: $\text{LT}(x_3) = \{x_1\} \cup \text{LT}(x_2)$.
- **inter:** Set the less-than set of a variable to be the intersection of multiple less-than sets: $\text{LT}(x) = \text{LT}(x_1) \cap \ldots \cap \text{LT}(x_n)$.
- **copy:** Sets the less-than set of a variable to be the less-than set of another variable: $\text{LT}(x) = \text{LT}(x')$.

**Example 3.4.** The rules in Figure 7 produce the following constraints for the program in Figure 5:

- $\text{LT}(x_0) = \emptyset$.
- $\text{LT}(x_1) = \{x_0\} \cup \text{LT}(x_0)$.
- $\text{LT}(x_2) = \text{LT}(x_1) \cap \text{LT}(x_3)$.
- $\text{LT}(x_3) = \{x_2\} \cup \text{LT}(x_2)$.
- $\text{LT}(x_4) = \emptyset$.
- $\text{LT}(x_5) = \{x_4\} \cup \text{LT}(x_2)$.
- $\text{LT}(x_{1f}) = \{x_{4f}\} \cup \text{LT}(x_{4f}) \cup \text{LT}(x_1)$.
- $\text{LT}(x_{1f}) = \{x_{4f}\} \cup \text{LT}(x_{4f}) \cup \text{LT}(x_1)$.
- $\text{LT}(x_{1f}) = \text{LT}(x_{1f}) \cap \text{LT}(x_1)$.
- $\text{LT}(x_{4f}) = \text{LT}(x_{4f}) \cap \text{LT}(x_1)$.
3.4 Constraint Solving

Constraints are solved via a worklist algorithm. We initialize \( LT(x) \) to \( V \), for every variable \( x \). During the resolution process, elements are removed from each LT, until a fixed point is achieved. Theorem 3.7 in Section 3.5 guarantees that this process terminates. Constraint solving is equivalent to finding transitive closures; thus, it is \( O(|V|^2) \). In practice, we have observed an \( O(|V|) \) behavior, as we show in Section 3.5.

Example 3.5. To solve the constraints in Example 3.4, we initialize every LT set to \( \{x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_{1f}, x_{1t}, x_{4f}, x_{4t}\} \), i.e., the set of program variables. The following sets are a fixed-point solution of this system: \( LT(x_0) = LT(x_4) = LT(x_{4t}) = \emptyset \); \( LT(x_1) = LT(x_2) = LT(x_{1f}) = LT(x_{1t}) = \{x_0\} \); \( LT(x_3) = \{x_0, x_2\} \); \( LT(x_5) = \{x_0, x_{4t}\} \); and \( LT(x_{1t}) = \{x_0, x_{4t}\} \).

3.5 Properties

There are a number of properties that we can prove about our dataflow analysis. In this section we focus on two core properties: termination and adequacy. Termination ensures that the constraint solving approach of Section 3.4 always reaches a fixed point. Adequacy ensures that our analysis conforms to the semantics of programs. To show termination, we start by proving Lemma 3.6.

Lemma 3.6 (Decreasing). If constraint resolution starts with \( LT(x) = V \) for every \( x \), then \( LT(x) \) is monotonically decreasing or stationary.

Proof: The proof follows from a case analysis on each constraint produced in Figure 4, plus induction on the number of elements in LT. Figure 4 reveals that we have only three kinds of constraints:

- \( LT(x) = \emptyset \): in this case, LT is stationary;
- \( LT(x) = \{x\} \cup LT(x') \): by induction, LT is decreasing or stationary, and LT(x) always contains \( \{x\} \);
- \( LT(x) : LT(x_1) \cap \ldots \cap LT(x_n) \): we apply induction on each LT(x_i), \( 1 \leq i \leq n \).

To prove Theorem 3.7, which states termination, we need to recall \( P^V \), the semi-lattice that underlines our less-than analysis, \( P^V = \{V, \cap, \bot = \emptyset, \top = V, \subseteq\} \) is the lattice formed by the partially ordered set of program variables. Ordering is given by subset inclusion \( \subseteq \). The meet operator (greatest lower bound) is set intersection \( \cap \). The lowest element in this lattice is the empty set, and the highest is \( V \).

Theorem 3.7 (Termination). The constraint resolution process terminates.

Proof: The proof of this theorem is the conjunction of two facts: (i) Constraint sets are monotonically decreasing; and (ii) they range on a finite lattice. Fact (i) follows from Lemma 3.6. Fact (ii) follows from the definition of \( P^V \).

We followed the framework of Tavares et al. 38 to build our intermediate program representation. Thus, we get correctness for free, as we are splitting live ranges at every program point where new information can appear. Lemma 3.8 formalizes this notion.

Lemma 3.8 (Sparsity). \( LT(x) \) is invariant along the live range of \( x \).

Proof: The proof follows from case analysis on the constraint generation rules in Figure 7. By matching constraints with the syntax that produce them, we find that the abstract state of a variable can only change at its definition point. This property is ensured by the live range splitting strategy that produces the program representation that we use. ☐

We close this section showing adequacy. In a nutshell, we want to show that if our constraint system determines that a variable \( x \) belongs into the less-than set of another variable \( x' \), then we know that \( x < x' \). This is a static notion: we consider the values of \( x \) and \( x' \) that exist at the same moment during the execution of a program. Theorem 3.9 formalizes this observation. The proof of that theorem requires the semantics of the transformed language, whose syntax appears in Figure 4. However, for the sake of space, we omit this semantics. We assume the existence of a small-step transition rule \( \rightarrow \), which receives an instruction \( \iota \), plus a store environment \( \Sigma \), and produces a new environment \( \Sigma' \). The store maps variables to integer values. Analogously, the constraints of Figure 7 show how instructions change the abstract state LT. We write \( \nu \vdash LT \triangleright LT' \) to denote an abstract transition. We let \( \models \) denote the following relation: if \( x' \in LT(x) \), then \( \Sigma(x') < \Sigma(x) \). If this relation is true for every element in the domain of \( \Sigma \), then we write \( LT \models \Sigma \).

Theorem 3.9 (Adequacy). If \( LT_1 \models \Sigma_1 \wedge \iota \vdash \Sigma_1 \rightarrow \Sigma_2 \wedge \iota \vdash LT_1 \triangleright LT_2 \), then \( LT_2 \models \Sigma_2 \).

Proof: The proof happens by case analysis on the five instructions in Figure 7. For brevity, we show two cases:

First case: \( \iota \) is \( x_1 \rightarrow x_2 + n \). Notice that \( n > 0 \) because otherwise our range analysis would have led us to transform that instruction into a subtraction, or would have produced no constraint at all. Henceforth, we shall write “\( f \setminus a \rightarrow b' \)” to denote function update, i.e., “\( \lambda x. if x = a \ then b \ else f(x) \)”.

We have that \( x_1 = x_2 + n \vdash \Sigma_1 \rightarrow \Sigma_1 \setminus x_1 \mapsto \Sigma_1(x_2 + n) \), and \( x_1 = x_2 + n \vdash LT_1 \triangleright LT_1 \setminus x_1 \mapsto LT_1(x_2) \cup \{x_2\} \). We look into possible variables \( x \in LT_1 \setminus x_1 \rightarrow LT_1(x_2) \cup \{x_2\} \):

- \( x = x_2 \): the theorem is true because \( \Sigma_2(x_2 + n) > \Sigma_2(x_2) \);
- \( x \in LT_2(x_2) \): From the hypothesis \( LT_2 \models \Sigma_1 \), we have that \( x < x_2 \). We know that \( x_2 < x_1 \); thus, by transitivity, \( x < x_1 \).

Second case: \( \iota \) is a \( \phi \)-function. Then, \( x = \phi(x_1, \ldots, x_n) \vdash \Sigma_1 \rightarrow \Sigma_1 \setminus x \mapsto \Sigma_1(x_1), \) for some \( i, 1 \leq i \leq n \), depending on the program’s dynamic control flow. From Figure 7, we have that \( LT_2 = LT_1 \setminus x \rightarrow LT_1(x_1) \cap \ldots \cap LT_1(x_n) \). Thus, any \( x' \in LT_2(x) \) is such that \( x' \subseteq LT_1(x_1) \). By
the hypothesis, $x' < x_i$. By the semantics of $\phi$-functions, $x = x_i$; hence, $x' < x$. □

**Corollary 3.10 (Invariance).** Let $x_i$ and $x_j$ be two variables simultaneously alive. If $x_i \in \text{LT}(x_j)$, then $x_i < x_j$.

**Proof:** In a SSA-form program, if two variables interfere, then one is alive at the definition point of the other \[8\] \[46\]. From Lemma 3.8 we know that \text{LT}(x)$ is constant along the live range of $x$. Theorem [39] gives us that this property holds at the definition point of the variables. □

### 3.6 Pointer Disambiguation

Pointers, in low-level languages, are used in conjunction with integer offsets to refer to specific memory locations. The combination of a base pointer plus an offset produces what we call a derived pointer. The less-than check that we have discussed in this paper lets us compare pointers directly, if they are bound to a less-than relation, or indirectly, if they are derived from a common base. This observation lets us state the disambiguation criteria below:

**Definition 3.11 (Pointer Disambiguation Criteria).** Let $p_1$ and $p_2$ be variables of pointer type, and $x_1$ and $x_2$ be variables (not constants) of arithmetic type. We consider two disambiguation criteria:

1. **Memory locations $p_1$ and $p_2$ will not alias if $p_1 \in \text{LT}(p_2)$ or $p_2 \in \text{LT}(p_1)$.
2. **Memory locations $p_1 = p + x_1$ and $p_2 = p + x_2$ will not alias if $x_1 \in \text{LT}(x_2)$ or $x_2 \in \text{LT}(x_1)$.

The C standard refers to arithmetic types and pointer types collectively as scalar types \[18\] \[6.2.5.21\]. Notice that the less-than analysis that we have discussed thus far works seamlessly for scalars; thus, it also builds relations between pointers. For instance, the common idiom "for(int* $pi = p$; $pi < p$; $pi++$);" gives us that $pi < p$ inside the loop. This fact justifies Definition 3.11 (1). Along similar lines, if $p_1 = p + x_1$, we have that $p \in \text{LT}(p_1)$; thus, Definition 3.11 (2) lets us disambiguate a base pointer from its non-null offsets, e.g., $p \neq p + n$, if we know that $n \neq 0$.

Definition 3.11 provides one, among several criteria, that can be used to disambiguate pointers. For instance, the C standard says that pointers of different types cannot alias. Aliasing is also impossible in well-defined programs between references derived from non-aliased base pointers. Additionally, derived pointers whose offsets have non-overlapping ranges cannot alias, as discussed in previous work \[4\] \[27\] \[51\]. Thus, our analysis says nothing about $p_1$ and $p_2$ in scenarios as simple as: $p_1 = \text{malloc}();$ $p_2 = \text{malloc}()$, or $p_1 = p + 1$; $p_2 = p + 2$. Previous work is already able to disambiguate $p_1$ and $p_2$ in both cases. Our pointer disambiguation criterion does not compete against these other approaches. Rather, as we further explain in Section 5 it complements them.
### Figure 8. Effectiveness of our alias analysis (LT), when compared to LLVM's basic alias analysis on the 100 largest benchmarks in the LLVM test suite. Each point in the X-axis represents one benchmark. The Y-axis represents total number of queries (one query per pair of pointers), and number of queries in which each algorithm got a “no-alias” response.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># Queries</th>
<th>BA</th>
<th>LT</th>
<th>BA + LT</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbm</td>
<td>31,944</td>
<td>5.90%</td>
<td>10.15%</td>
<td>15.74%</td>
</tr>
<tr>
<td>mcf</td>
<td>49,133</td>
<td>15.28%</td>
<td>8.95%</td>
<td>16.52%</td>
</tr>
<tr>
<td>astar</td>
<td>95,098</td>
<td>45.54%</td>
<td>16.05%</td>
<td>47.66%</td>
</tr>
<tr>
<td>libq</td>
<td>146,301</td>
<td>51.64%</td>
<td>3.45%</td>
<td>52.67%</td>
</tr>
<tr>
<td>sjeng</td>
<td>428,082</td>
<td>70.64%</td>
<td>2.03%</td>
<td>71.64%</td>
</tr>
<tr>
<td>milc</td>
<td>808,471</td>
<td>31.05%</td>
<td>23.90%</td>
<td>43.88%</td>
</tr>
<tr>
<td>soplex</td>
<td>1,787,190</td>
<td>21.43%</td>
<td>12.48%</td>
<td>23.53%</td>
</tr>
<tr>
<td>bzip2</td>
<td>2,472,234</td>
<td>21.48%</td>
<td>23.90%</td>
<td>26.70%</td>
</tr>
<tr>
<td>hmem</td>
<td>2,574,217</td>
<td>3.97%</td>
<td>4.48%</td>
<td>9.38%</td>
</tr>
<tr>
<td>gobmk</td>
<td>3,492,577</td>
<td>48.49%</td>
<td>22.91%</td>
<td>71.33%</td>
</tr>
<tr>
<td>namd</td>
<td>3,685,838</td>
<td>22.59%</td>
<td>0.93%</td>
<td>23.72%</td>
</tr>
<tr>
<td>omnetpp</td>
<td>12,943,554</td>
<td>18.71%</td>
<td>0.46%</td>
<td>18.18%</td>
</tr>
<tr>
<td>h264ref</td>
<td>20,068,605</td>
<td>12.86%</td>
<td>1.29%</td>
<td>13.16%</td>
</tr>
<tr>
<td>perl</td>
<td>23,849,576</td>
<td>9.92%</td>
<td>3.87%</td>
<td>13.79%</td>
</tr>
<tr>
<td>dealII</td>
<td>37,779,455</td>
<td>75.05%</td>
<td>20.21%</td>
<td>75.46%</td>
</tr>
<tr>
<td>gcc</td>
<td>186,008,992</td>
<td>4.26%</td>
<td>1.47%</td>
<td>4.65%</td>
</tr>
</tbody>
</table>

### Figure 9. Comparison between three alias analyses in SPEC 2006. “# Queries” is the total number of queries performed when testing a given benchmark. Percentages show the ratio of queries that yield “no-alias”, given a certain alias analysis. The higher the percentage, the more precise is the pointer disambiguation method. We have highlighted the cases in which our less-than check has increased by 10% or higher the precision of LLVM’s basic alias analysis.

BA cannot. For the entire LLVM test suite, our analysis (referred as LT), increases the precision of BA by 9.49% (56,192,064 vs 59,184,181 no-alias responses). Yet, in programs that make heavy use of pointer arithmetics, our results are even more impressive. For instance, in SPEC’s lbm we disambiguate 11,881 pairs of pointers, whereas BA provides precise answers to only 1,888. And, even in situations where LT falls behind BA, the former can increase the precision of the latter non-trivially. As an example, in SPEC’s gobmk, LT returns 852,368 “no-alias” answers and BA 1,705,559. Yet, these sets are mostly disjoint: the combination of both analyses solves 2,274,936 queries: an increase of 34% over BA. Figure 9 summarizes these results for SPEC 2006.

### How do we compare against Andersen’s analysis?
Andersen’s analysis is the quintessential pointer disambiguation technique. At the time of this writing, the most up-to-date version of LLVM did not contain an implementation of such technique. However, there exist algorithms available for LLVM 4.0, which are not yet part of the official distribution, such as Su’s [35] or Chen’s [9]. We have experimented with the latter. Henceforth, we shall call it CF, because it uses context free languages (CFL) to model the inclusion-based resolution of constraints, as proposed by Zheng and Rugina [47], and by Zhang et al. [45]. A detailed description of Chen’s implementation is publicly available.

Figure 10 compares our analysis and Andersen’s. Our numbers have been obtained in LLVM 3.7, whereas CF’s has been produced via LLVM 4.0. We emphasize that both versions of this compiler produce exactly the same number of alias queries, and, more importantly, BA outputs exactly the same answers in both cases. This experiment reveals that there is no clear winner in this alias analysis context. BA+LT is more than 20% more precise than BA+CF in three benchmarks: lbm, m11c and gobmk. BA+CF, in turn, is three times more precise in omnetpp. The main conclusions that we draw from this comparison are the following: (i) these analysis are complementary; and (ii) mainstream compilers still miss opportunities to disambiguate alias queries.

### 4.2 Scalability
We claim that the less-than analysis that we introduce in this paper presents – in practice – linear complexity on the size

---

1Available at [https://github.com/grievejia/andersen](https://github.com/grievejia/andersen)
of the target program. Size is measured as the number of instructions present in the intermediate representation of said program. In this section we provide evidence that corroborates this claim. Figure 11 relates the number of constraints that we produce for a program, using the rules in Figure 7, with the number of instructions in that program. We show results for our 50 largest benchmarks, in number of instructions, taken from SPEC and the LLVM test suite. The strong linear relation between these two quantities is visually apparent in Figure 11. And, going beyond visual clues, the coefficient of determination ($R^2$) between constraints and instructions is 0.992. The closer to 1.0 is $R^2$, the stronger the evidence of a linear behavior.

Figure 11. Comparison between the number of instructions and the number of constraints that we produce (using rules in Figure 7) per benchmark. X-axis represents benchmarks, sorted by number of instructions. The coefficient of determination ($R^2$) between these two metrics is 0.992, indicating a strong linear correlation.

As Figure 11 shows, the number of constraints that we produce is linearly proportional to the number of instructions they represent. But, what about the time to solve such constraints – is it also linear on the number of instructions? To solve constraints, we compute the transitive closure of the “less-than” relation between program variables. We use a cubic algorithm to build the transitive closure [25]. When fed with our benchmarks, this algorithm is likely to show linear behavior: the coefficient of determination between the number of constraints for all our benchmarks, and the runtime of our analysis is 0.988. This linearity surfaces in practice because most of the constraints enter the worklist at most twice. For instance, SPEC CPU 2006, plus the 308 programs that are part of the LLVM test suite give us 8,392,822 constraints to solve. For this lot, we pop the worklist 17,800,102 times: a ratio that indicates that each constraint is visited 2.12 times until a fixed point is achieved. We emphasize that our implementation still bears the status of a research prototype. Its runtime is far from being competitive, because, currently, it relies heavily on C++’s standard data-structures, instead of using data-types more customized to do static analyses. We use `std::set` to represent LT sets, `std::map` to bind LT sets to variables, and `std::vector` to implement the worklist. Therefore, our implementation still has much room to improve in terms of runtime. For instance, we took 340 seconds to solve all the less-than relations between all the scalar variables found in the 16 programs of SPEC CPU that the LLVM’s C frontend compiles on an 2.4GHz Intel Core i7. We have already observed that most of the LT sets end up empty, and that the vast majority of them, over 95%, contain only two or less elements. We intend to use such observations to improve the runtime of our analysis as future work.

4.3 Applicability

One way to measure the applicability of an alias analysis is to probe how it improves the quality of some compiler optimization, or the precision of other static analyses. In this work, we have opted to follow the second road, and show how our new alias analysis improves the construction of the Program Dependence Graph (PDG), a classic data structure
introduced by Ferrante et al. [13]. We use the implementation of PDGs available in the FlowTracker system [29], which has a distribution for LLVM 3.7. The PDG is a graph whose vertices represent program variables and memory locations, and the edges represent dependences between these entities. An instruction such as \(a[i] = b\) creates a data dependence edge from \(b\) to the memory node \(a[i]\). The more memory nodes the PDG contains, the more precise it is.

Figure 12 shows the results that we got in this experiment. We have produced 120 random programs, whose size vary from 50 to 4,030 lines. In total, the 120 PDGs produced with \(\text{BA}\) contain 1,299 memory nodes. When combined, \(\text{BA}\) and \(\text{LT}\) yield 8,114 nodes, an increase of 6.23x. We are much more precise than \(\text{BA}\) because the programs that Csmith produces do not read input values; they use constants instead. Because almost every memory indexing expression is formed by constants known at compilation time, \(\text{LT}\) can distinguish most of them. Although artificial, this experiment reveals a striking inability of LLVM’s current alias analyses to deal with pointer arithmetics. None of the other alias analyses available in LLVM 3.7 are able to increase the precision of \(\text{BA}\) – not even marginally. Although we have not used \(\text{CF}\) in this experiment – it is not available for LLVM 3.7 – we speculate, from reading its source code, that it will not be able to change this scenario.

Notice that our results do not depend on the nesting depth of pointers. Our 120 benchmarks contain 6 categories of programs, which we produced by varying the nesting depth of pointers from 2 to 7 levels. Thus, we had 20 programs in each category. A pointer to \(\text{int}\) of nesting depth 3, for instance, is declared as \(\text{int}***\). All these programs, regardless of their category, present an average of six static memory allocation sites. On average, \(\text{BA}\) produces PDGs with 11 memory nodes, independent on the bucket, and \(\text{BA}+\text{LT}\) produce PDGs with 68. The greater the number of static memory allocation sites, the better the results of both \(\text{BA}\) and \(\text{LT}\). The largest PDG observed with \(\text{BA}\) only has 52 memory nodes (and 88 nodes if we augment \(\text{BA}\) with \(\text{LT}\)). The largest graph produced by the combination of \(\text{BA}\) and \(\text{LT}\) has 342 nodes (and only 15 nodes if we use \(\text{BA}\) without \(\text{LT}\)).

5. Related Work

The insight of using a less-than dataflow analysis to disambiguate pointers is an original contribution of this paper. However, such a static analysis is not new, having been used before to eliminate array bound checks. We know of two different approaches to build less-than relations: Logozzo’s [20] [21] and Bodik’s [7]. Additionally, there exist non-relational analyses that produce enough information to solve less-than equations [11] [23]. In the rest of this section we discuss the differences between such work and ours.

The ABCD Algorithm. The work that most closely resembles ours is Bodik et al.’s ABCD (short for Array Bounds Checks on Demand) algorithm [7]. Similarities stem from the fact that Bodik et al. also build a new program representation to achieve a sparse less-than analysis. However, there are five key differences between that approach and ours. The first difference is a matter of presentation: Bodik et al. provide a geometric interpretation to the problem of building less-than relations, whereas we adopt an algebraic formalization. Bodik et al. keep track of such relations via a data-structure called the inequality graph. This graph is implicit in our approach: it appears if we create a vertex \(v_i\) to represent each program variable \(x_i\), and add a weighted edge
from $v_1$ to $v_2$ if, and only if, $v_1 \in LT(v_2)$. The weight of an edge is the difference $v_2 - v_1$, whenever known statically. The other four differences are more fundamental.

Bodik et al. use a different algorithm to prove that a variable is less than another. In the absence of cycles in the inequality graph, their approach works like ours: a positive path between $v_i$ to $v_j$ indicates that $x_i < x_j$. This path is implicit in the transitive closure that we produce after solving constraints. However, they use an extra step to handle cycles, which, in our opinion, makes their algorithm difficult to reason about. Upon finding a cycle in the inequality graph, Bodik et al. try to mark this cycle as increasing or decreasing. Cycles always exist due to $\phi$-functions. Decreasing cycles cause $\phi$-functions to be abstractly evaluated with the minimum operator applied on the weights of incoming edges; increasing cycles invoke maximum instead. Third, Bodik et al. do not use range analysis. This is understandable, because ABCD has been designed for just-in-time compilers, where runtime is an issue. Nevertheless, this limitation prevents ABCD from handling instructions such as $x_1 = x_2 + x_3$ if neither $x_2$ nor $x_3$ are constants. Fourth, Bodik et al.’s program representation does not split the live range of $x_3$ at an instruction such as $x_1 = x_2 - x_3, x_3 > 0$. This implementation detail lets us know that $x_2 > x_3$. Finally, we chose to compute a transitive closure of less-than relations, whereas ABCD works on demand.

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

**Fully-Relational Analyses.** Our less-than analysis, ABCD and Pentagons are said to be semi-relational, meaning that they associate single program variables with sets of other variables. Fully-relational analysis, such as Octogons [23] or Polyhedrons [11], associate tuples of variables with abstract information. For instance, Miné’s Octogons build relations such as $x_1 + x_2 \leq 1$, where $x_1$ and $x_2$ are variables in the target program. As an example, Polly-LLVM uses fully-relational techniques to analyze loops. Polly’s dependence analysis is able to distinguish $v[i]$ and $v[j]$ in Figure 1(a), given that $j - i \geq 1$; however, it cannot analyze $v[i]$ and $v[j]$ in Figure 1(b). These analyses are very powerful; however, they face scalability problems when dealing with large programs. Whereas a semi-relational sparse analysis generates $O(|V|)$ constraints, $|V|$ being the number of program variables, a relational one might produce $O(|V|^{k})$, $k$ being the number of variables used in relations.

**Range-Based Alias Analyses** There exist different efficient pointer disambiguation strategies that associate ranges with pointers $[2, 4, 5, 24, 27, 31, 32, 37, 39, 41]$. They all share a common idea: two memory addresses $p_1 + [l_1, u_1]$ and $p_2 + [l_2, u_2]$ do not alias if the intervals $[p_1 + l_1, p_1 + u_1]$ and $[p_2 + l_2, p_2 + u_2]$ do not overlap. These analyses differ in the way they represent intervals, e.g., with holes [4, 5, 37] or contiguously [2, 32]; with symbolic bounds [24, 27, 31] or with numeric bounds [4, 5, 37], etc. None of these previous work is strictly better than ours. For instance, none of them can disambiguate $v[i]$ and $v[j]$ in Figure 1(b), because these locations cover regions that overlap, albeit not at the same time. Nevertheless, range based disambiguation methods can solve queries that our less-than approach cannot. As an example, we are unable to disambiguate $p_1$ and $p_2$, given these definitions: $p_1 = p + 1$ and $p_2 = p + 2$. We know that $p < p_1$ and $p < p_2$, but we do not relate $p_1$ and $p_2$.

6. Conclusion

This paper has introduced a new technique to disambiguate pointers, which relies on a less-than analysis. Our new alias analysis uses the observation that if $p_1$ and $p_2$ are two pointers, such that $p_1 < p_2$, then they cannot alias. Even though this observation is obvious, it has not been used before as the cornerstone of an alias analysis. Testimony of this statement is the fact that our analysis has been able to improve the precision of LLVM’s standard suite of pointer disambiguation techniques by a large factor in some benchmarks. There are several ways in which our idea can be further developed. One future avenue that is particularly appealing to us concerns speed. Currently, our research prototype can handle large programs, but its runtime is not practical: it takes several minutes to produce the transitive closure of the less-than relation for our largest benchmarks. We believe that better algorithms can improve this scenario substantially. The design of such algorithms is a problem that we leave open.

**Acknowledgment**

This project is supported by CNPq, Intel (The eCoSoC grant), FAPEMIG (The Prospiel project), and by the French National Research Agency – ANR (LABEX MILYON of Université de Lyon, within the program “Investissement d’Avenir” (ANR-11-IDEX-0007)). We thank the CGO referees for the very insightful comments and suggestions.
References


A. Artifact Evaluation

This artifact is in the form of a virtual machine and it is distributed through the link: http://cuda.dcc.ufmg.br/CGO17/strict-inequalities/ This virtual machine contains the code and execution scripts for the work presented in this paper. In the desktop folder, you will find ways to run our analysis on any source code you choose, or on the benchmarks used in this paper's evaluation.

A.1 The Code

The code from this work can be found inside LLVM's build directory inside the virtual machine. The LLVM source and build directory is at /llvm/, and the code for the analyses are in /llvm/lib/Transforms/. In total we have used 4 passes in this paper:

- /llvm/lib/Transforms/sraa
  The main pointer alias analysis presented in this paper.
- /llvm/lib/Transforms/vSSA
  A pass that transforms the program to the e-SSA form, needed in order to use the Strict Relations Alias Analysis.
- /llvm/lib/Transforms/RangeAnalysis
  A symbolic Range Analysis also used in the Strict Relations Alias Analysis.
- /llvm/lib/Transforms/DepGraph
  A pass that builds the program dependence graph. This pass is used in one of our tests, to show the effectiveness of our new alias analysis when comparing the number of memory nodes yielded by the DepGraph when running it coupled with the Strict Relations Alias Analysis. We developed our work on LLVM 3.7.

A.2 Getting Started

The artifact is provided as a Virtual Machine image. To use the artifact, you need to download and install the VirtualBox VM player at https://www.virtualbox.org/wiki/Downloads After downloading and installing VirtualBox, to run the VM image, you must perform the following steps:

1. Download and decompress the VM image.
2. OpenVirtualBox.
3. Create a new virtual machine, by clicking in “Machine”, “New...”.
4. Give a name to the new VM, by filling the “Name” field.
5. In “Type”, select “Linux”.
6. In “Version”, select “Ubuntu (64 bit)”.
7. Click “continue”, select the desired amount of RAM memory, then click “continue” again.
8. In the hard drive selection screen, select the option “Use an existing virtual hard drive file”, then select the .vdi file containing the VM image that you just downloaded.
9. Your Virtual Machine is ready to use! To start, simply double-click it in the VirtualBox VM list.

If the VM prompts for a username and password, you may use the following combination:

- login: cgoartifact
- password: artifact

A.3 Running the Examples

Inside the folder ExecuteExamples you may find a script to compile any C code and run our analysis. If you wish to compile a program, you may use the script compile.sh and pass as first parameter the name of the program you wish to compile. For example, say you want to compile the program test.c:

$ bash compile.sh test

You may omit the file extension (.c) when passing the name of the program as a parameter. After the compilation, if you wish to run our Strict Relations Alias Analysis evaluation, you can run the script sraa.sh. The output will be the LLVM's default alias analysis evaluation, with a thorough list of alias information on the program.

$ bash sraa.sh test

If you wish to compare it with the LLVM's basic alias analysis alone, you can also run the script basicsaa.sh on any program you wish. You may also use the script random.sh to generate a random C program with the tool Csmith:

$ bash random.sh random

The clean.sh script will remove all files created by the other scripts.

A.4 Running the Benchmarks

We have executed 3 main evaluation tests in our paper: Precision, Scalability and Applicability. These tests are described in Section 4 of this paper. In the folder Benchmarks you will find the scripts to run the benchmarks used in the paper. You will find two folders: aaeval and memnodes. In the folder aaeval there is a script called run.sh that runs two of the 3 evaluation tests we ran in our paper, the tests of Precision and Scalability. Both tests are executed on the LLVM's test-suite and on the SpecCPU2006. The test of Precision evaluates how better our alias analysis is compared to the Basic Alias Analysis provided by LLVM. So it runs the basicaa, the sraa and both together on the entire test-suite. The test of Scalability evaluates the growth of the number of constraints generated during the Strict Relations Alias Analysis compared to the size of the programs being analyzed. The script run.sh will basically run the tests on the entire test-suite and copy the .csv files containing the statistics to the folder aaeval. At the end of the execution you may find...
the .csv files in the folder. **Warning:** Bear in mind that the tests may take a while (up to 4 hours), mainly because they are running on a virtual machine.

In the folder *memnodes* you may find another script called *run.sh* that executes the third test in our paper, the test of Applicability. The test of Applicability runs a Dependence Graph pass that generates a PDG (Program Dependence Graph) and counts the number of memory nodes in it. Our aim is to show that our new alias analysis yields more memory nodes in these graphs, because it is more precise. We also run this test in the entire test-suite plus SPEC-Cpu2006. In order to do improve our evaluation, we used the tool *Csmith* to generate 120 random C programs and then we run the Dependence Graph coupled with our Strict Relations Alias Analysis in all of them. We showed that the PDG generated without our analysis has less memory nodes than the PDG generated with our alias analysis. The random generated C programs are in the folder *SingleSource*, inside the LLVM's test-suite root, so this test already evaluates the entire test-suite and the random generated programs as well.

To use *csmith* to generate a random program, use the script *random.sh* in the folder *ExecuteExamples*.

### A.5 The Output

The output for our analysis is very simple. We use the *aa-eval* pass to compare, within the same function, all possible pairs of pointers and then we return how many comparisons issued a NoAlias, MayAlias or a MustAlias response. Regarding the Dependence Graph, we basically count the number of Memory Nodes yielded. The output for this analysis is formatted in CSV files. When you run the tests with the script *run.sh*, the CSV files will be copied to your working directory.

### Reviewing Methodology

This artifact has been reviewed according to the guidelines established by the *Artifact Evaluation Committee of CGO and PPoPP*. The reviewing methodology is described in [http://cTuning.org/ae/submission-20161020.html](http://cTuning.org/ae/submission-20161020.html).