Program verification

Scaling analyses: 2 case studies

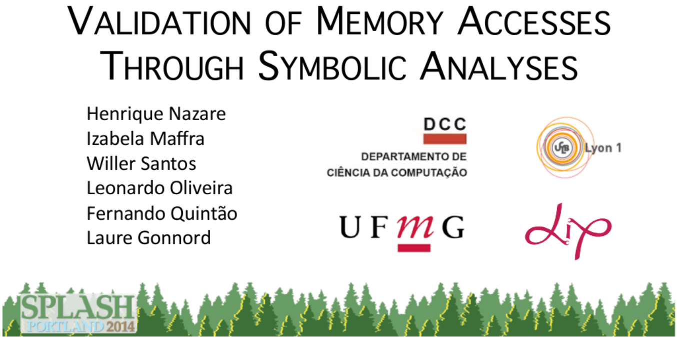
Laure Gonnord and David Monniaux

University of Lyon / LIP

November 17, 2015

Plan

Inspiration : OOPSLA'14 slides:



VALIDATION OF MEMORY ACCESSES THROUGH SYMBOLIC ANALYSES

Henrique Nazare
Izabela Maffra
Willer Santos
Leonardo Oliveira
Fernando Quintão
Laure Gonnord

DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

Lyon 1

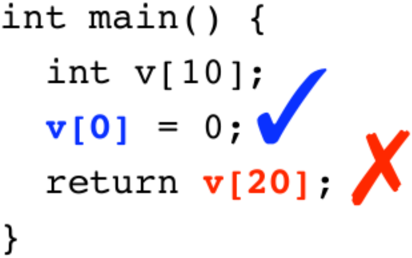
UFMG

SPLASH
PORTLAND 2014

`http://homepages.dcc.ufmg.br/~fernando/publications/presentations/OOPSLA14.pdf`

Goal : Safety

Prove that (some) memory accesses are safe:



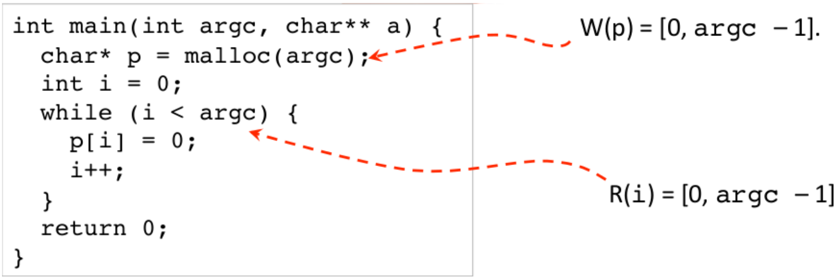
```
int main() {
   int v[10];
   v[0] = 0;
   return v[20];
}
```

- Fight against bugs and overflow attacks.

Contributions (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

Our key insight : Symbolic (parametric) Analyses



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

$W(p) = [0, \texttt{argc} - 1]$.

$R(i) = [0, \texttt{argc} - 1]$

- $R(i) \subseteq W(p)$ 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**.

Green Arrays : overview 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;
}
```

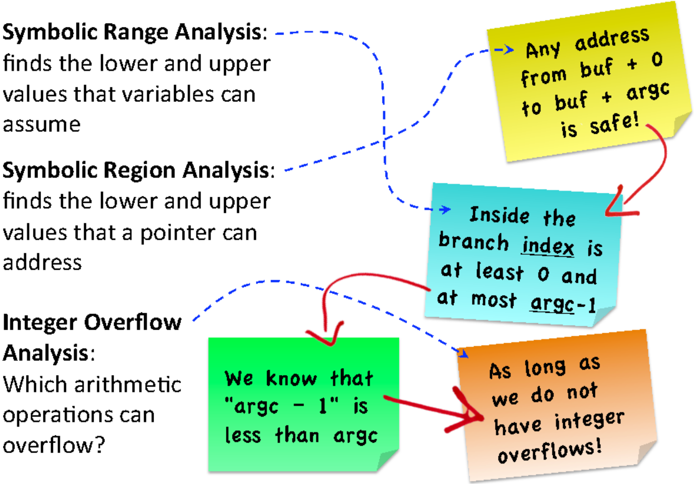
Any address from buf + 0 to buf + argc is safe!

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!

Green Arrays : overview 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?

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!

A bit on LLVM

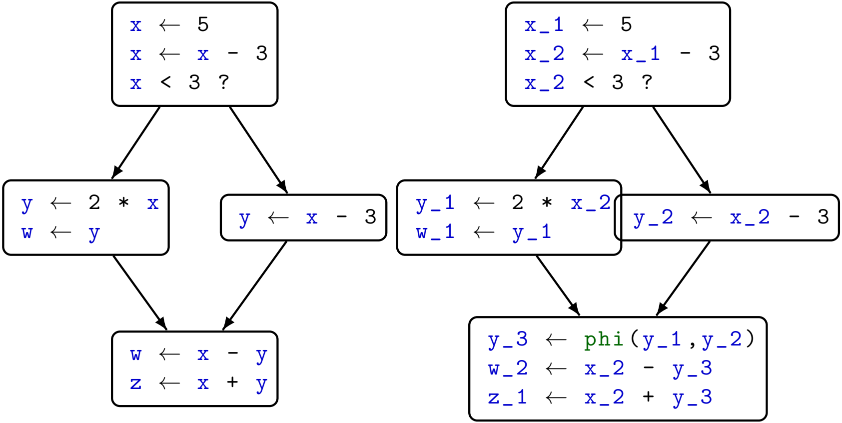
LLVM is a **compiler infrastructure**:

- 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



```
x ← 5
x ← x - 3
x < 3 ?
```

```
y ← 2 * x
w ← y
```

```
y ← x - 3
```

```
w ← x - y
z ← x + y
```

```
x_1 ← 5
x_2 ← x_1 - 3
x_2 < 3 ?
```

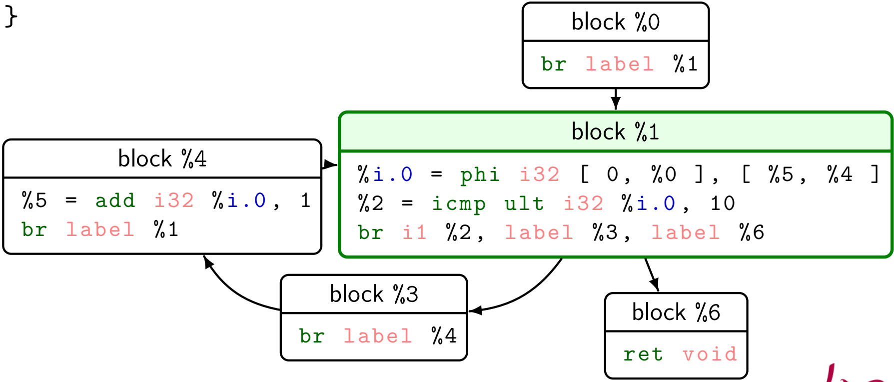
```
y_1 ← 2 * x_2
w_1 ← y_1
```

```
y_2 ← x_2 - 3
```

```
y_3 ← phi(y_1,y_2)
w_2 ← x_2 - y_3
z_1 ← x_2 + y_3
```

LLVM representation: a toy example

```
void simple_loop_constant() {
  for(unsigned i=0; i<10; i++) {
    // Do nothing
  }
}
```



block %0

```
br label %1
```

block %1

```
%i.0 = phi i32 [ 0, %0 ], [ %5, %4 ]
%2 = icmp ult i32 %i.0, 10
br i1 %2, label %3, label %6
```

block %4

```
%5 = add i32 %i.0, 1
br label %1
```

block %3

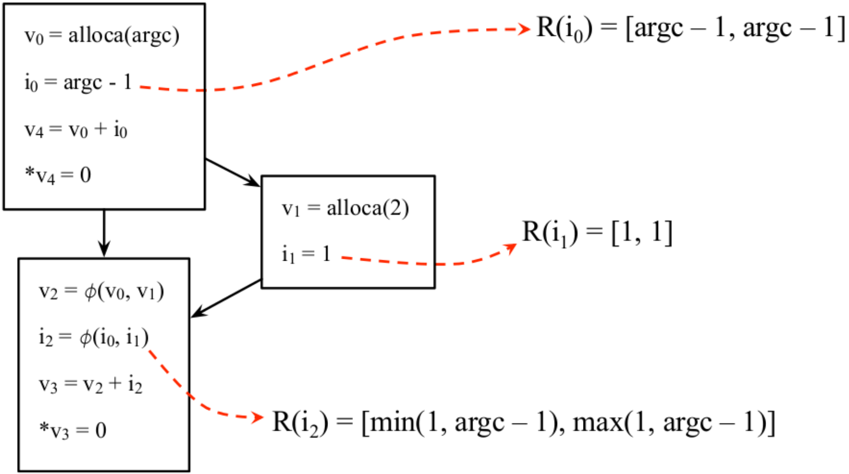
```
br label %4
```

block %6

```
ret void
```

Credits G. Radanne

Symbolic Ranges (SRA): Goal



$v_0 = \text{alloca}(\text{argc})$

$i_0 = \text{argc} - 1$

$v_4 = v_0 + i_0$

$*v_4 = 0$

$v_1 = \text{alloca}(2)$

$i_1 = 1$

$v_2 = \phi(v_0, v_1)$

$i_2 = \phi(i_0, i_1)$

$v_3 = v_2 + i_2$

$*v_3 = 0$

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

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

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

SRA on SSA form: a sparse analysis

- An abtract interpretation-based technique.
- Very similar to classic range analysis.
- One abstract value (R) **per variable**: sparsity.

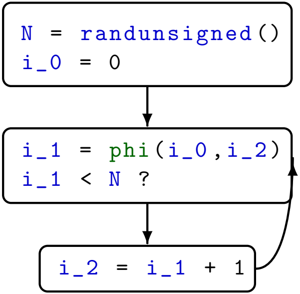
- Easy to implement (simple algorithm, simple data structure).

SRA on SSA form: constraint system

$$\begin{aligned}
v = \bullet \quad &\Rightarrow \quad R(v) = [v, v] \\
v = o \quad &\Rightarrow \quad R(v) = R(o) \\
v = v_1 \oplus v_2 \quad &\Rightarrow \quad R(v) = R(v_1) \oplus^I R(v_2) \\
v = \phi(v_1, v_2) \quad &\Rightarrow \quad R(v) = R(v_1) \sqcup R(v_2) \\
\text{other instructions} \quad &\Rightarrow \quad \emptyset
\end{aligned}$$

$\oplus^I$: abstract effect of the operation $\oplus$ on two intervals.
$\sqcup$: convex hull of two intervals. ▶ All these operation are performed symbolically thanks to **GiNaC**

SRA on SSA form: an example



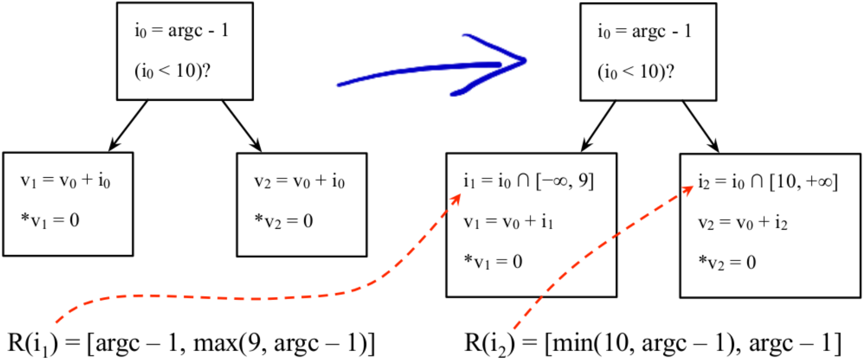
```
N = randunsigned()
i_0 = 0
```

```
i_1 = phi(i_0,i_2)
i_1 < N ?
```

```
i_2 = i_1 + 1
```

- $R(i_0) = [0, 0]$
- $R(i_1) = [0, +\infty]$
- $R(i_2) = [1, +\infty]$

Improving precision of SRA : live-range splitting 1/2



$i_0 = \text{argc} - 1$

$(i_0 < 10)?$

$v_1 = v_0 + i_0$

$*v_1 = 0$

$v_2 = v_0 + i_0$

$*v_2 = 0$

$i_0 = \text{argc} - 1$

$(i_0 < 10)?$

$i_1 = i_0 \cap [-\infty, 9]$

$v_1 = v_0 + i_1$

$*v_1 = 0$

$i_2 = i_0 \cap [10, +\infty]$

$v_2 = v_0 + i_2$

$*v_2 = 0$

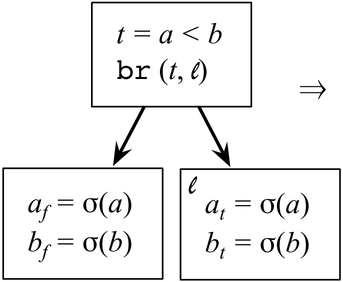
$R(i_1) = [\text{argc} - 1, \max(9, \text{argc} - 1)]$

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

- e-SSA form.

Improving precision of SRA : live-range splitting 2/2

Rule for live-range splitting :



$t = a < b$

`br` (t, ℓ)

$a_f = \sigma(a)$

$b_f = \sigma(b)$

ℓ

$a_t = \sigma(a)$

$b_t = \sigma(b)$

$$\Rightarrow$$

$$R(a_t) = [R(a)_\downarrow, \min(R(b)_\uparrow - 1, R(a)_\uparrow)]$$

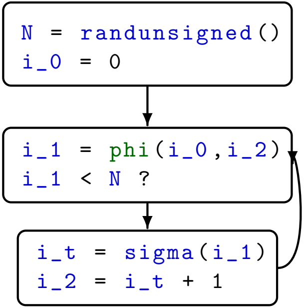
$$R(b_t) = [\max(R(a)_\downarrow + 1, R(a)_\downarrow), R(b)_\uparrow]$$

$$R(a_f) = [\max(R(a)_\downarrow, R(a)_\uparrow), R(a)_\uparrow]$$

$$R(b_t) = [R(b)_\downarrow, \min(R(a)_\uparrow, R(b)_\uparrow)]$$

- All simplications are done by GiNaC.

SRA + live-range on an example



```
N = randunsigned()
i_0 = 0
```

```
i_1 = phi(i_0,i_2)
i_1 < N ?
```

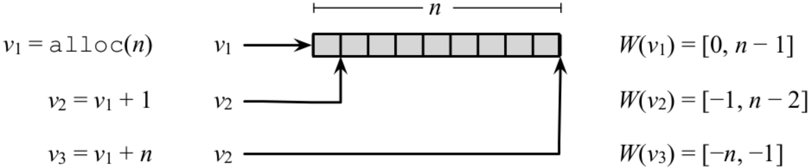
```
i_t = sigma(i_1)
i_2 = i_t + 1
```

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

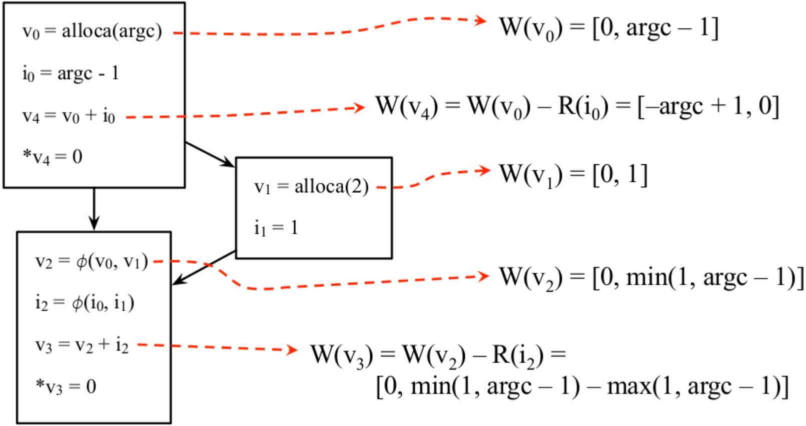
- $R(i_0) = [0, 0]$
- $R(i_1) = [0, N]$

Symbolic regions 1/2: Goal

Compute (an underapproximation of) the range of **valid accesses** from base pointers:



Symbolic regions 2/2: An example



$v_0 = \text{alloca}(\text{argc})$

$i_0 = \text{argc} - 1$

$v_4 = v_0 + i_0$

$*v_4 = 0$

$v_1 = \text{alloca}(2)$

$i_1 = 1$

$v_2 = \phi(v_0, v_1)$

$i_2 = \phi(i_0, i_1)$

$v_3 = v_2 + i_2$

$*v_3 = 0$

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

$W(v_4) = W(v_0) - R(i_0) = [-\text{argc} + 1, 0]$

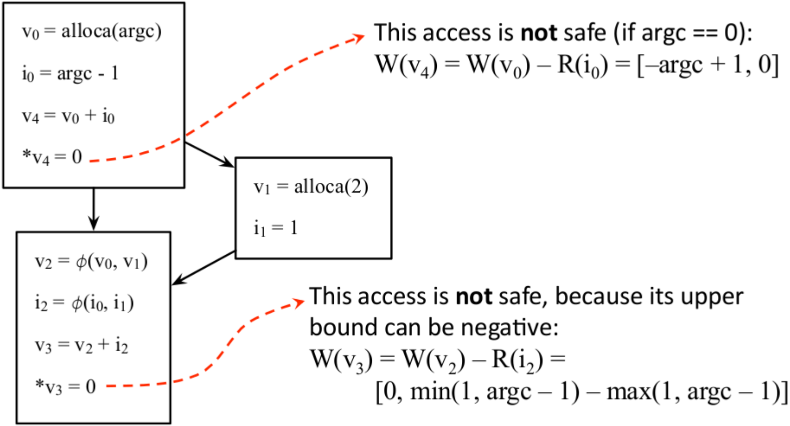
$W(v_1) = [0, 1]$

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

$W(v_3) = W(v_2) - R(i_2) =$
$[0, \min(1, \text{argc} - 1) - \max(1, \text{argc} - 1)]$

Safety: result

If $0 \in W(p)$, then $*p$ is **safe**, else **DK**



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

This access is **not** safe, because its upper bound can be negative:
$W(v_3) = W(v_2) - R(i_2) =$
$[0, \min(1, \text{argc} - 1) - \max(1, \text{argc} - 1)]$

Overflows 1/2

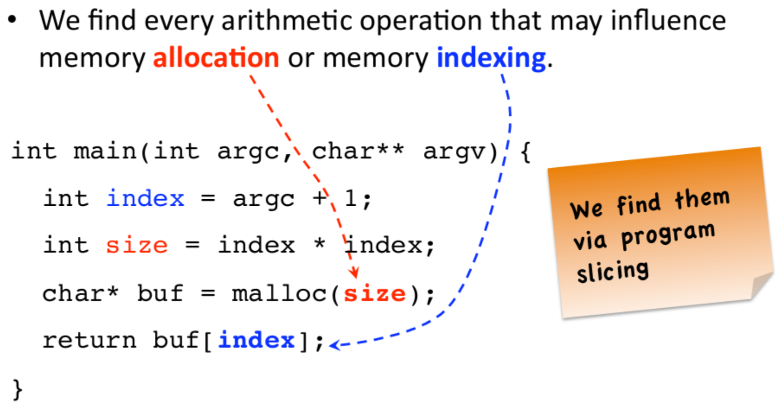
```
int main(int argc, char** argv) {
  int index = argc + 1;
  int size = index * index;
  char* buf = malloc(size);
  return buf[index];
}
```

index * index may wrap around.

Do you know what malloc will return?

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**.

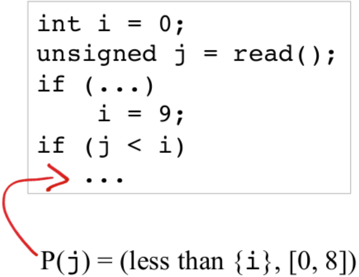
```
int main(int argc, char** argv) {
   int index = argc + 1;
   int size = index * index;
   char* buf = malloc(size);
   return buf[index];
}
```

We find them via program slicing

- 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".†

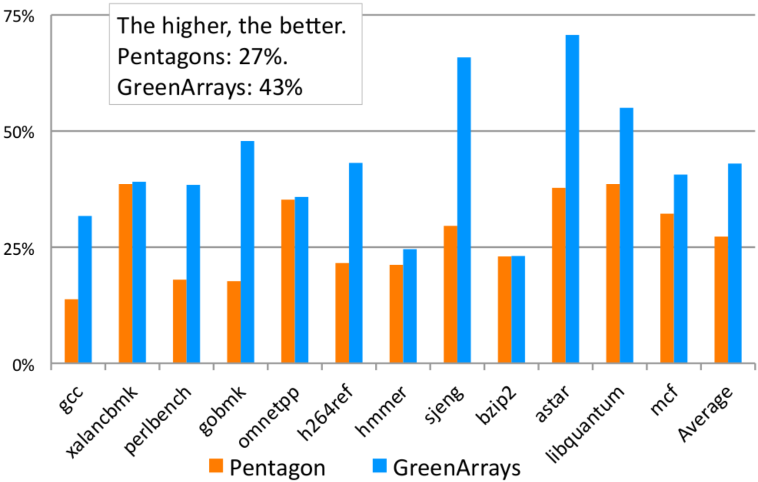


```
int i = 0;
unsigned j = read();
if (...)
    i = 9;
if (j < i)
    ...
```

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

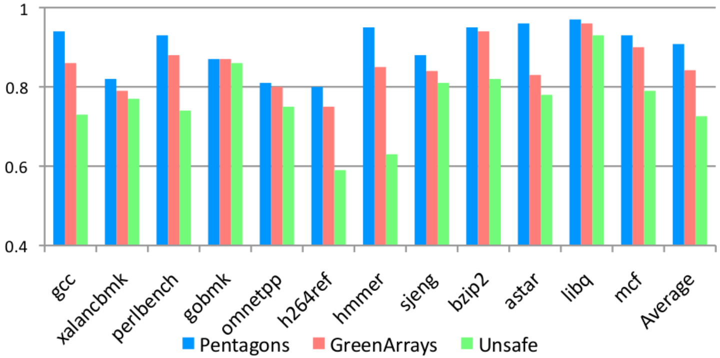
†: Pentagons: A weakly relational abstract domain for the efficient validation of array accesses, 2010, Science of Computer Programming

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

Symbolic range analysis

- Motivation and big picture
- Overview
- Technical context LLVM
- Symbolic Range Analysis
- A bit on the other analyses
- Experimental results
- Conclusion

Symbolic pointer analysis

- Range Analysis
- Pointer Range Analysis
- Experimental results

Static Single Information form (SSI)

Credit : M. Maalej. Accepted to CGO'16.

Goal + Contribution

Goal:

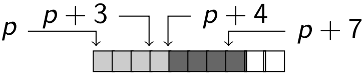
- Optimizing languages with pointers;
- Solving pointer arithmetic, disambiguating pointers;
- Low cost analysis.

Contribution:

- Combine alias analysis with range analysis;
- Speed up;

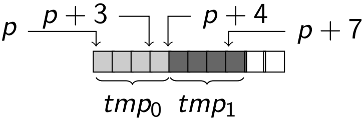
Motivating example

```
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 ;
}
```

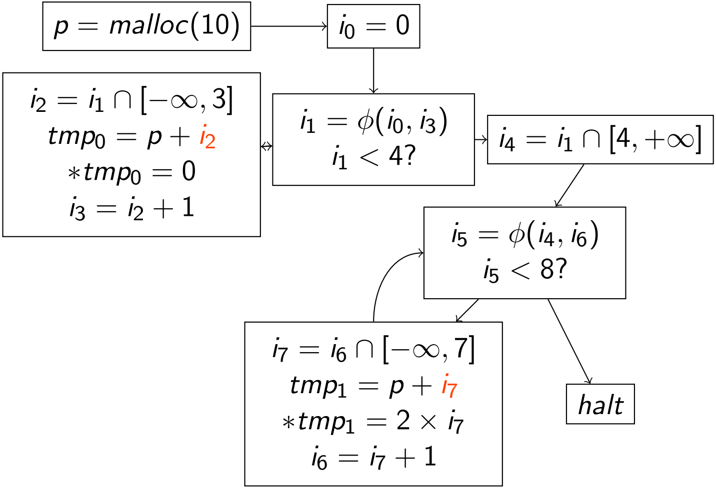


Motivating example

```
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
}
```



Range Analysis on e-SSA (recall)



$p = malloc(10)$

$i_0 = 0$

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

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

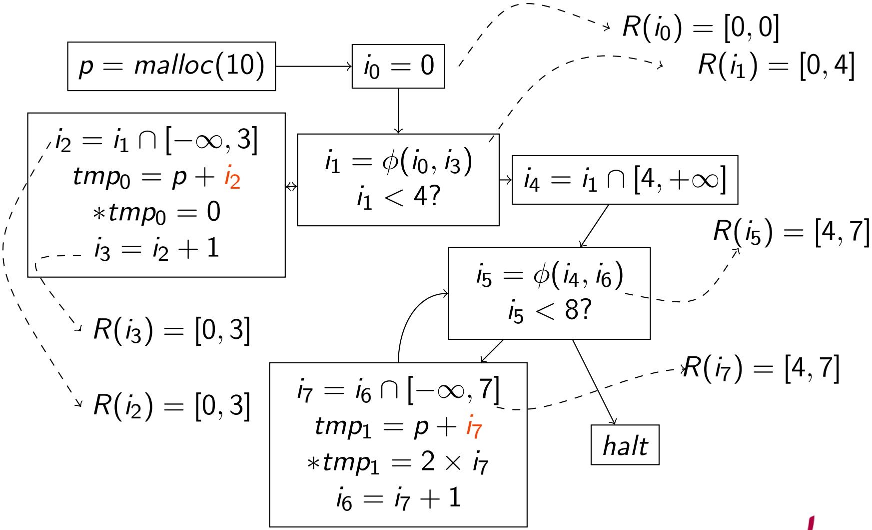
$i_4 = i_1 \cap [4, +\infty]$

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

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

$halt$

Range Analysis on e-SSA (recall)



$p = malloc(10)$

$i_0 = 0$

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

$R(i_1) = [0, 4]$

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

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

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

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

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

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

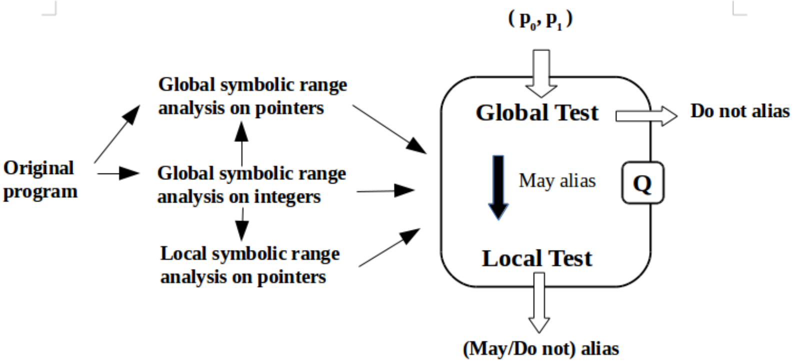
$R(i_2) = [0, 3]$

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

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

halt

Pointer Range Analysis



Abstract Analysis

$$\begin{array}{ll} p_i \rightsquigarrow loc_i + [l_i, u_i] & \multirow{2}{*}{} \\ p_j \rightsquigarrow loc_j + [l_j, u_j] & \end{array} \qquad \mathrm{i} = \mathrm{j} \text{ or } \mathrm{i} \neq \mathrm{i}$$

Property (overapprox):
If ($loc_i = loc_j$ and $[l_i, u_i] \sqcap [l_j, u_j] = \emptyset$)
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 $\uplus \perp)^n$: GR(p) = $(p_0, \ldots, p_{n-1})$.

Notation

- GR(p) = $\{loc_i + p_i, loc_j + p_j, \ldots\}$

Constraint System:

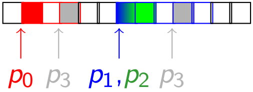
$$j : p = \text{ malloc } (v) \quad \Rightarrow \quad GR(p) = (\perp, \ldots, [0,0]_j, \ldots)$$

$$v = v_1 \quad \Rightarrow \quad GR(v) = GR(v_1)$$

$$\begin{array}{r} q = p + c \\ \text{with } c \text{ scalar} \end{array} \quad \Rightarrow \quad q_i = \begin{cases} \perp \text{ if } p_i = \perp \\ p_i + R(c) \text{ else} \end{cases}$$

$$q = \phi(p^1, p^2) \quad \Rightarrow \quad \text{GR}(q) = \text{GR}(p^1) \sqcup \text{GR}(p^2)$$

Global Pointer Range Analysis



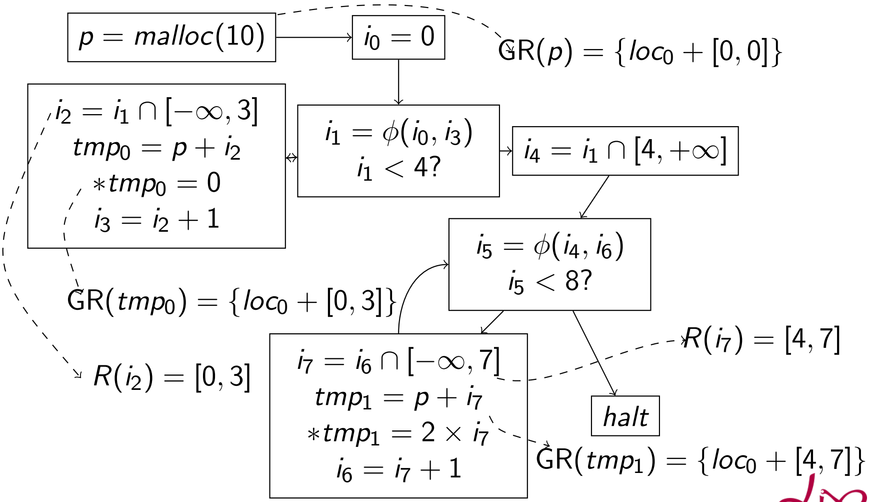
$GR(p_0) = loc_0 + [0, 0]$
$GR(p_1) = loc_1 + [0, 0]$
$GR(p_2) = loc_1 + [0, 1]$
$GR(p_3) = \{loc_0 + [2, 2], loc_1 + [3, 3]\}$

$p_0 = \text{malloc}(3)$;
$p_1 = \text{malloc}(5)$;

if (...) $p_2 = p_1$;
else $p_2 = p_1 + 1$;
if (...) $p_3 = p_0 + 2$;
else $p_3 = p_1 + 3$

Global Pointer Range Analysis

Example:



$p = malloc(10)$

$i_0 = 0$

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

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

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

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

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

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

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

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

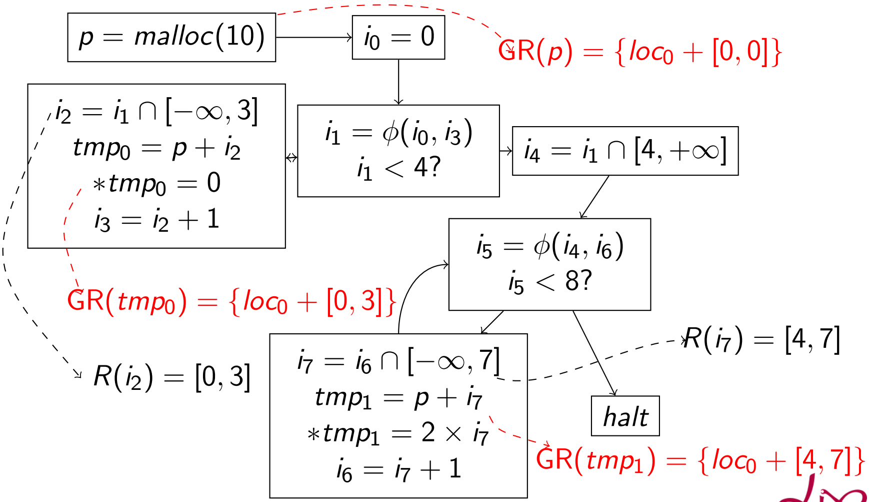
$i_7 = i_6 \cap [-\infty, 7]$
$tmp_1 = p + i_7$
$*tmp_1 = 2 \times i_7$
$i_6 = i_7 + 1$

$halt$

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

Global Pointer Range Analysis

Example:



$p = malloc(10)$

$i_0 = 0$

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

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

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

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

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

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

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

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

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

halt

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

Local Pointer Range Analysis

Motivation:

a_0 = malloc (N) ; $\rightsquigarrow$ GR (a_0) = {loc_0 + [0, 0]}
if (...)
  $a_1 = a_0 + 1$; $\rightsquigarrow$ GR (a_1) = {loc_0 + [1, 1]}
  $a_2 = a_1$; $\rightsquigarrow$ GR (a_2) = {loc_0 + [1, 1]}
else $a_2 = a_0$; $\rightsquigarrow$ GR (a_2) = {loc_0 + [0, 1]}

$a_3 = a_2 + 1$; $\rightsquigarrow$ GR (a_3) = {loc_0 + [1, 2]}

$a_4 = a_2 + 2$; $\rightsquigarrow$ GR (a_4) = {loc_0 + [2, 3]}

$$[1, 2] \cup [2, 3] \neq \emptyset$$

Local Pointer Range Analysis

Motivation:

a_0 = malloc (N) ; $\rightsquigarrow$ GR $(a_0) = \{loc_0 + [0, 0]\}$
if (...)
  $a_1 = a_0 + 1$; $\rightsquigarrow$ GR $(a_1) = \{loc_0 + [1, 1]\}$
  $a_2 = a_1$; $\rightsquigarrow$ GR $(a_2) = \{loc_0 + [1, 1]\}$
else $a_2 = a_0$; $\rightsquigarrow$ GR $(a_2) = \{loc_0 + [0, 1]\}$ $\rightsquigarrow$ LR $(a_2) = \{loc_1 + [0, 0]\}$
$a_3 = a_2 + 1$; $\rightsquigarrow$ GR $(a_3) = \{loc_0 + [1, 2]\}$ $\rightsquigarrow$ LR $(a_3) = \{loc_1 + [1, 1]\}$
$a_4 = a_2 + 2$; $\rightsquigarrow$ GR $(a_4) = \{loc_0 + [2, 3]\}$ $\rightsquigarrow$ LR $(a_4) = \{loc_1 + [2, 2]\}$

$$[1, 2] \cup [2, 3] \neq \emptyset$$

Local Pointer Range Analysis

Constraint System:

$$\begin{array}{r} p = \text{ malloc } (v) \\ \text{with } v \text{ scalar} \end{array} \quad \Rightarrow \quad \text{LR}(p) = \mathcal{N}ewLocs() + [0,0]$$

$$\begin{array}{r} j : q = \phi(p_1, p_2) \\ \text{with } loc_j = \mathcal{N}ewLocs() \end{array} \quad \Rightarrow \quad \text{LR}(q) = loc_j + [0,0]$$

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

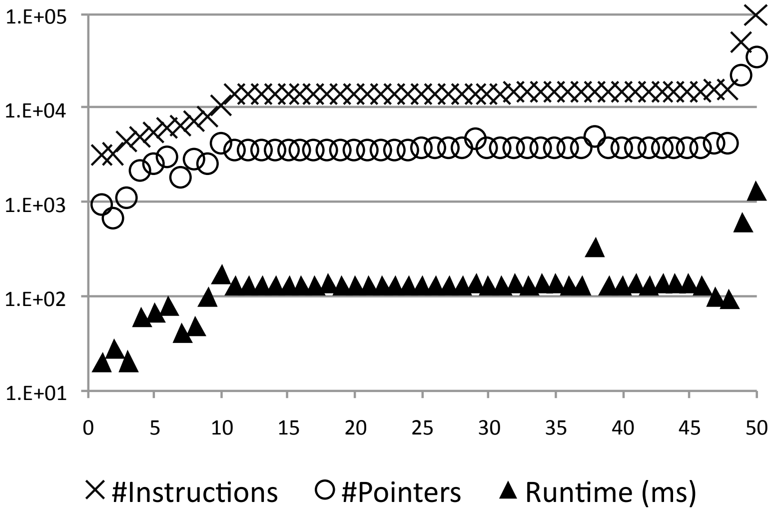
Experimental results

	#Queries	**scev**	**basic**	**rbaa**	**rbaa + basic**
Total %	7,243,418	2.8	17.8	16.9	22.1

- #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.

Experimental results



Conclusion

This analysis scales well!

Plan

- Symbolic range analysis
  - Motivation and big picture
  - Overview
  - Technical context LLVM
  - Symbolic Range Analysis
  - A bit on the other analyses
  - Experimental results
  - Conclusion
- Symbolic pointer analysis
  - Range Analysis
  - Pointer Range Analysis
  - Experimental results
- Static Single Information form (SSI)

Credit : F. Peirera and Fabrice Rastello. (Acaces 2015)

Goal + Contribution

Goal:

- Static Analyses that scale.
- Static but precise.

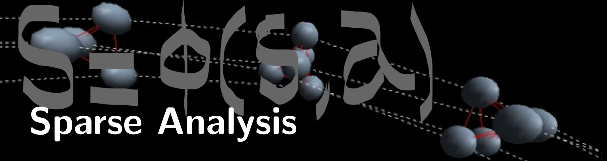
Contribution:

- A generic framework.
- A general way to solve the problem.

Introduction

- Data-flow analysis: discover facts (information) that are true about a program. Bind to *Variables* × *PogramPoints*.
- Static Single Information (SSI) property: IR such that information of a variable invariant along its whole live-range
- ϕ-functions split live-ranges where reaching definitions collide: SSA fulfills SSI property for constant analysis. Not for class inference (backward from uses).
- Extended SSA: SSI property for forward analysis flowing from definitions and conditional tests.
- SSU: SSI property for backward analysis flowing from uses

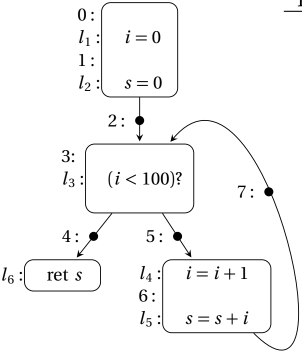
Can we generalize?



Sparse Analysis

Non-relational (dense) analysis: bind information to pairs *Variables* × *ProgPoints*

```
i = 0;
s = 0;
while (i < 100)
   i = i + 1;
   s = s + i;
ret
```

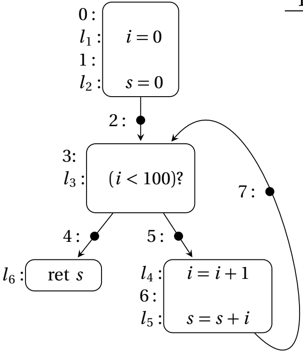


prog. point	$[i]$	$[s]$
0	$\top$	$\top$
1	$[0, 0]$	$\top$
2	$[0, 0]$	$[0, 0]$
3	$[0, 100]$	$[0, +\infty[$
4	$[100, 100]$	$[0, +\infty[$
5	$[0, 99]$	$[0, +\infty[$
6	$[0, 100]$	$[0, +\infty[$
7	$[0, 100]$	$[0, +\infty[$

Sparse Analysis

Range Analysis: $[v]^p$ intervals of possible values variable v might assume at program point p

```
i = 0;
s = 0;
while (i < 100)
   i = i + 1;
   s = s + i;
ret
```

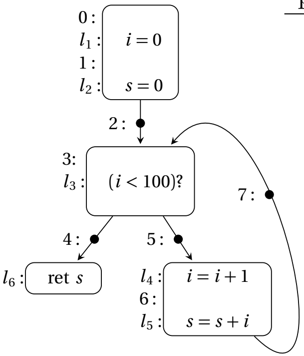


prog. point	$[i]$	$[s]$
0	$\top$	$\top$
1	$[0, 0]$	$\top$
2	$[0, 0]$	$[0, 0]$
3	$[0, 100]$	$[0, +\infty[$
4	$[100, 100]$	$[0, +\infty[$
5	$[0, 99]$	$[0, +\infty[$
6	$[0, 100]$	$[0, +\infty[$
7	$[0, 100]$	$[0, +\infty[$

Sparse Analysis

Redundancies: e.g. $[i]^1 = [i]^2$; because identity transfer function for $[i]$ from 1 to 2.

```
$i = 0$;
$s = 0$;
while $(i < 100)$
    $i = i + 1$;
    $s = s + i$;
ret
```

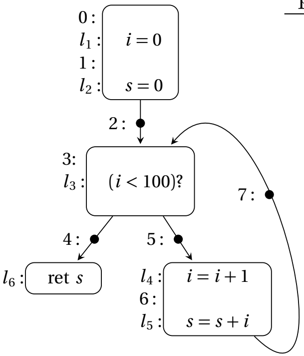


prog. point	$[i]$	$[s]$
0	$\top$	$\top$
1	$[0, 0]$	$\top$
2	$[0, 0]$	$[0, 0]$
3	$[0, 100]$	$[0, +\infty[$
4	$[100, 100]$	$[0, +\infty[$
5	$[0, 99]$	$[0, +\infty[$
6	$[0, 100]$	$[0, +\infty[$
7	$[0, 100]$	$[0, +\infty[$

Sparse Analysis

Sparse data-flow analysis: shortcut identity transfer functions by grouping contiguous program points bound to identities into larger regions

```
$i = 0$;
$s = 0$;
while ($i < 100$)
  $i = i + 1$;
  $s = s + i$;
ret
```

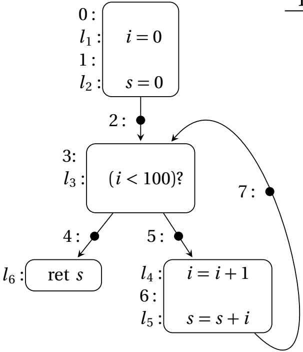


prog. point	$[i]$	$[s]$
0	$\top$	$\top$
1	$[0, 0]$	$\top$
2	$[0, 0]$	$[0, 0]$
3	$[0, 100]$	$[0, +\infty[$
4	$[100, 100]$	$[0, +\infty[$
5	$[0, 99]$	$[0, +\infty[$
6	$[0, 100]$	$[0, +\infty[$
7	$[0, 100]$	$[0, +\infty[$

Sparse Analysis

Sparse data-flow analysis: replace all $[v]^p$ by $[v]$ $(\forall v,\ p \in \mathrm{live}(v))$; propagate along def-use chains.

```
$i = 0$;
$s = 0$;
while $(i < 100)$
   $i = i + 1$;
   $s = s + i$;
ret
```



prog. point	$[i]$	$[s]$
0	$\top$	$\top$
1	$[0, 0]$	$\top$
2	$[0, 0]$	$[0, 0]$
3	$[0, 100]$	$[0, +\infty[$
4	$[100, 100]$	$[0, +\infty[$
5	$[0, 99]$	$[0, +\infty[$
6	$[0, 100]$	$[0, +\infty[$
7	$[0, 100]$	$[0, +\infty[$

Partitioned Lattice per Variable Problems

Partitioned Lattice per Variable (PLV) Problem

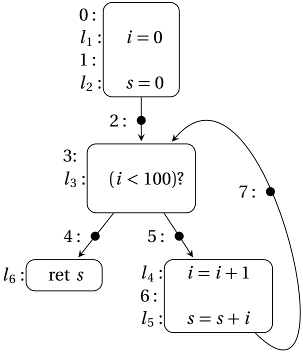
- program variables: v_i; program points: p; lattice: $\mathcal{L}$
- abstract state associated to prog. point p: x^p
- transfer function associated with $s \in \mathit{preds}(p)$: $F^{s,p}$
- constraint system: $x^p = x^p \wedge F^{s,p}(x^s)$ (or eq. $x^p \sqsubseteq F^{s,p}(x^s)$)

The corresponding Max. Fixed Point (MFP) problem is a PLV problem iff $\mathcal{L} = \mathcal{L}_{v_1} \times \cdots \times \mathcal{L}_{v_n}$ where each $\mathcal{L}_{v_i}$ is the lattice associated with v_i i.e. $x^s = ([v_1]^s, \ldots, [v_n]^s)$. Thus $F^{s,p} = F^{s,p}_{v_1} \times \cdots \times F^{s,p}_{v_n}$ and $[v_i]^p = [v_i]^p \wedge F^{s,p}_{v_i}([v_1]^s, \ldots, [v_n]^s)$.

Partitioned Lattice per Variable Data-Flow Problem

Range analysis

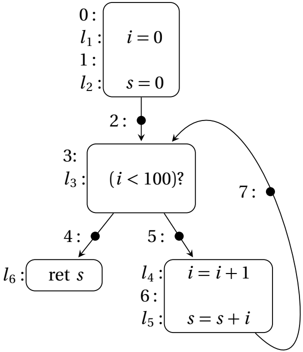
- $[i]^0 = [i]^0 \wedge F_i^{r,0}([i]^r, [s]^r)$
- $[i]^1 = [i]^1 \wedge F_i^{l_1}([i]^0, [s]^0)$
- $[i]^2 = [i]^2 \wedge F_i^{l_2}([i]^1, [s]^1)$
- $[i]^3 = [i]^3 \wedge F_i^{2,3}([i]^2, [s]^2)$
- $[i]^3 = [i]^3 \wedge F_i^{7,3}([i]^7, [s]^7)$
- $[i]^4 = [i]^4 \wedge F_i^{\overline{l_3}}([i]^3, [s]^3)$
- $[i]^5 = [i]^5 \wedge F_i^{l_3}([i]^3, [s]^3)$
- $[i]^6 = [i]^6 \wedge F_i^{l_4}([i]^5, [s]^5)$
- $[i]^7 = [i]^7 \wedge F_i^{l_5}([i]^6, [s]^6)$



Partitioned Lattice per Variable Data-Flow Problem

Range analysis

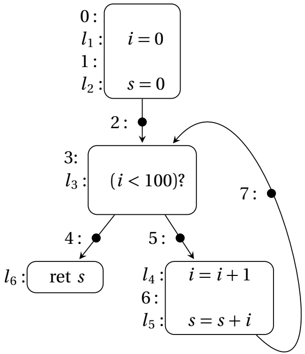
- $[i]^0 = [i]^0 \cup F_i^{r,0}([i]^r, [s]^r)$
- $[i]^1 = [i]^1 \cup F_i^{l_1}([i]^0, [s]^0)$
- $[i]^2 = [i]^2 \cup F_i^{l_2}([i]^1, [s]^1)$
- $[i]^3 = [i]^3 \cup F_i^{2,3}([i]^2, [s]^2)$
- $[i]^3 = [i]^3 \cup F_i^{7,3}([i]^7, [s]^7)$
- $[i]^4 = [i]^4 \cup F_i^{\overline{l_3}}([i]^3, [s]^3)$
- $[i]^5 = [i]^5 \cup F_i^{l_3}([i]^3, [s]^3)$
- $[i]^6 = [i]^6 \cup F_i^{l_4}([i]^5, [s]^5)$
- $[i]^7 = [i]^7 \cup F_i^{l_5}([i]^6, [s]^6)$



Partitioned Lattice per Variable Data-Flow Problem

Range analysis

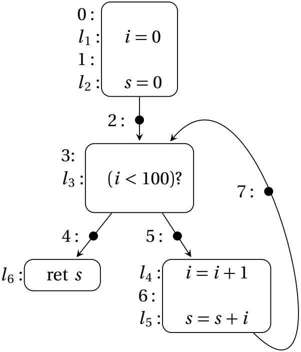
- $[i]^0 = [i]^0$
- $[i]^1 = [i]^1 \cup [0,0]$
- $[i]^2 = [i]^2 \cup [i]^1$
- $[i]^3 = [i]^3 \cup [i]^2$
- $[i]^3 = [i]^3 \cup [i]^7$
- $[i]^4 = [i]^4 \cup \left([i]^3 \cap [100, +\infty[\right)$
- $[i]^5 = [i]^5 \cup \left([i]^3 \cap]-\infty, 99[\right)$
- $[i]^6 = [i]^6 \cup \left([i]^5 + 1\right)$
- $[i]^7 = [i]^7 \cup [i]^6$

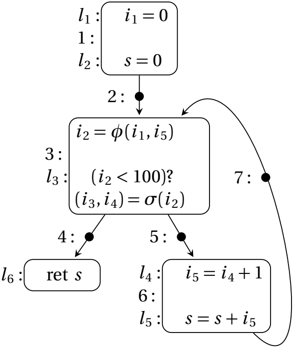


The Static Single Information Property

SSIfy (forward)

Modify the code (split live-ranges) without modifying its semantic s.t. fullfils SSI property

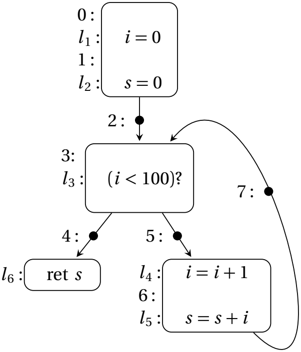


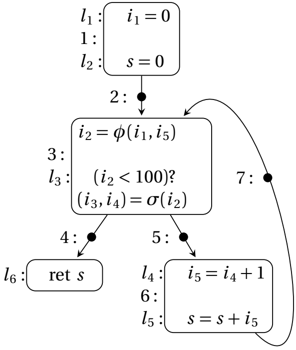


The Static Single Information Property

SPLIT

if s unique pred. of $p \in \mathrm{live}(v)$ and such that $F_v^{s,p} \neq \lambda x.\top$ is non-trivial, then s should contain a definition of v

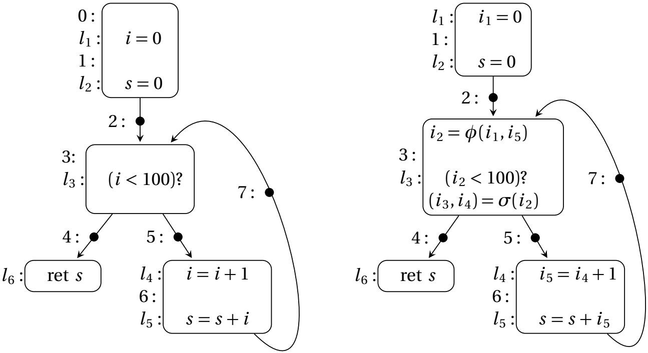




The Static Single Information Property

SPLIT

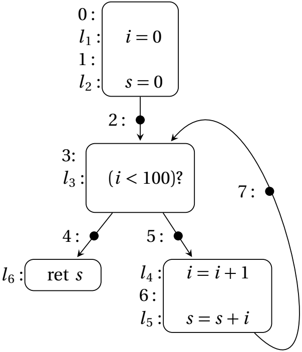
if s and t two preds of p such that $F_v^{s,p}(Y) \neq F_v^{t,p}(Y)$ (Y a MFP solution), then there must be a ϕ-function at entry of p

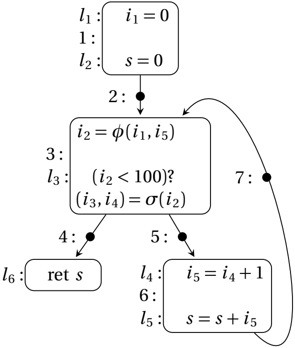


The Static Single Information Property

INFO

if $F_v^{s,p} \neq \lambda x.\top$,
then $v \in \text{live}(p)$

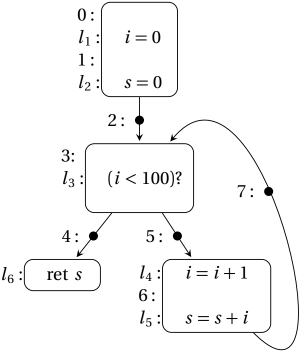


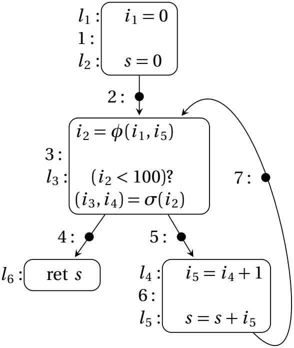


The Static Single Information Property

VERSION

for each variable v, $\mathrm{live}(v)$ is a connected component of the CFG.

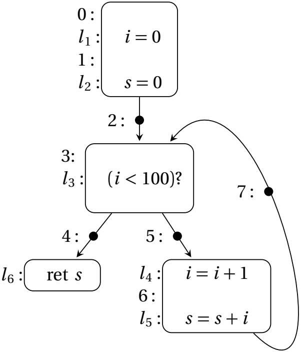


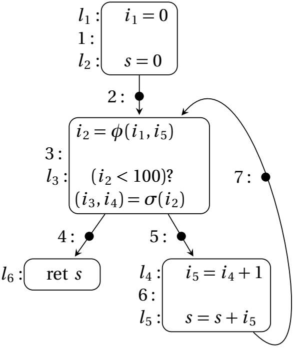


The Static Single Information Property

LINK

if F_v^{inst} depends on some $[u]^s$,
then *inst* should contain an use of u live-in at *inst*.

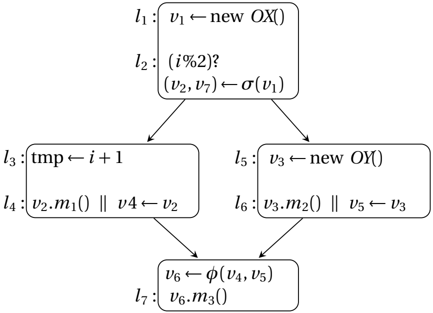




Special instructions used to split live ranges

Interior nodes (unique predecessor, unique successor)

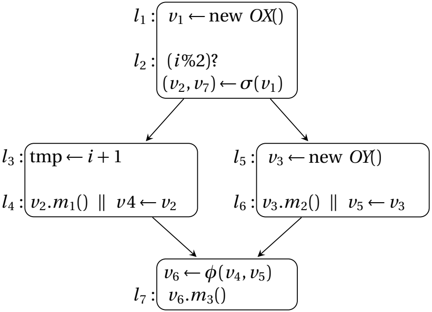
$$\mathit{inst} \;\|\; v_1 = v_1' \;\|\; \ldots \;\|\; v_m = v_m'$$



Special instructions used to split live ranges

joins (multiple predecessors, one successor)

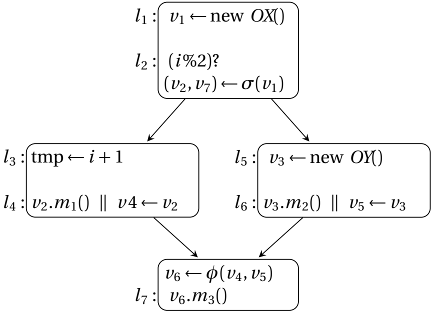
ϕ-functions



Special instructions used to split live ranges

branch points (one predecessor, mulitple successors)

$$(l^1 : v_1^1, \ldots, l^q : v_1^q) = \sigma(v_1) \;\; \| \;\; \ldots \;\; \| \;\; (l^1 : v_m^1, \ldots, l^q : v_m^q) = \sigma(v_m)$$



l_1 : $v_1 \leftarrow \text{new } OX()$

l_2 : $(i\%2)?$

$(v_2, v_7) \leftarrow \sigma(v_1)$

l_3 : $\text{tmp} \leftarrow i + 1$

l_4 : $v_2.m_1() \;\|\; v4 \leftarrow v_2$

l_5 : $v_3 \leftarrow \text{new } OY()$

l_6 : $v_3.m_2() \;\|\; v_5 \leftarrow v_3$

$v_6 \leftarrow \phi(v_4, v_5)$

l_7 : $v_6.m_3()$

Propagating Information Forwardly and Backwardly

Dense constrained system

$[v]^p = [v]^p \wedge F_v^{s,p}([v_1]^s, \ldots, [v_n]^s)$

Sparse SSI constrained system

$[v] = [v] \wedge G_v^i([a], \ldots, [z])$ where $a, \ldots, z$ are used (resp. defined) at i

Proof

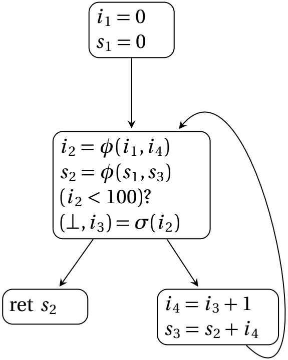
- coalesce all $[v]^p$ such that $v \in \mathrm{live}(p)$ into $[v]$; replace all $[v]^p$ such that $v \notin \mathrm{live}(p)$ by $\top$
- for each instruction *inst* with uses $a \ldots z$, let $G_v^i([a], \ldots, [z]) = F_v^i([v_1], \ldots, [v_n])$
- remove redundancies

Propagating Information Forwardly and Backwardly

Backward propagation engine under SSI

```
function back_propagate(transfer_functions $\mathcal{G}$)
    $\mathit{worklist} = \emptyset$
    foreach  $v \in \mathrm{vars}$:  $[v] = \top$
    foreach  $i \in \mathrm{insts}$:  $\mathit{worklist} \mathrel{+}= i$
    while  $\mathit{worklist} \neq \emptyset$:
        let  $i \in \mathit{worklist}$; $\mathit{worklist} \mathrel{-}= i$
        foreach  $v \in i.\mathit{uses}()$:
            $[v]_{new} = [v] \wedge G_v^i([i.\mathit{defs}()])$
            if  $[v] \neq [v]_{new}$:
                $\mathit{worklist} \mathrel{+}= v.\mathit{defs}()$
                $[v] = [v]_{new}$
```

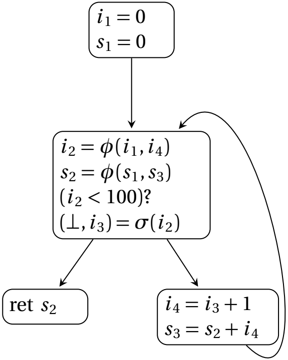
Examples of sparse data-flow analyses



Range analysis (forward from defs & conds)

$[i_1] \cup= [0,0]$	$\emptyset$
$[s_1] \cup= [0,0]$	$\emptyset$
$[i_2] \cup= [i_1] \cup [i_4]$	$\emptyset$
$[s_2] \cup= [s_1] \cup [s_3]$	$\emptyset$
$[i_3] \cup= ([i_2] \cap]-\infty, 99])$	$\emptyset$
$[i_4] \cup= ([i_3] + 1)$	$\emptyset$
$[s_3] \cup= ([s_2] + [i_4])$	$\emptyset$

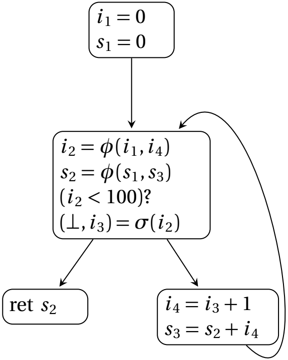
Examples of sparse data-flow analyses



Range analysis (forward from defs & conds)

$$\begin{array}{l|l}
[i_1] \cup= [0,0] & \emptyset \\
[s_1] \cup= [0,0] & \emptyset \\
[i_2] \cup= [i_1] \cup [i_4] & \emptyset \\
[s_2] \cup= [s_1] \cup [s_3] & \emptyset \\
[i_3] \cup= ([i_2] \cap]-\infty, 99]) & \emptyset \\
[i_4] \cup= ([i_3] + 1) & \emptyset \\
[s_3] \cup= ([s_2] + [i_4]) & \emptyset
\end{array}$$

Examples of sparse data-flow analyses



$i_1 = 0$
$s_1 = 0$

$i_2 = \phi(i_1, i_4)$
$s_2 = \phi(s_1, s_3)$
$(i_2 < 100)?$
$(\perp, i_3) = \sigma(i_2)$

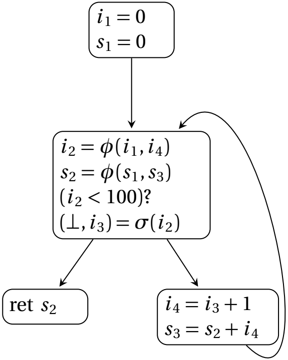
ret s_2

$i_4 = i_3 + 1$
$s_3 = s_2 + i_4$

Range analysis (forward from defs & conds)

$$
\begin{array}{l|l}
[i_1] \cup= [0,0] & \color{red}{[0,0]} \\
[s_1] \cup= [0,0] & \emptyset \\
[i_2] \cup= [i_1] \cup [i_4] & \emptyset \\
[s_2] \cup= [s_1] \cup [s_3] & \emptyset \\
[i_3] \cup= ([i_2] \cap]-\infty, 99]) & \emptyset \\
[i_4] \cup= ([i_3] + 1) & \emptyset \\
[s_3] \cup= ([s_2] + [i_4]) & \emptyset
\end{array}
$$

Examples of sparse data-flow analyses



$i_1 = 0$
$s_1 = 0$

$i_2 = \phi(i_1, i_4)$
$s_2 = \phi(s_1, s_3)$
$(i_2 < 100)?$
$(\perp, i_3) = \sigma(i_2)$

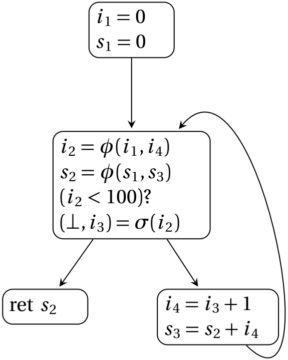
ret s_2

$i_4 = i_3 + 1$
$s_3 = s_2 + i_4$

Range analysis (forward from defs & conds)

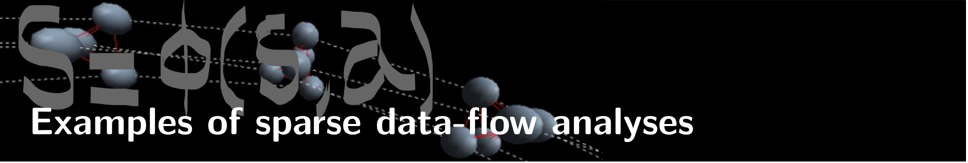
$$\begin{array}{l|l}
[i_1] \cup= [0,0] & [0,0] \\
[s_1] \cup= [0,0] & \emptyset \\
[i_2] \cup= [i_1] \cup [i_4] & \emptyset \\
[s_2] \cup= [s_1] \cup [s_3] & \emptyset \\
[i_3] \cup= ([i_2] \cap]-\infty, 99]) & \emptyset \\
[i_4] \cup= ([i_3] + 1) & \emptyset \\
[s_3] \cup= ([s_2] + [i_4]) & \emptyset
\end{array}$$

Examples of sparse data-flow analyses

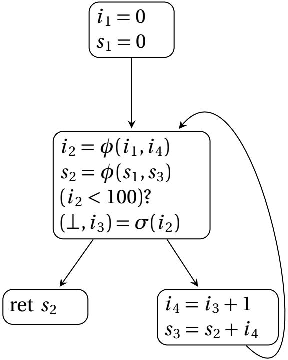


Range analysis (forward from defs & conds)

$$\begin{array}{l|l}
[i_1] \cup= [0,0] & [0,0] \\
[s_1] \cup= [0,0] & [0,0] \\
[i_2] \cup= [i_1] \cup [i_4] & \emptyset \\
[s_2] \cup= [s_1] \cup [s_3] & \emptyset \\
[i_3] \cup= ([i_2] \cap]-\infty, 99]) & \emptyset \\
[i_4] \cup= ([i_3] + 1) & \emptyset \\
[s_3] \cup= ([s_2] + [i_4]) & \emptyset
\end{array}$$



Examples of sparse data-flow analyses



$i_1 = 0$
$s_1 = 0$

$i_2 = \phi(i_1, i_4)$
$s_2 = \phi(s_1, s_3)$
$(i_2 < 100)?$
$(\bot, i_3) = \sigma(i_2)$

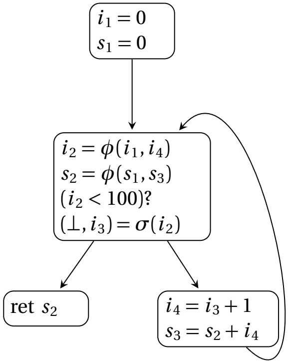
ret s_2

$i_4 = i_3 + 1$
$s_3 = s_2 + i_4$

Range analysis (forward from defs & conds)

$$
\begin{array}{l|l}
[i_1] \cup= [0,0] & [0,0] \\
[s_1] \cup= [0,0] & [0,0] \\
\color{red}{[i_2] \cup= [i_1] \cup [i_4]} & \emptyset \\
[s_2] \cup= [s_1] \cup [s_3] & \emptyset \\
[i_3] \cup= ([i_2] \cap]-\infty, 99]) & \emptyset \\
[i_4] \cup= ([i_3] + 1) & \emptyset \\
[s_3] \cup= ([s_2] + [i_4]) & \emptyset
\end{array}
$$

Examples of sparse data-flow analyses



$i_1 = 0$
$s_1 = 0$

$i_2 = \phi(i_1, i_4)$
$s_2 = \phi(s_1, s_3)$
$(i_2 < 100)?$
$(\perp, i_3) = \sigma(i_2)$

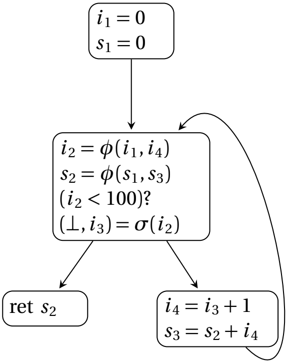
ret s_2

$i_4 = i_3 + 1$
$s_3 = s_2 + i_4$

Range analysis (forward from defs & conds)

$$\begin{array}{l|l}
[i_1] \cup= [0,0] & [0,0] \\
[s_1] \cup= [0,0] & [0,0] \\
[i_2] \cup= [i_1] \cup [i_4] & {\color{red}[0,0]} \\
[s_2] \cup= [s_1] \cup [s_3] & \emptyset \\
[i_3] \cup= ([i_2] \cap]-\infty, 99]) & \emptyset \\
[i_4] \cup= ([i_3] + 1) & \emptyset \\
[s_3] \cup= ([s_2] + [i_4]) & \emptyset
\end{array}$$

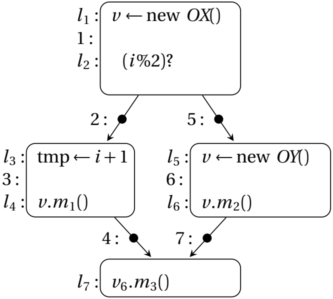
Examples of sparse data-flow analyses



Range analysis (forward from defs & conds)

$[i_1] \cup= [0,0]$	$[0,0]$
$[s_1] \cup= [0,0]$	$[0,0]$
$[i_2] \cup= [i_1] \cup [i_4]$	$[0,100]$
$[s_2] \cup= [s_1] \cup [s_3]$	$[0,+\infty[$
$[i_3] \cup= ([i_2] \cap]-\infty, 99])$	$[0,99]$
$[i_4] \cup= ([i_3] + 1)$	$[1,100]$
$[s_3] \cup= ([s_2] + [i_4])$	$[1,+\infty[$

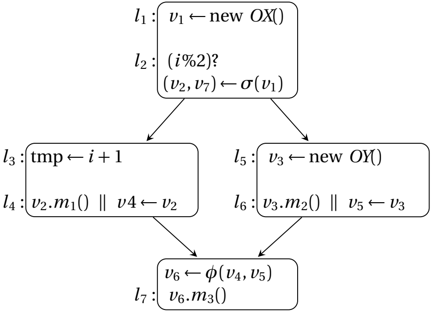
Examples of sparse data-flow analyses



Class inference (backward from uses)

prog. point	$[v]$
1	$\{m_1, m_3\}$
2	$\{m_1, m_3\}$
3	$\{m_1, m_3\}$
4	$\{m_3\}$
5	$\top$
6	$\{m_2, m_3\}$
7	$\{m_3\}$

Examples of sparse data-flow analyses



$l_1:$ $v_1 \leftarrow \text{new } OX()$

$l_2:$ $(i\%2)?$

$(v_2, v_7) \leftarrow \sigma(v_1)$

$l_3:$ $\text{tmp} \leftarrow i+1$

$l_4:$ $v_2.m_1() \parallel v4 \leftarrow v_2$

$l_5:$ $v_3 \leftarrow \text{new } OY()$

$l_6:$ $v_3.m_2() \parallel v_5 \leftarrow v_3$

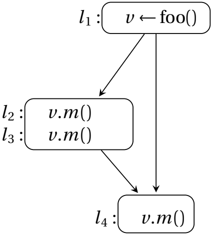
$v_6 \leftarrow \phi(v_4, v_5)$

$l_7:$ $v_6.m_3()$

Class inference (backward from uses)

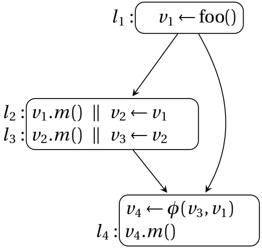
$[v_7]$	$\top$
$[v_6] \cup= \{m_3\}$	$\{m_3\}$
$[v_5] \cup= [v_6]$	$\{m_3\}$
$[v_4] \cup= [v_6]$	$\{m_3\}$
$[v_2] \cup= (\{m_1\} \wedge [v_4])$	$\{m_1, m_3\}$
$[v_3] \cup= (\{m_2\} \wedge [v_4])$	$\{m_2, m_3\}$
$[v_1] \cup= [v_2]$	$\{m_1, m_3\}$
$[v_1] \cup= [v_7]$	$\{m_1, m_3\}$

Examples of sparse data-flow analyses



Null pointer (forward from defs & uses)

Examples of sparse data-flow analyses



Null pointer (forward from defs & uses)

$$\begin{array}{ll|l}
[v_1] & \wedge= 0 & 0 \\
[v_2] & \wedge= \cancel{0} & \cancel{0} \\
[v_3] & \wedge= \cancel{0} & \cancel{0} \\
[v_4] & \wedge= ([v_3] \wedge [v_1]) & 0
\end{array}$$

Splitting strategy

Live range splitting strategy $\mathcal{P}_v = I_\uparrow \cup I_\downarrow$

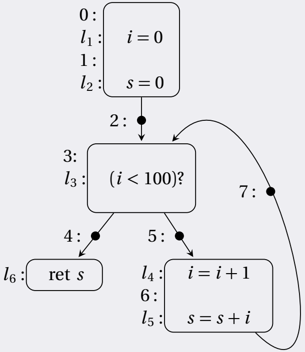
$I_\downarrow$: set of points i with forward direction

$I_\uparrow$: set of points i with backward direction

```
function SSIfy(var v, Splitting_Strategy $\mathcal{P}_v$)
    split($v$, $\mathcal{P}_v$)
    rename($v$)
    clean($v$)
```

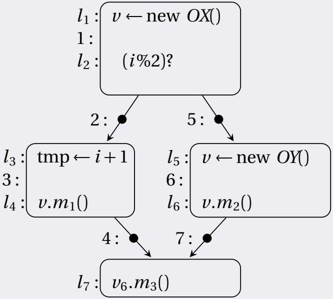
Splitting strategy

Range analysis: $\mathcal{P}_i = \{l_1, \mathrm{Out}(l_3), l_4\}_\downarrow$



Splitting strategy

Class inference: $\mathcal{P}_v = \{l_4, l_6, l_7\}_\uparrow$



l_1 : $v \leftarrow \text{new } OX()$
1 :
l_2 : $(i\%2)?$

2 :
5 :

l_3 : $\text{tmp} \leftarrow i+1$
3 :
l_4 : $v.m_1()$

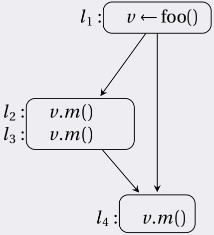
l_5 : $v \leftarrow \text{new } OY()$
6 :
l_6 : $v.m_2()$

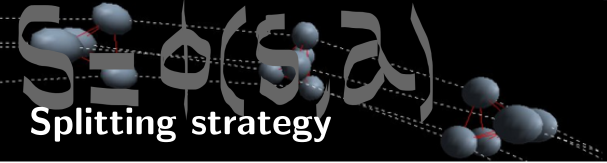
4 :
7 :

l_7 : $v_6.m_3()$

Splitting strategy

Null pointer: $\mathcal{P}_v = \{l_1, l_2, l_3, l_4\}_\downarrow$





Splitting strategy

Client	**Splitting strategy $\mathcal{P}$**
Alias analysis, reaching definitions cond. constant propagation	$\mathit{Defs}_{\downarrow}$
Partial Redundancy Elimination	$\mathit{Defs}_{\downarrow} \bigcup \mathit{LastUses}_{\uparrow}$
ABCD, taint analysis, range analysis	$\mathit{Defs}_{\downarrow} \bigcup \mathcal{Out}(\mathit{Conds})_{\downarrow}$
Stephenson's bitwidth analysis	$\mathit{Defs}_{\downarrow} \bigcup \mathcal{Out}(\mathit{Conds})_{\downarrow} \bigcup \mathit{Uses}_{\uparrow}$
Mahlke's bitwidth analysis	$\mathit{Defs}_{\downarrow} \bigcup \mathit{Uses}_{\uparrow}$
An's type inference, Class inference	$\mathit{Uses}_{\uparrow}$
Hochstadt's type inference	$\mathit{Uses}_{\uparrow} \bigcup \mathcal{Out}(\mathit{Conds})_{\uparrow}$
Null-pointer analysis	$\mathit{Defs}_{\downarrow} \bigcup \mathit{Uses}_{\downarrow}$

Splitting live ranges

- Split live range of v at each $p \in \mathcal{P}_v$
- Split live range where the information collide (join set $\mathcal{J}(I_\downarrow)$ and split set $\mathcal{S}(I_\uparrow)$)
- Iterated dominance frontier $\mathrm{DF}^+(S) = \mathcal{J}(S \cup \{r\})$ can be computed efficiently (as opposed to $\mathcal{J}(S)$)
- Iterated post dominance frontier $\mathrm{pDF}^+(S) = \mathcal{J}(S \cup \{r\})$ for the reverse CFG

function split(var *v*, Splitting_Strategy $\mathcal{P}_v = I_\downarrow \cup I_\uparrow$)

$$\left[I_\downarrow \cup \mathrm{In}(\mathrm{DF}^+(I_\downarrow))\right]$$

Splitting live ranges

- Split live range of v at each $p \in \mathcal{P}_v$
- Split live range where the information collide (join set $\mathcal{J}(I_\downarrow)$ and split set $\mathcal{S}(I_\uparrow)$)
- Iterated dominance frontier $\mathrm{DF}^+(S) = \mathcal{J}(S \cup \{r\})$ can be computed efficiently (as opposed to $\mathcal{J}(S)$)
- Iterated post dominance frontier $\mathrm{pDF}^+(S) = \mathcal{J}(S \cup \{r\})$ for the reverse CFG

function split(var *v*, Splitting_Strategy $\mathcal{P}_v = I_\downarrow \cup I_\uparrow$)

$$\left[I_\downarrow \cup \mathrm{In}(\mathrm{DF}^+(I_\downarrow))\right] \cup \left[I_\uparrow \cup \mathrm{Out}(\mathrm{pDF}^+(I_\uparrow))\right]$$

Splitting live ranges

- Split live range of v at each $p \in \mathcal{P}_v$
- Split live range where the information collide (join set $\mathcal{J}(I_\downarrow)$ and split set $\mathcal{S}(I_\uparrow)$)
- Iterated dominance frontier $\mathrm{DF}^+(S) = \mathcal{J}(S \cup \{r\})$ can be computed efficiently (as opposed to $\mathcal{J}(S)$)
- Iterated post dominance frontier $\mathrm{pDF}^+(S) = \mathcal{J}(S \cup \{r\})$ for the reverse CFG

function split(var *v*, Splitting_Strategy $\mathcal{P}_v = I_\downarrow \cup I_\uparrow$)

$$\mathcal{P}_v \cup \mathrm{In}\left[\mathrm{DF}^+(I_\downarrow \cup \left[I_\uparrow \cup \mathrm{Out}(\mathrm{pDF}^+(I_\uparrow))\right])\right]$$

Variable Renaming

function rename(var v)

- traverses the CFG along topological order
- give a unique version to each definition of v
- stack the versions that dominates the current program point
- rename each use of v with the version of immediately dominating definition

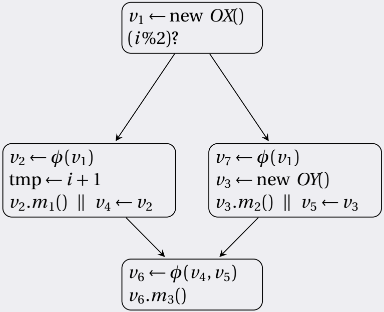
Dead and Undefined Code Elimination

clean(var v)

- actual instructions: instructions originally in the code
- SSA graph: nodes are instructions; edges are def-use chains
- active instructions: instructions connected to an actual instruction
- simple traversal of the SSA graph from actual instructions that mark active ones
- remove non-active instructions (inserted phi and sigma functions)

Implementation Details

Implementing σ-functions



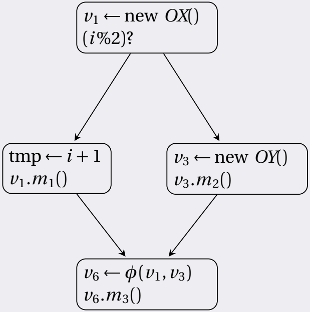
$v_1 \leftarrow$ new $OX()$
$(i\%2)$?

$v_2 \leftarrow \phi(v_1)$
tmp $\leftarrow i+1$
$v_2.m_1() \parallel v_4 \leftarrow v_2$

$v_7 \leftarrow \phi(v_1)$
$v_3 \leftarrow$ new $OY()$
$v_3.m_2() \parallel v_5 \leftarrow v_3$

$v_6 \leftarrow \phi(v_4, v_5)$
$v_6.m_3()$

SSI destruction



$v_1 \leftarrow$ new $OX()$
$(i\%2)$?

tmp $\leftarrow i+1$
$v_1.m_1()$

$v_3 \leftarrow$ new $OY()$
$v_3.m_2()$

$v_6 \leftarrow \phi(v_1, v_3)$
$v_6.m_3()$