Special Topics in Logic and Security 1

Reduced Product. Type Casting and Wrapping.

Paul Irofti

Master Year II, Sem. I, 2023-2024

Memory Access

What happens with the *Pts* domain in the program below?

```
int A[4][8] = {...};
uint i, j;
uint sum = 0;

for (i = 0; i < 4; i++)
    for (j = 0; j < 8; j++)
        sum += A[i][j];

printf("sum = %d\n", sum);
```

The Pts Abstract Domain

Definition

Define $\mathcal{X}$ the finite set of variables of a program P and $\mathcal{A}$ the finite set of addresses towards which these variables can point.

Then $Pts = \mathcal{X} \to \mathcal{P}(\mathcal{A})$ represents the set of maps that tie each variable $x \in \mathcal{X}$ to a subset of addresses $A(x) \in \mathcal{A}$.

Let $A_1, A_2, A' \in Pts$.

Update: $A \in Pts$ becomes $A' = \{A \cup [x \to a] \mid a \in \mathcal{A}\}$ such that $A'(x) = a$ and $A'(y) = A(y), \forall y \neq x$.

Order: $A_1 \leq_A A_2 \iff A_1(x) \subseteq A_2(x), \, \forall x \in \mathcal{X}$

Join: $A' = A_1 \vee_A A_2$ s.t. $A'(x) = A_1(x) \cup A_2(x), \, \forall x \in \mathcal{X}$.

Meet: The meet operation can be seen as an update operation that helps us filter the elements of A.

The Poly Abstract Domain

The lattice $(Poly, \leq_P, \vee_P, \wedge_P)$:

- $\leq_P$ is the inclusion operator $\subseteq$
- $\vee_P = \overline{\Upsilon}$ is the join operation for polyhedra
- $\wedge_P$ is the meet operation for sets

The lattice is incomplete because the join and meet operations, when applied to an arbitrary number of polyhedra, can lead to a non-polyhedra object.

The widening operator together with the incomplete lattice restrain the number of fixed points that can be attained.

Definition

A stable polyhedra obtained at convergence is generally a post-fixpoint: a polyhedra that contains the polyhedra of the fixed point. An approximation.

General assignment operations can be implemented as:

$$P \triangleright x := e = \exists_t(\llbracket\{x = t\}\rrbracket \wedge_P \exists_x(P \wedge_P \llbracket\{t = e\}\rrbracket))$$

The Mult Abstract Domain

Let $M, M', M_1, M_2 \in Mult$.

Update: $M \to M' = M[x \to n'] \implies M'(x) = n'$ and $M'(y) = M(y), \forall y \neq x$.

Join: $M' = M_1 \vee_M M_2$ s.t. $M'(x) = \min(M_1(x), M_2(x)), \forall x \in \mathcal{X}$.

Inclusion: $M_1 \subseteq_M M_2 \iff M_1(x) \geq M_2(x), \forall x \in \mathcal{X}$.

Exercise: Find the $\top$ element: the largest element from the lattice. Explain.

Let $Equ = Lin \times \mathbb{Z}$ be the set of linear equations of the type $e = c$, where $e \in Lin, c \in \mathbb{Z}$.

Meet: $\wedge_M : Mult \times Equ \to (Mult \cup \{\perp_M\})$, where $\perp_M$ tags invalid states. The intersection operator adds the information provided by a new equation: $M' = M \wedge_M (e = c)$.

$$M' = M\left[x_j \to \max\left(M(x_j), \min(\delta(c), \min_{i, i \neq j} \delta(a_i) + M(x_i)) - \delta(a_j)\right)\right]$$

Invalid state if $\min_{i=1,\dots,n} \delta(a_i) + M(x_i) > \delta(c)$.

The Num Abstract Domain

Let $Num = (Poly \times Mult) \cup \{\bot_N\}$, where $\bot_N$ represents an unreachable state, that is impossible to attain, in the program definition. We define:

- $(P, M) \subseteq_N (P', M') \iff (P \subseteq_P P') \wedge (M \subseteq_M M')$
- $(P', M') = (P_1, M_1) \vee_N (P_2, M_2) \iff (P' = P_1 \vee_P P_2) \wedge (M' = M_1 \vee_M M_2)$
- $(P', M') = (P, M) \triangleright x := e \iff (P' = P \triangleright x := e) \wedge (M' = M \triangleright x := e)$
- $(P', M') = (P, M) \triangleright x := e \gg n \iff (P' = P \triangleright x := e \gg n) \wedge (M' = M \triangleright x := e \gg n)$
- $(P', M') = \exists_x(P, M) \iff (P' = \exists_x(P)) \wedge (M' = \exists_x(M))$
- $(P, M) \wedge_N \{e = c\} = \begin{cases} \bot_N & \text{if } P' = \emptyset \text{ or } M' = \bot_M \\ (P', M') & \text{otherwise} \end{cases}$, where

  $P' = P \wedge_P [\![\{e = c\}]\!]$ and $M' = M \wedge_M \{e = c\}$.

Num reductions

Note that the *Num* meet operator $\wedge_N$ has the following reduction property:

$$(P, M) \wedge_N \{e = c\} = \bot_N \quad \text{if } P' = \emptyset \text{ or } M' = \bot_M$$

where states such as $(\emptyset, M)$ or $(P, \bot_M)$ lead to $\bot_N$.

This reduction avoids the propagation of unsatisfaiable domains as seen in the `strings` example.

Definition

Reduced product. Combination of two domains that is implemented as one in order to provide states where no further reduction is possible.

Thus such a reduction is possible between the Poly and Mult domains.

In the following we are going to see an example that leads to ways of incorporating information $Mult \to Poly$ and $Poly \to Mult$.

Example: Reduction

Let N denote the initial state in which the variable x is unbound such that

```
L1: x = 4*y;
L2: if (rand())
L3:     y--;
```

Let us analyse this from the *Num* perspective:

Example: Reduction

Let N denote the initial state in which the variable x is unbound such that

```
L1: x = 4*y;
L2: if (rand())
L3:     y--;
```

Let us analyse this from the *Num* perspective:

- `L1` defines $N_1 = N \triangleright x := 4y$

Example: Reduction

Let N denote the initial state in which the variable x is unbound such that

```
L1: x = 4*y;
L2: if (rand())
L3:     y--;
```

Let us analyse this from the *Num* perspective:

- `L1` defines $N_1 = N \triangleright x := 4y$
- `L3` defines $N_2 = N_1 \triangleright y := y - 1$ guarded by the `if` at `L2`

Example: Reduction

Let N denote the initial state in which the variable x is unbound such that

```
L1: x = 4*y;
L2: if (rand())
L3:     y--;
```

Let us analyse this from the *Num* perspective:

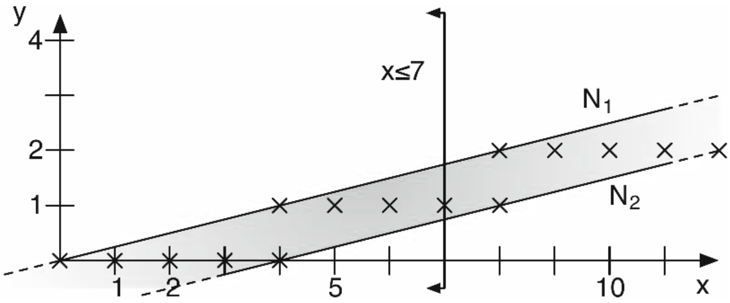
- L1 defines $N_1 = N \triangleright x := 4y$
- L3 defines $N_2 = N_1 \triangleright y := y - 1$ guarded by the if at L2
- $N_{12} = N_1 \vee_N N_2$ represents the state after the if statement

Example:

- $\{(0,0),(4,1),(8,2),(12,3)\ldots(4k,k)\} \in N_1$
- $\{(0,-1),(4,0),(8,1),(12,2)\ldots(4k,k-1)\} \in N_2$
- $N_{12} = N_1 \vee_N N_2$ and for the first element $(0,0)\overline{\Upsilon}(4,0) = ([0,4],0)$
- we just got three new possible elements!
- the same is true for $y = 1$ with points $(4,1)$ and $(8,1)$

Poly to Mult Propagation

The two lines represent N_1 and N_2, while the grey area represents N_{12}.



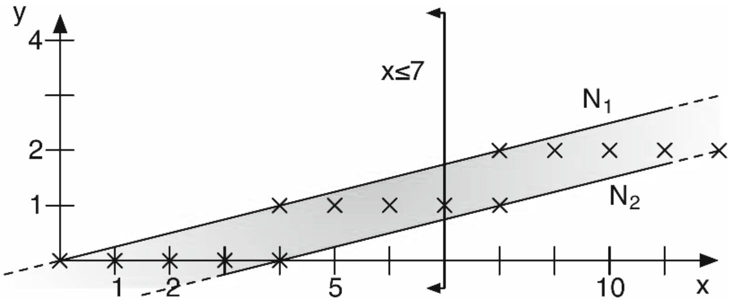
Source: A. Simon, Value Range Analysis of C Programs, 2009

Notice that adding the inequality $x \leq 7$ restricts the maximum value of x to 7!

Is that OK?

Poly to Mult Propagation

The two lines represent N_1 and N_2, while the grey area represents N_{12}.



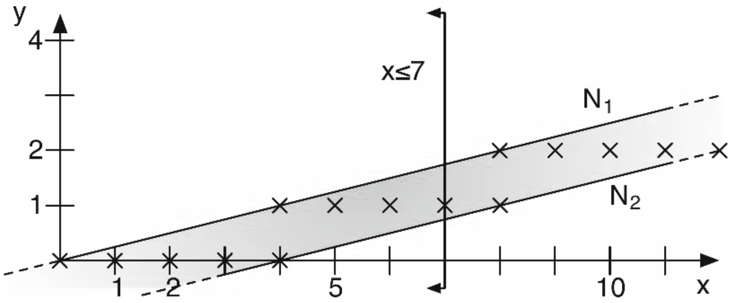
Source: A. Simon, Value Range Analysis of C Programs, 2009

Notice that adding the inequality $x \leq 7$ restricts the maximum value of x to 7!

Is that OK? Why not?

Poly to Mult Propagation

The two lines represent N_1 and N_2, while the grey area represents N_{12}.



Source: A. Simon, Value Range Analysis of C Programs, 2009

Notice that adding the inequality $x \leq 7$ restricts the maximum value of x to 7!

Is that OK? Why not? Because x is supposed to be a multiple of 4.

Example: Reduction via Mult

We should be able to restrict $N_{12} \wedge_N \{x \leq 7\}$ by information from the Mult domain:

- from $M_1 \in N_1$ we have $M_1(x) = 2$
- a linear translation by 4 implies that its multiplicity remains the same in N_2
- from N_2 it means it also remains the same in N_{12} due to the properties of $\vee_M$
- so the value of x after $x \leq 7$ is $x = 4$

More generally we reduce the states to $N_{12} \wedge_N \{x \leq 4\}$.

Counter Example

Let us add two more instructions to our program:

```
L1: x = 4*y;
L2: if (rand())
L3:     y--;
L4: z = x+1
L5: if (z <= 8) {}
```

This adds to the analysis:

- L4 defines $N_3 = N_{12} \triangleright z := x + 1$
- L5 defines $N_4 = N_3 \wedge_N \{z \leq 8\}$
- which should be equivalent to $x \leq 7$
- still we do not know anything about the multiplicity of z
- we assume $M(z) = 0$!

We can not refine N_4 without analyzing all the possible relationships of z with other variables in N_3.

Incorporating *Mult* $\to$ *Poly*

Idea: scale each variable $x \in P$ by $1/2^{M(x)}$

- intersection: $(P, M) \wedge_N \{ax \leq c\}$
- scaled version: $P' = P \wedge_N [\![\{(2^{M(x_1)}a_1, \dots, 2^{M(x_n)}a_n)x \leq c\}]\!]$
- *Num* with different multiplicities M and M' affect $\subseteq_N$ and $\vee_N$ operations
- $M(x) > M'(x)$ leads to scaling by $2^{M(x)-M'(x)}$

Example: $P_3 \subseteq_P [\![\{2^{M_3(z)}z = 2^{M_3(x)}x + 1\}]\!]$ where $M_3(z) = 0$ and $M_3(x) = 2$.

Thus $[\![\{2^{M_3(z)}z = 2^{M_3(x)}x + 1\}]\!] = [\![\{2^0 z = 2^2 x + 1\}]\!] = [\![\{z = 4x + 1\}]\!]$

$$\implies z \leq 8 \iff 4x + 1 \leq 8 \iff x \leq \frac{7}{4} = 1\frac{3}{4} \iff x \leq 1 \implies z \leq 5$$

Remark: Introducing the multiplicity information to polyhedras reduces their coefficients (see coef. growth issue). In our example the reduction tightens $x \leq 1 \cdot 2^{M(x)} = 4$ and $z \leq 5$.

Incorporating *Poly* $\rightarrow$ *Mult*

We can also incorporate information from *Poly* to *Mult*.

Example: $P \subseteq_P [\![\{x = 0\}]\!]$ then $M \in \mathit{Mult}$ is $M(x) = 64$.

Remark: In fact scaling by $1/2^{M(x)}$ in *Poly* can only be done through information propagation from *Mult*.

Notations: Let $N(ax + c) = [l, u]_{\equiv d}$ be the set of values $\{l, l + d, \ldots, u\} \subseteq \mathbb{Z}$ that $ax + c$ can take in N.

Let $[\![N]\!] \subseteq \mathbb{Z}^{|\mathcal{X}|}$ be the set of all *feasible* points in $N \in \mathit{Num}$.

Casting and Wrapping

Casting

Let us study the following code snippet:

```
while(*str) {
    dist[*str]++;
    str++;
};
```

Casting

Let us study the following code snippet:

```
while(*str) {
    dist[*str]++;
    str++;
};
```

Which warning will we get?

Casting

Let us study the following code snippet:

```
while(*str) {
    dist[*str]++;
    str++;
};
```

Which warning will we get? C standard asks for `int` iterators not `char`'s.

Casting

Let us study the following code snippet:

```
while(*str) {
    dist[*str]++;
    str++;
};
```

Which warning will we get? C standard asks for `int` iterators not `char`'s.

We fix it with a cast:

```
while(*str) {
    dist[(int)*str]++;
    str++;
};
```

Casting

Let us study the following code snippet:

```
while(*str) {
    dist[*str]++;
    str++;
};
```

Which warning will we get? C standard asks for `int` iterators not `char`'s.

We fix it with a cast:

```
while(*str) {
    dist[(int)*str]++;
    str++;
};
```

Does this pass peer-review?

Casting

Let us study the following code snippet:

```
while(*str) {
    dist[*str]++;
    str++;
};
```

Which warning will we get? C standard asks for `int` iterators not `char`'s.

We fix it with a cast:

```
while(*str) {
    dist[(int)*str]++;
    str++;
};
```

Does this pass peer-review? No! Negative indices are possible.

Casting

Let us study the following code snippet:

```
while(*str) {
    dist[*str]++;
    str++;
};
```

Which warning will we get? C standard asks for `int` iterators not `char`'s.

We fix it with a cast:

```
while(*str) {
    dist[(int)*str]++;
    str++;
};
```

Does this pass peer-review? No! Negative indices are possible.

Fine, make it unsigned...

```
while(*str) {
    dist[(uint)*str]++;
    str++;
};
```

Casting: Warnings Fixed

```
while(*str) {
    dist[(uint)*str]++;
    str++;
};
```

Casting: Warnings Fixed

```
while(*str) {
    dist[(uint)*str]++;
    str++;
};
```

Happy?

Casting: Warnings Fixed

```
while(*str) {
    dist[(uint)*str]++;
    str++;
};
```

Happy?

You should not be: C standard dictates: `char -> int -> uint`!

Casting: Warnings Fixed

```
while(*str) {
    dist[(uint)*str]++;
    str++;
};
```

Happy?

You should not be: C standard dictates: `char -> int -> uint`!

So what are the possible dist iterators?

Casting: Warnings Fixed

```
while(*str) {
    dist[(uint)*str]++;
    str++;
};
```

Happy?

You should not be: C standard dictates: `char -> int -> uint`!

So what are the possible dist iterators? $[2^{32}-128, 2^{32}-1] \cup [0, 127]$

Casting: Warnings Fixed

```
while(*str) {
    dist[(uint)*str]++;
    str++;
};
```

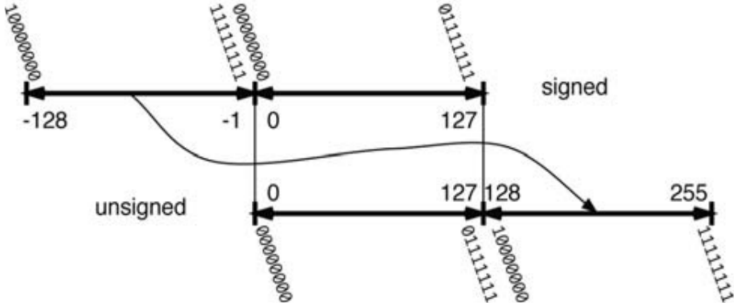
Happy?

You should not be: C standard dictates: `char -> int -> uint`!

So what are the possible dist iterators? $[2^{32} - 128, 2^{32} - 1] \cup [0, 127]$

Conclusion: we get positive indices but some are out-of-bounds! This is due to the **wrapping** of the negative indices.

Signed versus Unsigned



Source: A. Simon, Value Range Analysis of C Programs, 2009

Remarks

- subtracting from an integer is the same as adding the largest integer
- example: $(1, 1, 1, 1) + (0, 0, 0, 1) = (0, 0, 0, 0)$
- negative range of signed wraps to upper range of unsigned
- miss-match against the possible infinite range of polyhedral variables

Useful notations

Before handling the out-of-bounds case in our model, let us settle notations.

- Let $\mathbb{B} = \{0, 1\}$ be the Boolean set
- Let $b = (b_{w-1}, \dots, b_0) \in \mathbb{B}^w$ be a vector of bits
- `uint`: $\mathrm{val}^{w,\mathrm{uint}}(b) = \sum_{i=0}^{w-1} b_i 2^i$
- `int`: $\mathrm{val}^{w,\mathrm{int}}(b) = \sum_{i=0}^{w-2} b_i 2^i - b_{w-1} 2^{w-1}$
- Let $\mathrm{bin}^w : \mathbb{Z} \to \mathbb{B}^w$ which converts an integer to the lower w bits
- $\mathrm{bin}^w(v) = b \iff \exists b' \in \mathbb{B}^q \text{s.t.} \mathrm{val}^{q+w,\mathrm{int}}(b' \| b) = v$
- in the above $\|$ is the concatenation operator
- examples: $\mathrm{bin}^3(15) = (1, 1, 1) \quad \mathrm{val}^{5,\mathrm{int}}((0, 1, 1, 1, 1)) = 15$
- denote $+^w$ and $*^w$ addition and multiplication with truncation at w bits
- sign agnostic: $(1, 1, 1, 1) +^4 (0, 0, 0, 1) = (0, 0, 0, 0)$
- let $\mathcal{B} = \mathbb{B}^8$ the set of bytes and $\Sigma = \mathcal{B}^{2^{32}}$ all states of 4GB processes
- a given memory state is then $\sigma \in \Sigma$
- a byte access is $\sigma^s : [0, 2^{32} - 1] \to \mathcal{B}^s$ with $s \in \{1, 2, 4, 8\}$ #bytes to read

Implicit Wrapping

Relationship between *Poly* variables and process memory state

Example: let x be a `char` and $P(x) = [-1, 2]$.

Then we have 11111111_2, 00000000_2, 00000001_2, 00000010_2
or $\mathrm{bin}^{8s}(v)$ with $v \in [-1, 2]$ represented by a sequence of s bytes.

Remark: we can define $\mathrm{bits}_a^s : \mathbb{Z} \to \mathcal{P}(\Sigma)$ for all stores of $8s$ bits at address $a = addr(x)$ corresponding to $v \in P(x)$.

$$\mathrm{bits}_a^s(v) = \{(r_{8\cdot 2^{32}} \ldots r_{8(a+s)}) \| \mathrm{bin}^{8s}(v) \| (r_{8a-1} \ldots r_0))\}$$

This considers only the lower $8s$ bits of v; $\mathrm{bits}_a^1(0) = \mathrm{bits}_a^1(256)$.

For values $(v_1, \ldots, v_n) \in \mathbb{Z}^n$ we have variables $(x_1, \ldots, x_n)$ leading to stores $\bigcap_{i\in[1,n]} \mathrm{bits}_{a_i}^{s_i}(v_i)$ where a_i is the address of x_i and s_i is the store size in bytes.

The polyhedron P is then a set of stores $\gamma_a^s : Poly \to \mathcal{P}(\Sigma)$

$$\gamma_a^s(P) = \bigcup_{v \in P \cap \mathbb{Z}^n} \left(\bigcap_{i\in[1,n]} \mathrm{bits}_{a_i}^{s_i}(v_i) \right)$$

Implicit Wrapping: Set of Stores and Wraping

The polyhedron P is then a set of stores $\gamma_a^s : Poly \to \mathcal{P}(\Sigma)$

$$\gamma_a^s(P) = \bigcup_{v \in P \cap \mathbb{Z}^n} \left(\bigcap_{i \in [1,n]} \mathsf{bits}_{a_i}^{s_i}(v_i) \right)$$

- γ_a^s maps the abstract result to the actual wrapped result in the concrete process
- it gets us implicit wrapping
- the operator models without explicit checks for wrapping (overflows)
- a guard such as $x \leq y$ can not be modeled through $P \wedge_P [\![x \leq y]\!]$
- we need explicit wrapping

Example: Explicit Wrapping

Let $P = [\![x + 1024 = 8y, -64 \leq x \leq 448]\!]$ and the `uint8` variables x and y.

Suppose P feeds into the guard $x \leq y$.

Let $(x, y) = (384, 176) \in P$.

Given $\sigma \in \gamma_a^s(384, 176)$ implicit wrapping dictates that:

$$\mathsf{val}^{8,\mathsf{uint}}(\sigma^1(addr(x))) = 128 \qquad \mathsf{val}^{8,\mathsf{uint}}(\sigma^1(addr(y))) = 176$$

which implies that $x \leq y$ is true when x, y are `uint8` in σ.

Example: Explicit Wrapping

Let $P = [\![x + 1024 = 8y, -64 \leq x \leq 448]\!]$ and the `uint8` variables x and y.

Suppose P feeds into the guard $x \leq y$.

Let $(x, y) = (384, 176) \in P$.

Given $\sigma \in \gamma_a^s(384, 176)$ implicit wrapping dictates that:

$$\mathsf{val}^{8,\mathsf{uint}}(\sigma^1(addr(x))) = 128 \qquad \mathsf{val}^{8,\mathsf{uint}}(\sigma^1(addr(y))) = 176$$

which implies that $x \leq y$ is true when x, y are `uint8` in σ.

But notice that $(384, 176) \wedge_P [\![x \leq y]\!] = \emptyset$!

Example: Explicit Wrapping

Let $P = \llbracket x + 1024 = 8y, -64 \leq x \leq 448 \rrbracket$ and the `uint8` variables x and y.

Suppose P feeds into the guard $x \leq y$.

Let $(x, y) = (384, 176) \in P$.

Given $\sigma \in \gamma_a^s(384, 176)$ implicit wrapping dictates that:

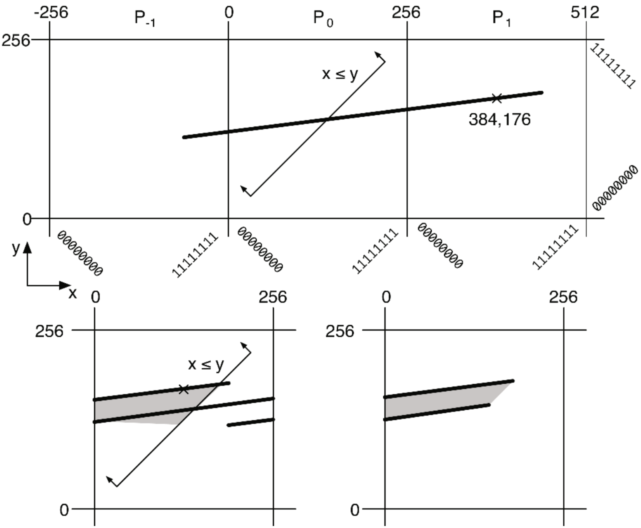
$$\mathsf{val}^{8,\mathsf{uint}}(\sigma^1(addr(x))) = 128 \qquad \mathsf{val}^{8,\mathsf{uint}}(\sigma^1(addr(y))) = 176$$

which implies that $x \leq y$ is true when x, y are `uint8` in σ.

But notice that $(384, 176) \wedge_P \llbracket x \leq y \rrbracket = \emptyset$!

This shows that it is not correct to model the guard as $P \wedge_P \llbracket x \leq y \rrbracket$.

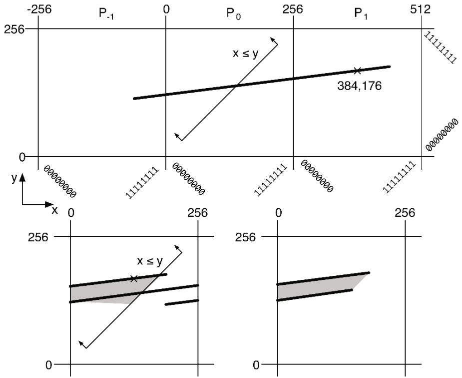
Explicit Wrapping



Source: A. Simon, Value Range Analysis of C Programs, 2009

- x range overflows on the two neighbouring quadrants

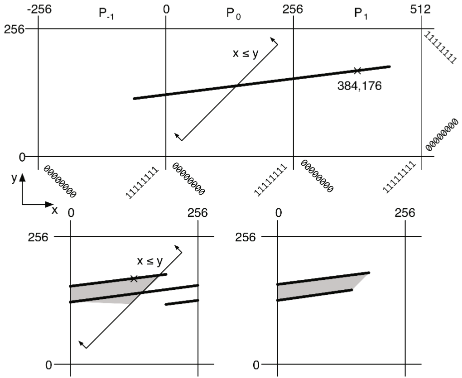
Explicit Wrapping



- x range overflows on the two neighboring quadrants
- partition P
- $P_{-1} = P \wedge_P [\![-256 \leq x \leq -1]\!]$
- $P_0 = P \wedge_P [\![0 \leq x \leq 255]\!]$
- $P_1 = P \wedge_P [\![256 \leq x \leq 511]\!]$
- translate by 256 units P_{-1} and P_1 towards P_0
- gray region is $P' \wedge_P [\![x \leq y]\!]$

$$P' = (P_0 \vee_P (P_{-1} \triangleright x := x + 256) \vee_P (P_1 \triangleright x := x - 256)) \vee_P [\![x \leq y]\!])$$

Explicit Wrapping



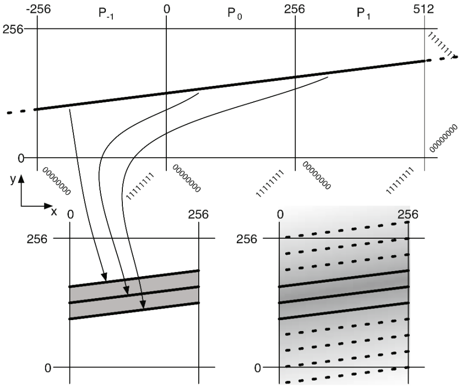
- x range overflows on the two neighboring quadrants
- partition P
- $P_{-1} = P \wedge_P [\![-256 \leq x \leq -1]\!]$
- $P_0 = P \wedge_P [\![0 \leq x \leq 255]\!]$
- $P_1 = P \wedge_P [\![256 \leq x \leq 511]\!]$
- translate by 256 units P_{-1} and P_1 towards P_0
- gray region is $P' \wedge_P [\![x \leq y]\!]$

$$P' = (P_0 \vee_P (P_{-1} \rhd x := x + 256) \vee_P (P_1 \rhd x := x - 256)) \vee_P [\![x \leq y]\!])$$

Or more precise P'':

$$(P_0 \wedge_P [\![x \leq y]\!]) \vee_P ((P_{-1} \rhd x := x+256) \wedge_P [\![x \leq y]\!]) \vee_P ((P_1 \rhd x := x-256) \wedge_P [\![x \leq y]\!])$$

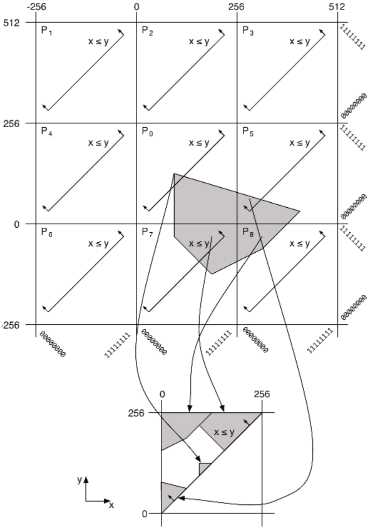
Infinite Wrapping



Source: A. Simon, Value Range Analysis of C Programs, 2009

- depicts $P = [\![x + 1024 = 8y]\!]$
- in general we do not have only 3 quadrants
- wrapping can require infinite join of state spaces
- $P_i = (P \triangleright x := x + i \cdot 2^8 \wedge_P [\![0 \leq x \leq 255]\!]) \vee_P (P \triangleright x := x - i \cdot 2^8 \wedge_P [\![0 \leq x \leq 255]\!])$
- right figure is equivalent to full type range: $\exists_x(P) \wedge_p [\![0 \leq x \leq 255]\!]$

Precise Wrapping of Two Variables



Source: A. Simon, Value Range Analysis of C Programs, 2009

Wrapping Algorithm

Algorithm 1 Explicitly wrapping an expression to the range of a type.

procedure $wrap(P, t\ s, x)$ where $P \neq \emptyset, t \in \{\mathbf{uint}, \mathbf{int}\}$ and $s \in \{1, 2, 4, 8\}$

$b_l \leftarrow 0$

$b_h \leftarrow 2^s$

if $t = \mathbf{int}$ **then** /* *Adjust ranges when wrapping to a signed type.* */

  $b_l \leftarrow b_l - 2^{s-1}$

  $b_h \leftarrow b_h - 2^{s-1}$

end if

$[l, u] \leftarrow P(x)$

if $l \neq -\infty \wedge u \neq \infty$ **then** /* *Calculate quadrant indices.* */

  $q_l \leftarrow \lfloor (l - b_l)/2^s \rfloor$

  $q_u \leftarrow \lfloor (u - b_l)/2^s \rfloor$

end if

if $l = -\infty \vee u = \infty \vee (q_u - q_l) > k$ **then** /* *Set to full range.* */

  return $\exists_x(P) \sqcap_P [\![b_l \leq x < b_h]\!]$

else /* *Shift and join quadrants* $\{q_l, \ldots q_u\}$. */

  return $\bigsqcup_{q \in [q_l, q_u]}((P \rhd x := x - q2^s) \sqcap_P [\![b_l \leq x < b_h]\!])$

end if

Source: A. Simon, Value Range Analysis of C Programs, 2009