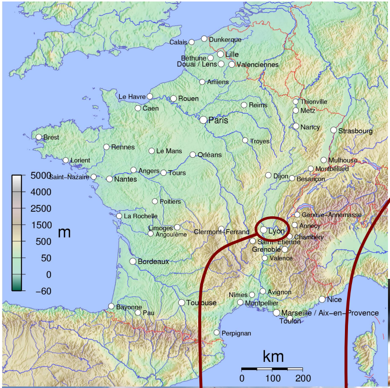
Experiences in designing scalable static analyses

LOPSTR/ WFLP invited talk

Laure Gonnord, University Lyon 1
& Laboratoire d'Informatique du Parallélisme
Lyon - France
http://laure.gonnord.org/pro

Sept 4th, 2018

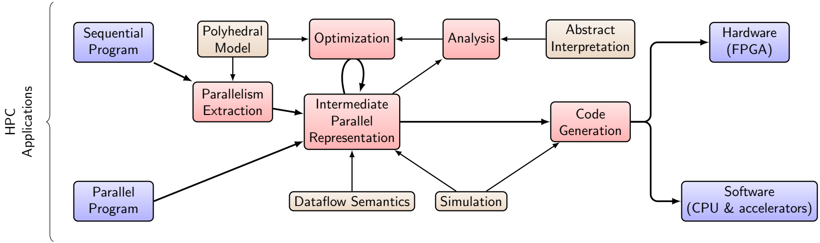
Compilation and Analysis for Software and Hardware - Location



LIP:
Laboratoire
de l'Informatique du
Parallélisme

CASH: Topics - People

Optimized (software/hardware) compilation for HPC software with data-intensive computations.
$\rightsquigarrow$ Means: dataflow IR, static analyses, optimisations, simulation.



Christophe Alias, Laure Gonnord, Matthieu Moy
`http://www.ens-lyon.fr/LIP/CASH/`

Outline

- Motivations
  - Static analyses, examples
  - Static analysis of software, how?
- Abstract Interpretation 101
- Abstract Interpretation for optimising compilers
  - Example 1: a scalable analysis for pointers
  - Example 2: array bound check elimination
  - Impact on compiler optimisation pathes
- Conclusion

Software needs safety and performance

- For safety-critical systems . . .
- **and** general purpose systems!

Software needs safety and performance

- For safety-critical systems . . .
- **and** general purpose systems!

▶ Programs crash because of array out-of-bounds accesses, complex pointer behaviour, . . .

Software guarantees, how?

- Development processes: coding rules, ...
- Testing: do not cover all cases.
- Proof assistants: expensive.

▶ **Static analysis of programs**.

Outline

Goal: safety 1/2

Prove that (some) memory accesses are safe:

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

- This program has an illegal array access.

Goal: safety 2/2

Prove program correctness/absence of functional bug:

```
void find_mini (int a[N], int l, int u){
  unsigned int i=l;
  int b=a[l]
  while (i <= u){
    if(a[i]<b) b=a[i] ;
    i++ ;
  }
  // here b = min(a[l..u])
}
```

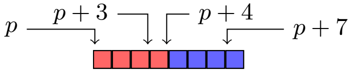
- This program finds the minimum of the sub-array.

Goal: performance 1/2

Enable loop parallelism:

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

Parallel loops



- The two regions do not overlap.

Goal: performance 2/2

Enable code motion:

```
void code_motion(int* p1, int *p2, int *p){
  // ...
  while(p2>p1){
    a = *p;
    *p2 = 4;
    p2 --;
  }
}
```

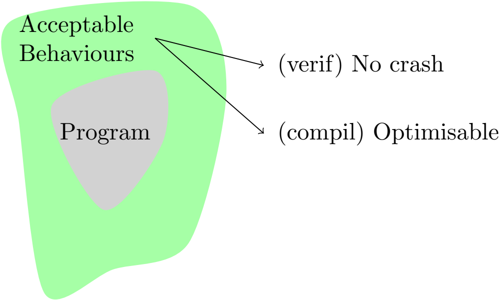
hoist!

- If p and p_2 do not alias, then `a=*p` is invariant.
- Hoisting this instruction saves one `load` per loop.

Outline

Proving non trivial properties of software

- Basic idea: software has **mathematically defined behaviour**.
- **Automatically** prove properties.



There is no free lunch

i.e. no magical static analyser. It is impossible to prove interesting properties:

- automatically
- exactly
- on unbounded programs

There is no free lunch

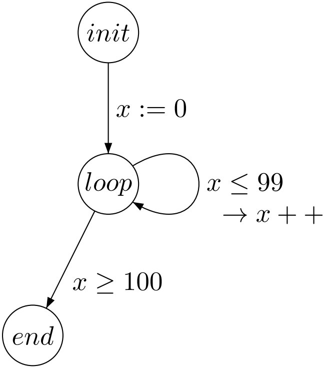
i.e. no magical static analyser. It is ~~im~~ possible to prove interesting properties:

- automatically
- ~~exactly~~ with false positives!
- on unbounded programs

▶ **Abstract Interpretation** = conservative approximations.

Outline

Computing (inductive) invariants



- $\{x \in \mathbb{N}, 0 \leq x \leq 100\}$ is the most precise invariant in control point `loop`.

Problems and solution

We want to:

- Compute infinite sets.
- In finite time.

▶ How?

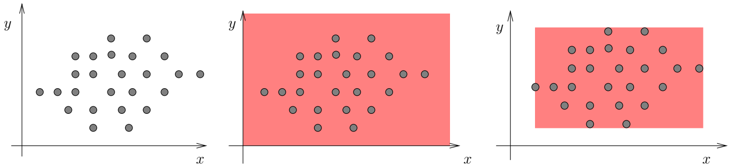
- Approximate sets (abstract domains), compute in this abstract world.
- Extrapolate (widening).

Main ingredient: abstract values

Idea: represent values of variables:

$$R_{pc} \in \mathcal{P}(\mathbb{N}^d)$$

by a **finite computable superset** $R^\sharp_{pc}$:



- And compute such **abstract values** for each control point.
- How? mimic the program operations

$$\mathbb{N}^d \times pcs \to \mathbb{N}^d \times pcs$$

by their abstract versions.

There is also this magical widening stuff, let's forget it in this talk

Example (Pagai, Verimag)

```
int main(int argc, char** argv){

  int x, y;
  x = 1;
  y = 2;
  /* reachable */
  /* invariant:
  3-2*y+x = 0
  5-y >= 0
  -2+y >= 0
  */
  while (x<8){
    x = x+2;
    y = y+1;
  }
  /* reachable */
  return 0;
}
```

see http://pagai.forge.imag.fr

Other famous AI tools

- Frama-C: “Evolved Value Analysis”.
- Astree: originally designed for safety critical C Compiled from Scade (synchronous programming).
- Polyspace (Mathworks).

Complexity in Abstract Interpretation

Classical abstract interpretation analyses:

- Information attached to (*block*, *variables*).
- A new information is computed after each statement.
- Abstract operations are sometimes costly.

▶ For the polyhedral abstract domain, the complexity is **3EXP**.

Challenges in Abstract Interpretation

- Precision of the abstract domain.
- Thousands, millions of lines of code to analyze.
- Static analyzers and compilers are complex programs (that also have bugs).

- Growing need for simple **specialized** analyses that **scale**

Credo: future of Abstract Intepretation

- Focus more on applicability, and less on expressivity.
- Scale and demonstrate that it scales.
- For general-purpose programs.

▶ and use techniques from other communities (optimization, model-checking, logic, rewriting)

Outline

Safe compilation?

- **Correct-by-construction** non-optimising compilers: Lustre, Scade.
- Translation validation: specialized proof of the generated code.
- Compcert.

▶ An evolution toward more trustable compilers. But what about code optimisation?

Motivation

Classical analyses (and optimisation) inside (production) compilers:

- Apart from classical dataflow algorithm, often **syntactic**.
- Usual abstract-interpretation based algorithms are too costly.
- Expressive algorithms: rely on “high level information”.

Motivation

Classical analyses (and optimisation) inside (production) compilers:

- Apart from classical dataflow algorithm, often **syntactic**.
- Usual abstract-interpretation based algorithms are too costly.
- Expressive algorithms: rely on “high level information”.

▶ Need for safe and precise quasi linear-time algorithms at **low-level**.

▶ Illustrations in the rest of the talk.

Some contributions

- Abstract domains/iteration strategies for numerical invariants [SAS11], [OOPSLA14].
- Applications to memory analysis [OOPSLA14], just in time compilers [WST14].
- Pointer analysis with “sparse” abstract interpretation [CGO16] [CGO17] [SCP17].

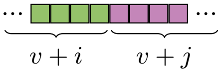
Collaborations with M. Maalej, F. Pereira and his team at UFMG, Brasil

Outline

Less than information for pointers [CGO17,SCP17]

```
void partition(int *v, int N) {
  int i, j, p, tmp;
  p = v[N/2];
  for (i = 0, j = N - 1;; i++, j--) {
    while (v[i] < p) i++;
    while (p < v[j]) j--;
    if (i >= j)
      break;
    tmp = v[i];
    v[i] = v[j];
    v[j] = tmp;
  }
}
```

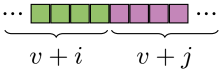
`v[i] = *(v+i)`



Less than information for pointers [CGO17,SCP17]

```
void partition(int *v, int N) {
  int i, j, p, tmp;
  p = v[N/2];
  for (i = 0, j = N - 1;; i++, j--) {
    while (v[i] < p) i++;
    while (p < v[j]) j--;
    if (i >= j)
      break;
    tmp = v[i];
    v[i] = v[j];
    v[j] = tmp;
  }
}
```

`v[i] = *(v+i)`



- Range information is not sufficient to disambiguate `v[i]` and `v[j]`.
- We need to propagate **relational information**.

Our setting for scaling analyses

Classical abstract interpretation analyses:

- Information attached to (*block*, *variable*).
- A new information is computed after each statement.

Sparse analyses $\Rightarrow$ **Static Single Information (SSI) Property [Ana99]**:

- Attach information to variables.
- The information must be invariant throughout the live range of the variable.

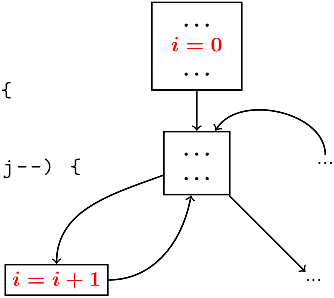
▶ A simple assignment breaks SSI!

▶ Work on suitable intermediate representations

Scaling analyses: program representation 1/2

Static Single Assignment (**SSA**) form: each variable is defined/assigned once.

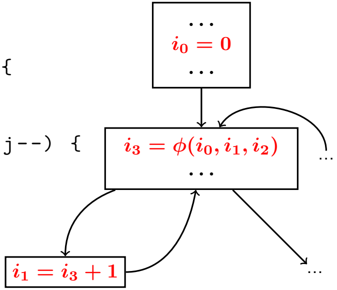
```
void partition(int *v, int N) {
  int i, j, p, tmp;
  p = v[N/2];
  for (i = 0, j = N - 1;; i++, j--) {
    while (v[i] < p) i++;
    ...
}
```



Scaling analyses: program representation 1/2

Static Single Assignment (**SSA**) form: each variable is defined/assigned once.

```
void partition(int *v, int N) {
  int i, j, p, tmp;
  p = v[N/2];
  for (i = 0, j = N - 1;; i++, j--) {
    while (v[i] < p) i++;
    ...
}
```

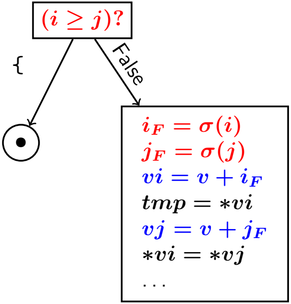


- Sparse storage of **value** information (one value range per variable name).

Scaling analyses: program representation 2/2

Within **SSA** form, tests information cannot be propagated!

```
void partition(int *v, int N) {
  ...
    if (i >= j)
        break;
    tmp = v[i];
    v[i] = v[j];
}
```



- $i \geq j$ is invariant nowhere.
- The σ renaming (**e-SSA**) enables to propagate "$i_F < j_F$".

Scaling analyses: relational information

Recall the SSI setting:

- Information must be invariant throughout the live range of the variable. ✓
- Attach information to variables (and not blocks).

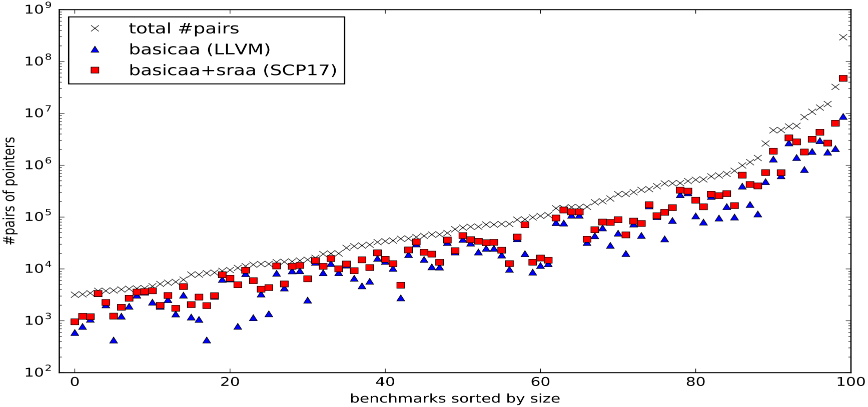
- Work on semi-relational domains, for instance:
  - Parametric ranges [OOPSLA14] $x \mapsto [0, N + 1]$
  - Pentagons [LF10]: $x \mapsto \{u, t\}$ means $u, t \leq x$.

Contributions on static analyses for pointers

(with Maroua Maalej) [CGO16, CGO17, SCP17]

- A new sequence of static analyses for pointers.
- Based on semi-relational sparse abstract domains:
    - In CGO'16: $p \mapsto loc + [a, b]$.
    - In CGO'17: adaptation of Pentagons.
- Implemented in LLVM.
- Used as oracles for a common pass called `AliasAnalysis`.
- Experimental evaluation on classical benchmarks.

Experimental results [SCP17]



- Comparison with LLVM basic alias analysis.
- Our sraa outperforms basicaa in the majority of the tests.
- The combination outperforms each of these analyses separately in every one of the 100 programs.

Outline

Contribution [OOPSLA'14]

- 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

A bit on sanitizing memory accesses

Different techniques : but all have an overhead.

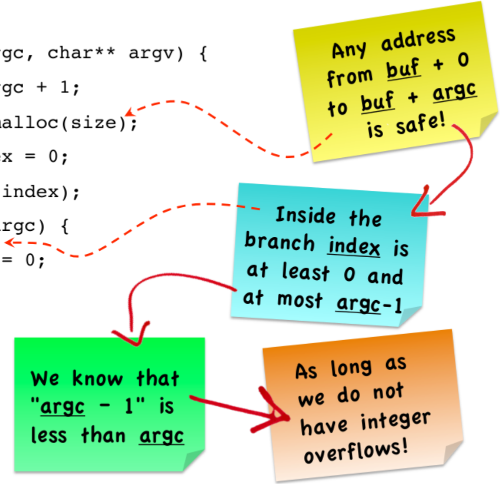
Ex : Address Sanitizer

- Shadow every memory allocated : 1 byte $\rightarrow$ 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**.

Green Arrays: a set of sparse analyses 1/2

```
int main(int argc, char** argv) {
  int size = argc + 1;
  char* buf = malloc(size);
  unsigned index = 0;
  scanf("%u", &index);
  if (index < argc) {
    buf[index] = 0;
  }
  return index;
}
```

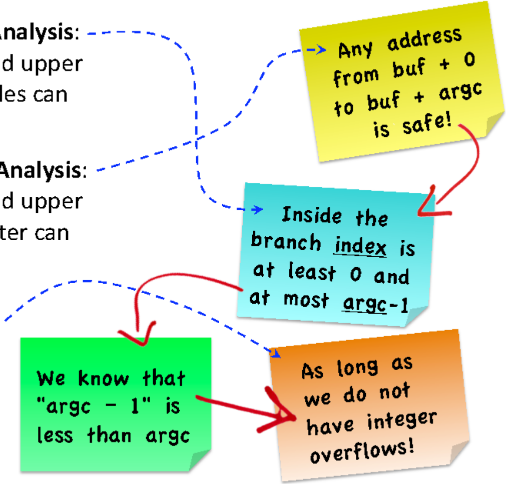


Green Arrays: a set of sparse analyses 2/2

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?



Experimental setup

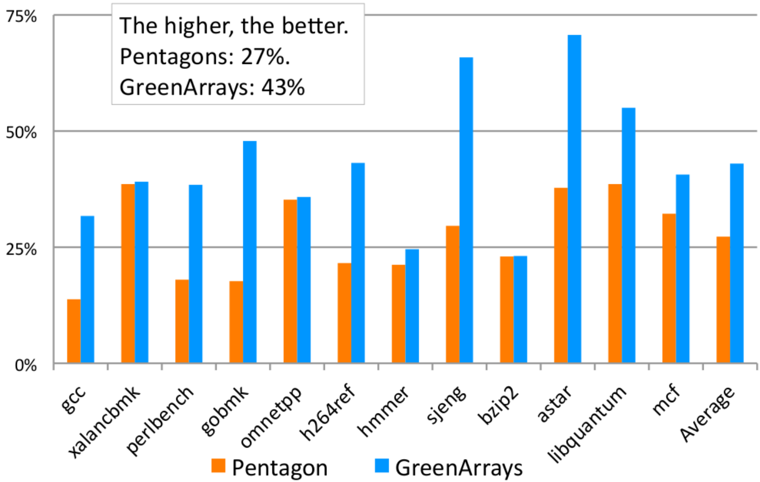
- **Implementation**: LLVM + AddressSanitizer
- **Benchmarks**: SPEC CPU 2006 + LLVM test suite
- **Machine**: Intel(R) Xeon(R) 2.00GHz,with 15,360KB of cache and 16GB or RAM
- **Baseline**: Pentagons
  - Abstract interpretation that combines "less-than" and "integer ranges".†

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

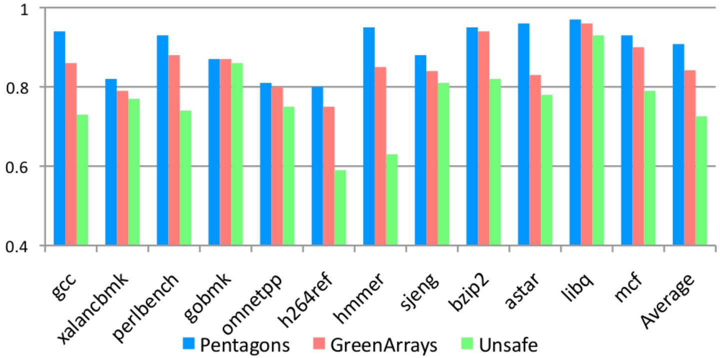
P(j) = (less than {i}, [0, 8])

†: Pentagons: A weakly relational abstract domain for the efficient validation of array accesses, 2010, Science of Computer Programming

Percentage of bound checks removed



Runtime improvement



The lower the bar, the faster. Time is normalized to AddressSanitizer without bound-check elimination. Average speedup: Pentagons = 9%. GreenArrays = 16%.

Outline

Some comments on the methodology

LLVM compiler:

- comes with a test infrastructure and benchmarks.
- analysis and optimisation passes log information.
- you can add your own pass, but **where?**

```
clang -c -emit-llvm $1 -o $name.bc
opt -mem2reg -instnamer $name.bc -o $name.rbc
sage-opt -load $lib_path/$ssify_so -break-crit-edges -ssify -set 1000  $name.rbc -o $name.rbc
sage-opt -stats -load $lib_path/$python_so -load $lib_path/$sage_so -load $lib_path/$sra_so -load $lib_path/$rbaa_so -lo
ad $lib_path/$licm2_so -range-based-aa -aa-eval -no-aa -tbaa -targetlibinfo -basicaa -notti -verify -simplifycfg -domtre
e -sroa -lower-expect -targetlibinfo -no-aa -tbaa -basicaa -notti -ipsccp -globalopt -deadargelim -domtree -instcombine
-simplifycfg -basiccg -prune-eh -inline-cost -inline -functionattrs -argpromotion -sroa -domtree -lazy-value-info -jump-
threading -correlated-propagation -simplifycfg -domtree -instcombine -tailcallelim -simplifycfg -reassociate -domtree -l
oops -loop-simplify -lcssa -loop-rotate -licm2 -loop-unswitch -instcombine -scalar-evolution -loop-simplify -lcssa -indv
ars -loop-idiom -loop-deletion -loop-unroll -memdep -mldst-motion -domtree -memdep -range-based-aa -gvn -view-cfg -memde
p -memcpyopt -sccp -domtree -instcombine -lazy-value-info -jump-threading -correlated-propagation -domtree -memdep -dse
-range-based-aa -adce -simplifycfg -domtree -instcombine -barrier -domtree -loops -loop-simplify -lcssa -branch-prob -bl
ock-freq -scalar-evolution -loop-vectorize -instcombine -scalar-evolution -slp-vectorizer -simplifycfg -domtree -instcom
bine -loops -loop-simplify -lcssa -scalar-evolution -loop-unroll -strip-dead-prototypes -globaldce -constmerge -verify $
name.rbc -o $name.opt.rbc
```

- Evaluating the impact of a given analysis is a **nightmare**!

Impact on LLVM code motion 1/2

Loop invariant code motion (LICM):

```
void code_motion(int* p1, int *p2, int *p){
  // ...
  while(p2>p1){
    a = *p;      // hoist!
    *p2 = 4;
    p2 --;
  }
}
```

- If p and p_2 do not alias, then a=*p is invariant.

Impact of our analyses (excerpt) 2/2

Program	#Inst	#moved	
		O3	O3+our analysis (CGO16)
fixoutput	369	1	**5**
compiler	3515	0	0
bison	15645	165	**179**
archie-client	5939	0	0
TimberWolfMC	98792	1287	**1447**
allroots	574	0	0
unix-smail	5435	3	3
plot2fig	3217	3	3
bc	10632	18	**19**
yacr2	6583	144	**190**
ks	1368	8	**11**
cfrac	7353	5	**6**
espresso	50751	301	**398**
gs	55281	20	X

More in Maroua Maalej's thesis.

Outline

Summary

Static analyses for compilers:

- Application domain: code optimisation.
- Adaptation of abstract interpretation algorithms inside this particular context.
- Algorithmic and compilation techniques to scale.
- Future work: more relational domains (and data structures).

Take home message

- Code optimisation are good applications for static analyses/formal methods!
- 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.

References I

[ADFG10] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord.
Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs.
In Proceedings of the 17th International Static Analysis Symposium (SAS'10), Perpignan, France, September 2010.

[ADFG13] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord.
Rank: a tool to check program termination and computational complexity.
In Workshop on Constraints in Software Testing Verification and Analysis (CSTVA'13), Luxembourg, March 2013.

[Ana99] Scott Ananian.
The static single information form.
Master's thesis, MIT, September 1999.

[Fea92] Paul Feautrier.
Some efficient solutions to the affine scheduling problem. part ii. multidimensional time.
International Journal of Parallel Programming, 21(6):389–420, Dec 1992.

[GMR15] Laure Gonnord, David Monniaux, and Gabriel Radanne.
Synthesis of ranking functions using extremal counterexamples.
In Proceedings of the 2015 ACM International Conference on Programming Languages, Design and Implementation (PLDI'15), Portland, Oregon, United States, June 2015.

[LF10] Francesco Logozzo and Manuel Fähndrich.
Pentagons: A weakly relational abstract domain for the efficient validation of array accesses.
Sci. Comput. Program., 75(9):796–807, 2010.

[MPMQPG17] Maroua Maalej, Vitor Paisante, Fernando Magno Quintao Pereira, and Laure Gonnord.
Combining Range and Inequality Information for Pointer Disambiguation.
Science of Computer Programming, 2017.
Accepted in Oct 2017, final published version of https://hal.inria.fr/hal-01429777v2.

References II

[MPR+17] Maroua Maalej, Vitor Paisante, Pedro Ramos, Laure Gonnord, and Fernando Pereira.
Pointer Disambiguation via Strict Inequalities.
In International Symposium of Code Generation and Optimization (CGO'17), Austin, United States, February 2017.

[PMB+16] Vitor Paisante, Maroua Maalej, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintao Pereira.
Symbolic Range Analysis of Pointers.
In International Symposium of Code Generation and Optimization (CGO'16), pages 791–809, Barcelon, Spain, March 2016.

[SMO+14] Henrique Nazaré Willer Santos, Izabella Maffra, Leonardo Oliveira, Fernando Pereira, and Laure Gonnord.
Validation of Memory Accesses Through Symbolic Analyses.
In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages And Applications (OOPSLA'14), Portland, Oregon, United States, October 2014.