Analysing C programs: arrays, pointers with precision and scale

IBISC Seminar

Laure Gonnord and Maroua Maalej (LIP) and Fernando Pereira (UFMG, Brasil) and . . .

University of Lyon / LIP

October 13th, 2016
Plan

Motivations and big picture

Technical context: LLVM

Symbolic range analysis

Application 1: GreenArrays toolsuite
  Overview
  A bit on the other analyses
  Experimental results
  Conclusion

Application 2: Symbolic Pointer Analysis
  Overview
  Experimental results

Conclusion
Goal: Safety

Prove that (some) memory accesses are safe:

```c
int main () {
    int v[10] ;
    v[0]=0; ✔
    return v[20] ✗
}
```

- Fight against bugs and overflow attacks.
Goal: Performance

Prove that (some) memory accesses are non-overlapping:

```c
void fill_array(char* p)
    int i;
    for (i=0;i<4;i++) {
        p[i]=0;
    }
    for (i=4;i<8;i++) {
        p[i]=2*i;
    }
```

Base analysis for automatic parallelisation, code motion, ...
Issues

Challenges:

- Thousands, millions of lines of code to analyse
- Compilers are complex programs.
- Growing need for simple **specialized** analyses that **scale**
Scaling analyses: two experiences

In this talk, I will present:

▶ (OOPSLA’14) The design of an efficient symbolic range analysis.

▶ (OOPSLA’14) Its application to the array out-of-bound problem.

▶ (CG0’16) Its use in pointer disambiguisation.
Plan

Motivations and big picture

**Technical context: LLVM**

Symbolic range analysis

Application 1: GreenArrays toolsuite
  - Overview
  - A bit on the other analyses
  - Experimental results
  - Conclusion

Application 2: Symbolic Pointer Analysis
  - Overview
  - Experimental results

Conclusion
A bit on LLVM

LLVM is a **compiler infrastructure**:  
- Open source  
- Various frontends (C, C++, Fortran)  
- Various code generators (x86, …)

Writing optimisations is easier:  
- A unique IR (**intermediate representation**)  
- C++ iterators (functions, blocks, …)
LLVM representation: SSA form

\[
\begin{align*}
x & \leftarrow 5 \\
x & \leftarrow x - 3 \\
x & < 3 ?
\end{align*}
\]

\[
\begin{align*}
y & \leftarrow 2 \times x \\
w & \leftarrow y \\
w & \leftarrow x - y \\
z & \leftarrow x + y
\end{align*}
\]
void simple_loop_constant() {
    for (unsigned i = 0; i < 10; i++) {
        // Do nothing
    }
}
Plan

Motivations and big picture

Technical context: LLVM

Symbolic range analysis

Application 1: GreenArrays toolsuite
  Overview
  A bit on the other analyses
  Experimental results
  Conclusion

Application 2: Symbolic Pointer Analysis
  Overview
  Experimental results

Conclusion
Symbolic Ranges (SRA): Goal

\[ v_0 = \text{alloca}(\text{argc}) \]
\[ i_0 = \text{argc} - 1 \]
\[ v_4 = v_0 + i_0 \]
\[ *v_4 = 0 \]

\[ R(i_0) = [\text{argc} - 1, \text{argc} - 1] \]

\[ v_1 = \text{alloca}(2) \]
\[ i_1 = 1 \]

\[ R(i_1) = [1, 1] \]

\[ v_2 = \phi(v_0, v_1) \]
\[ i_2 = \phi(i_0, i_1) \]
\[ v_3 = v_2 + i_2 \]
\[ *v_3 = 0 \]

\[ R(i_2) = [\min(1, \text{argc} - 1), \max(1, \text{argc} - 1)] \]
SRA on SSA form: a sparse analysis

▶ An abstract interpretation-based technique.
▶ Very similar to classic range analysis.
▶ One abstract value (R) per variable: sparsity.
▶ Easy to implement (simple algorithm, simple data structure).
SRA on SSA form: constraint system

\[ v = \bullet \quad \Rightarrow \quad R(v) = [v, v] \]

\[ v = o \quad \Rightarrow \quad R(v) = R(o) \]

\[ v = v_1 \oplus v_2 \quad \Rightarrow \quad R(v) = R(v_1) \oplus^I R(v_2) \]

\[ v = \phi(v_1, v_2) \quad \Rightarrow \quad R(v) = R(v_1) \sqcup R(v_2) \]

other instructions \quad \Rightarrow \quad \emptyset

\[ \oplus^I \]: abstract effect of the operation \( \oplus \) on two intervals.

\[ \sqcup \]: convex hull of two intervals. ▶ All these operation are performed symbolically thanks to \textbf{GiNaC}
SRA on SSA form: an example

\[
\begin{align*}
N &= \text{randunsigned}() \\
i_0 &= 0
\end{align*}
\]

\[
\begin{align*}
i_1 &= \phi(i_0, i_2) \\
i_1 &< N ~? \\
i_2 &= i_1 + 1
\end{align*}
\]

- \( R(i_0) = [0, 0] \)
- \( R(i_1) = [0, +\infty] \)
- \( R(i_2) = [1, +\infty] \)
Improving precision of SRA: live-range splitting

1/2

\[ i_0 = \text{argc} - 1 \]
\( (i_0 < 10) \)?

\[ v_1 = v_0 + i_0 \]
\[ *v_1 = 0 \]

\[ v_2 = v_0 + i_0 \]
\[ *v_2 = 0 \]

\[ i_1 = i_0 \cap [\infty, 9] \]
\[ v_1 = v_0 + i_1 \]
\[ *v_1 = 0 \]

\[ i_2 = i_0 \cap [10, +\infty] \]
\[ v_2 = v_0 + i_2 \]
\[ *v_2 = 0 \]

\[ R(i_1) = [\text{argc} - 1, \max(9, \text{argc} - 1)] \]

\[ R(i_2) = [\min(10, \text{argc} - 1), \text{argc} - 1] \]

\[ e\text{-SSA form.} \]
Improving precision of SRA: live-range splitting

Rule for live-range splitting:

\[
\begin{align*}
t &= a < b \\
\bar{r}(t, \ell) &
\end{align*}
\]

\[
\begin{align*}
a_f &= \sigma(a) \\
b_f &= \sigma(b) \\
a_t &= \sigma(a) \\
b_t &= \sigma(b)
\end{align*}
\]

\[
\begin{align*}
R(a_t) &= [R(a)_\downarrow, \min(R(b)_\uparrow - 1, R(a)_\uparrow)] \\
R(b_t) &= [\max(R(a)_\downarrow + 1, R(a)_\downarrow), R(b)_\uparrow] \\
R(a_f) &= [\max(R(a)_\downarrow, R(a)_\uparrow), R(a)_\uparrow] \\
R(b_f) &= [R(b)_\downarrow, \min(R(a)_\uparrow, R(b)_\uparrow)]
\end{align*}
\]

- All simplications are done by GiNaC.
SRA + live-range on an example

\[
\begin{align*}
N &= \text{randunsigned}() \\
i_0 &= 0 \\
i_1 &= \phi(i_0, i_2) \\
i_1 &< N ? \\
i_t &= \sigma(i_1) \\
i_2 &= i_t + 1
\end{align*}
\]

\[
R(i_t) = [R(i_1) \downarrow, min(N - 1, R(i_1) \uparrow)]
\]

- \( R(i_0) = [0, 0] \)
- \( R(i_1) = [0, N] \)
Plan

Motivations and big picture

Technical context: LLVM

Symbolic range analysis

Application 1: GreenArrays toolsuite
  Overview
  A bit on the other analyses
  Experimental results
  Conclusion

Application 2: Symbolic Pointer Analysis
  Overview
  Experimental results

Conclusion
A technique to prove that (some) memory accesses are safe:
  - Less need for additional guards.
  - Based on abstract interpretation.
  - Precision and cost compromise.

Implemented in LLVM-compiler infrastructure:
  - Eliminate 50% of the guards inserted by AddressSanitizer
  - SPEC CPU 2006 17% faster
Our key insight: Symbolic (parametric) Analyses

```c
int main(int argc, char** a) {
    char* p = malloc(argc);
    int i = 0;
    while (i < argc) {
        p[i] = 0;
        i++;
    }
    return 0;
}
```

$R(i) \subseteq W(p)$ thus $p[i]$ is safe.
A bit on sanitizing memory accesses

Different techniques: but all have an overhead.

Ex: Address Sanitizer

- Shadow every memory allocated: 1 byte → 1 bit (allocated or not).
- Guard every array access: check if its shadow bit is valid.
  - Slows down SPEC CPU 2006 by 25%
- We want to remove these guards.
1. int main(int argc, char** argv) {
2.     int size = argc + 1;
3.     char* buf = malloc(size);
4.     unsigned index = 0;
5.     scanf("%u", &index);
6.     if (index < argc) {
7.         buf[index] = 0;
8.     }
9.     return index;
10. }

Inside the branch index is at least 0 and at most argc-1

As long as we do not have integer overflows!

We know that "argc - 1" is less than argc

Any address from buf + 0 to buf + argc is safe!
Symbolic Range Analysis:
finds the lower and upper values that variables can assume

Symbolic Region Analysis:
finds the lower and upper values that a pointer can address

Integer Overflow Analysis:
Which arithmetic operations can overflow?

- Any address from buf + 0 to buf + argc is safe!
- Inside the branch index is at least 0 and at most argc-1
- We know that "argc - 1" is less than argc
- As long as we do not have integer overflows!
Compute (an underapproximation of) the range of valid accesses from base pointers:

\[ v_1 = \text{alloc}(n) \]
\[ v_2 = v_1 + 1 \]
\[ v_3 = v_1 + n \]

\[ W(v_1) = [0, n - 1] \]
\[ W(v_2) = [-1, n - 2] \]
\[ W(v_3) = [-n, -1] \]
Symbolic regions 2/2: An example

\( v_0 = \text{alloca}(\text{argc}) \)
\( i_0 = \text{argc} - 1 \)
\( v_4 = v_0 + i_0 \)
\( *v_4 = 0 \)

\( W(v_0) = [0, \text{argc} - 1] \)

\( v_1 = \text{alloca}(2) \)
\( i_1 = 1 \)

\( W(v_1) = [0, 1] \)

\( v_2 = \phi(v_0, v_1) \)
\( i_2 = \phi(i_0, i_1) \)
\( v_3 = v_2 + i_2 \)
\( *v_3 = 0 \)

\( W(v_2) = [0, \min(1, \text{argc} - 1)] \)

\( W(v_3) = W(v_2) - R(i_2) = [0, \min(1, \text{argc} - 1) - \max(1, \text{argc} - 1)] \)
If \( 0 \in W(p) \), then \( *p \) is \textbf{safe}, else \textbf{DK}

\[
\begin{align*}
v_0 &= \text{alloca}(\text{argc}) \\
i_0 &= \text{argc} - 1 \\
v_4 &= v_0 + i_0 \\
*_{v_4} &= 0
\end{align*}
\]

This access is \textbf{not} safe (if \texttt{argc} == 0):
\[
W(v_4) = W(v_0) - R(i_0) = [-\text{argc} + 1, 0]
\]

\[
\begin{align*}
v_1 &= \text{alloca}(2) \\
i_1 &= 1
\end{align*}
\]

This access is \textbf{not} safe, because its upper bound can be negative:
\[
W(v_3) = W(v_2) - R(i_2) = [0, \min(1, \text{argc} - 1) - \max(1, \text{argc} - 1)]
\]
int main(int argc, char** argv) {
    int index = argc + 1;
    int size = index * index;
    char* buf = malloc(size);
    return buf[index];
}

Because we manipulate symbols, "argc + 1 < (argc + 1) * (argc + 1)"
only in the absence of integer overflows
Overflows 2/2

- We find every arithmetic operation that may influence memory allocation or memory indexing.

```c
int main(int argc, char** argv) {
    int index = argc + 1;
    int size = index * index;
    char* buf = malloc(size);
    return buf[index];
}
```

We instrument the code to detect overflows.
Experimental setup

- **Implementation**: LLVM + AddressSanitizer
- **Benchmarks**: SPEC CPU 2006 + LLVM test suite
- **Machine**: Intel(R) Xeon(R) 2.00GHz, with 15,360KB of cache and 16GB or RAM
- **Baseline**: Pentagons
  - Abstract interpretation that combines "less-than" and "integer ranges".†

```
int i = 0;
unsigned j = read();
if (...)
    i = 9;
if (j < i)
    ...
```

\[ P(j) = (\text{less than } \{i\}, [0, 8]) \]

Percentage of bound checks removed

The higher, the better.
Pentagons: 27%.
GreenArrays: 43%
Runtime improvement

The lower the bar, the faster. Time is normalized to AddressSanitizer without bound-check elimination. Average speedup: Pentagons = 9%. GreenArrays = 16%.
In the paper (OOPSLA’14)

A complete formalisation of all the analyses:
- Concrete and abstract semantics.
- Safety is proved.
- Interprocedural analysis.

https://code.google.com/p/ecosoc/

Remaining question: improving precision of the symbolic range analysis?
Plan

Motivations and big picture

Technical context: LLVM

Symbolic range analysis

Application 1: GreenArrays toolsuite
  Overview
  A bit on the other analyses
  Experimental results
  Conclusion

Application 2: Symbolic Pointer Analysis
  Overview
  Experimental results

Conclusion
Goal + Contribution (CGO ’16)

Goal:
- Optimizing languages with pointers;
- Solving pointer arithmetic, disambiguating pointers;
- Low cost analysis.

Contribution:
- Combine alias analysis with range analysis;
- Speed up;
Motivating example

```c
void fill_array (char* p) {
    int i;
    for (i = 0; i < 4; i++)
        p[i] = 0;
    for (i = 4; i < 8; i++)
        p[i] = 2 × i;
}
```

![Diagram of memory access patterns]
Motivating example

```c
void fill_array (char* p) {
    int i ;
    for (i = 0; i < 4; i++)
        p[i] = 0 ; // char* tmp0 = p + i ; *tmp0 = 0
    for (i = 4; i < 8; i++)
        p[i] = 2 × i ; // char* tmp1 = p + i ; *tmp0 = 2 × i
}
```

```
\[ p \quad p + 3 \quad p + 4 \quad p + 7 \]

\[ tmp0 \quad tmp1 \]
```
First step: Range Analysis on e-SSA

\[ p = malloc(10) \]

\[ i_0 = 0 \]

\[ i_2 = i_1 \cap [-\infty, 3] \]
\[ tmp_0 = p + i_2 \]
\[ *tmp_0 = 0 \]
\[ i_3 = i_2 + 1 \]

\[ i_1 = \phi(i_0, i_3) \]
\[ i_1 < 4? \]

\[ i_4 = i_1 \cap [4, +\infty] \]

\[ i_5 = \phi(i_4, i_6) \]
\[ i_5 < 8? \]

\[ i_7 = i_6 \cap [-\infty, 7] \]
\[ tmp_1 = p + i_7 \]
\[ *tmp_1 = 2 \times i_7 \]
\[ i_6 = i_7 + 1 \]

\[ halt \]
First step: Range Analysis on e-SSA

\[ p = \text{malloc}(10) \]  \quad \rightarrow \quad i_0 = 0

\[ i_2 = i_1 \cap [-\infty, 3] \]
\[ tmp_0 = p + i_2 \]
\[ *tmp_0 = 0 \]
\[ i_3 = i_2 + 1 \]

\[ R(i_3) = [0, 3] \]

\[ R(i_2) = [0, 3] \]

\[ i_1 = \phi(i_0, i_3) \]
\[ i_1 < 4? \]

\[ i_4 = i_1 \cap [4, +\infty] \]

\[ R(i_4) = [0, 4] \]

\[ i_5 = \phi(i_4, i_6) \]
\[ i_5 < 8? \]

\[ i_7 = i_6 \cap [-\infty, 7] \]
\[ tmp_1 = p + i_7 \]
\[ *tmp_1 = 2 \times i_7 \]
\[ i_6 = i_7 + 1 \]

\[ R(i_7) = [4, 7] \]

\[ R(i_5) = [4, 7] \]

\[ R(i_0) = [0, 0] \]

\[ halt \]
Pointer Range Analysis: Overview

Original program → Global symbolic range analysis on pointers → Global Test

Global symbolic range analysis on integers → Global Test

Local symbolic range analysis on pointers → Local Test

Q

Global Test → Do not alias

Local Test → (May/Do not) alias

(p₀, p₁)
Abstract Analysis

\[ p_i \leadsto loc_i + [l_i, u_i] \quad \text{if } i = j \text{ or } i \neq j \]

\[ p_j \leadsto loc_j + [l_j, u_j] \]

**Property (overapprox):**

If \((loc_i = loc_j \text{ and } [l_i, u_i] \cap [l_j, u_j] = \emptyset)\)

Then \(p_i\) and \(p_j\) **do not alias** Else **may alias**.
Global Pointer Range Analysis

Let \( n \) be the number of program sites where memory is allocated. We associate pointers with tuples of size \( n \):

\[
(SymbRanges \cup \bot)^n: \ GR(p) = (p_0, \ldots, p_{n-1}).
\]

Notation

\[
GR(p) = \{ loc_i + p_i, loc_j + p_j, \ldots \}
\]

Constraint System:

\[
j : p = \text{malloc} (v) \Rightarrow GR(p) = (\bot, \ldots, [0, 0]_j, \ldots)
\]

\[
v = v_1 \Rightarrow GR(v) = GR(v_1)
\]

\[
q = p + c \quad \text{with } c \text{ scalar} \Rightarrow q_i = \begin{cases} 
\bot \text{ if } p_i = \bot \\
p_i + R(c) \text{ else}
\end{cases}
\]

\[
q = \phi(p^1, p^2) \Rightarrow GR(q) = GR(p^1) \sqcup GR(p^2)
\]
Global Pointer Range Analysis

\[
\begin{align*}
&\text{p}_0 = \text{malloc}(3) ; \\
&\text{p}_1 = \text{malloc}(5) ; \\
&\text{if (...) } \text{p}_2 = \text{p}_1 ; \\
&\text{else } \text{p}_2 = \text{p}_1 + 1 ; \\
&\text{if (...) } \text{p}_3 = \text{p}_0 + 2 ; \\
&\text{else } \text{p}_3 = \text{p}_1 + 3
\end{align*}
\]

\[
\begin{align*}
&\text{GR} (\text{p}_0) = \text{loc}_0 + [0, 0] \\
&\text{GR} (\text{p}_1) = \text{loc}_1 + [0, 0] \\
&\text{GR} (\text{p}_2) = \text{loc}_1 + [0, 1] \\
&\text{GR} (\text{p}_3) = \{\text{loc}_0 + [2, 2], \text{loc}_1 + [3, 3]\}
\end{align*}
\]
Global Pointer Range Analysis

Example:

\[ p = \text{malloc}(10) \]

\[ i_0 = 0 \]

\[ \text{GR}(p) = \{ loc_0 + [0, 0] \} \]

\[ i_2 = i_1 \cap [-\infty, 3] \]
\[ tmp_0 = p + i_2 \]
\[ *tmp_0 = 0 \]
\[ i_3 = i_2 + 1 \]

\[ \text{GR}(tmp_0) = \{ loc_0 + [0, 3] \} \]

\[ R(i_2) = [0, 3] \]

\[ i_1 = \phi(i_0, i_3) \]
\[ i_1 < 4? \]

\[ i_4 = i_1 \cap [4, +\infty] \]

\[ i_5 = \phi(i_4, i_6) \]
\[ i_5 < 8? \]

\[ R(i_7) = [4, 7] \]

\[ i_7 = i_6 \cap [-\infty, 7] \]
\[ tmp_1 = p + i_7 \]
\[ *tmp_1 = 2 \times i_7 \]
\[ i_6 = i_7 + 1 \]

\[ \text{GR}(tmp_1) = \{ loc_0 + [4, 7] \} \]

\[ \text{halt} \]
Global Pointer Range Analysis

Example:

\[ p = \text{malloc}(10) \]
\[ i_0 = 0 \]
\[ i_2 = i_1 \cap [-\infty, 3] \]
\[ tmp_0 = p + i_2 \]
\[ *tmp_0 = 0 \]
\[ i_3 = i_2 + 1 \]
\[ i_1 = \phi(i_0, i_3) \]
\[ i_1 < 4? \]
\[ i_4 = i_1 \cap [4, +\infty] \]
\[ i_5 = \phi(i_4, i_6) \]
\[ i_5 < 8? \]
\[ i_7 = i_6 \cap [-\infty, 7] \]
\[ tmp_1 = p + i_7 \]
\[ *tmp_1 = 2 \times i_7 \]
\[ i_6 = i_7 + 1 \]
\[ \text{halt} \]

\[ R(i_2) = [0, 3] \]
\[ GR(tmp_0) = \{loc_0 + [0, 3]\} \]
\[ GR(p) = \{loc_0 + [0, 0]\} \]
\[ GR(tmp_1) = \{loc_0 + [4, 7]\} \]
\[ GR(tmp_0) = \{loc_0 + [0, 3]\} \]
\[ GR(p) = \{loc_0 + [0, 0]\} \]
\[ GR(tmp_1) = \{loc_0 + [4, 7]\} \]

\[ R(i_7) = [4, 7] \]
Local Pointer Range Analysis

**Motivation:**

\[ a_0 = \text{malloc} (N) \]; \quad \leadsto \quad \text{GR} (a_0) = \{\text{loc}_0 + [0, 0]\}

if (...) 
\[ a_1 = a_0 + 1 \]; \quad \leadsto \quad \text{GR} (a_1) = \{\text{loc}_0 + [1, 1]\}

else \quad a_2 = a_0 ; \quad \leadsto \quad \text{GR} (a_2) = \{\text{loc}_0 + [0, 0]\}

\[ a_3 = \phi(a_1, a_2) ; \quad \leadsto \quad \text{GR} (a_3) = \{\text{loc}_0 + [0, 1]\}\]

\[ a_4 = a_3 + 1 ; \quad \leadsto \quad \text{GR} (a_4) = \{\text{loc}_0 + [1, 2]\}\]

\[ a_5 = a_3 + 2 ; \quad \leadsto \quad \text{GR} (a_5) = \{\text{loc}_0 + [2, 3]\}\]

\[ [1, 2] \cap [2, 3] \neq \emptyset \]
Local Pointer Range Analysis

Motivation:

\[ a_0 = \text{malloc} \left( N \right) ; \quad \rightarrow \quad \text{GR} \left( a_0 \right) = \left\{ \text{loc}_0 + [0, 0] \right\} \]

if (...)

\[ a_1 = a_0 + 1 ; \quad \rightarrow \quad \text{GR} \left( a_1 \right) = \left\{ \text{loc}_0 + [1, 1] \right\} \]

else \[ a_2 = a_0 ; \quad \rightarrow \quad \text{GR} \left( a_2 \right) = \left\{ \text{loc}_0 + [0, 0] \right\} \]

\[ a_3 = \phi \left( a_1, a_2 \right) ; \quad \rightarrow \quad \text{GR} \left( a_3 \right) = \left\{ \text{loc}_0 + [0, 1] \right\} \quad \rightarrow \quad \text{LR} \left( a_3 \right) = \left\{ \text{loc}_1 + [0,0] \right\} \]

\[ a_4 = a_3 + 1 ; \quad \rightarrow \quad \text{GR} \left( a_4 \right) = \left\{ \text{loc}_0 + [1, 2] \right\} \quad \rightarrow \quad \text{LR} \left( a_4 \right) = \left\{ \text{loc}_1 + [1, 1] \right\} \]

\[ a_5 = a_3 + 2 ; \quad \rightarrow \quad \text{GR} \left( a_5 \right) = \left\{ \text{loc}_0 + [2, 3] \right\} \quad \rightarrow \quad \text{LR} \left( a_5 \right) = \left\{ \text{loc}_1 + [2, 2] \right\} \]

\[ [1, 2] \cap [2, 3] \neq \emptyset \]
Local Pointer Range Analysis

Constraint System:

\[ p = \text{malloc}(v) \quad \text{with } v \text{ scalar} \]
\[ \Rightarrow \quad \text{LR}(p) = \text{NewLocs}() + [0, 0] \]

\[ j : q = \phi(p_1, p_2) \quad \text{with } loc_j = \text{NewLocs}() \]
\[ \Rightarrow \quad \text{LR}(q) = loc_j + [0, 0] \]
Experimental setup

- Implementation: LLVM 3.5
- Benchmarks: LLVM test suite + Micro benchmarks + PtrDist + Prolangs + MallocBench
- Machine: Intel i7-4770K, 8GB of memory Ubuntu 14.04.2
Experimental results

<table>
<thead>
<tr>
<th></th>
<th>#Queries</th>
<th>scev</th>
<th>basic</th>
<th>rbaa</th>
<th>rbaa + basic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total %</strong></td>
<td>7,243,418</td>
<td>2.8</td>
<td>17.8</td>
<td>16.9</td>
<td>22.1</td>
</tr>
</tbody>
</table>

- **#Queries**: number of pair of pointers.
- **scev**: scalar evolution based alias-analysis.
- **basic**: -O3 LLVM analysis (global + local pointers).
- **rbaa**: range based alias analysis.

**Answering queries**: number of pairs that do not alias.

- The analysis is precise but still a room for improvement.
Experimental results

The analysis scales well!
Work in progress (with M. Maalej)

- Submitted to CGO’17: an adaptation of Pentagons for pointer disambiguation. (less-than constraints).
- A combination of 3 different analyses: journal paper in preparation.
Motivations and big picture

Technical context: LLVM

Symbolic range analysis

Application 1: GreenArrays toolsuite
  Overview
  A bit on the other analyses
  Experimental results
  Conclusion

Application 2: Symbolic Pointer Analysis
  Overview
  Experimental results

Conclusion
A Bit of Perspective

- **Non-Relational Analyses:**
  - They associate variables with information that do not depend on other variables. **Examples:** the usual data flow analyses.

- **Semi-Relational analyses:**
  - They associate variables with information that may contain other variable names. **Examples:** pentagons, **symbolic range analysis**, **symbolic region analysis**.

- **Relational analyses:**
  - They associate sets of variables with information. **Examples:** octagons and polyhedrons.
Take home message

- Code optimisation are good applications for static analyses!
- They have to be thought in terms of scaling as well as precision.
- Sparse analyses are the key but they still have to be invented/redesigned.