

Multi-Interval Domain for cXprop

Kevin Atkinson

Overview

- Created a New Domain for cXprop
- cXprop stands for conditional X propagation.
- It is a whole program context insensitive analysis
- My domain is a combination of the Value Set Domain and the Interval Domain
- The idea is to convert multiple values into intervals instead of going to bottom

Benefits of the Interval Domain

- Can be very useful for eliminating redundant bound checks. For example the interval domain should be able to eliminate the two checks in the code below:

```
int x[100]
for (int i = 0; i < 100; ++i) {
    assert(i >= 0 && i < 100)
    x[i] = i;
}
```

Implementation

- Maintain a fixed number of intervals
- When there are more intervals than can fit combine one or more based on some heuristic

Interval Merging Heuristic Used

- Heuristic Used:
 - Preserve information about 0 if at all possible
 - Combine intervals which are closest to each other.
 - For example, in the set $\{[-5,-1], [1, 5], [8,10], [20,30]\}$
 - The intervals $[1,5]$ and $[8, 10]$ will be combined since their distance is 3, $[-5,-1]$, $[1, 5]$ are closer (with distance 2) but combining them will lose the fact that 0 can't be a possible value

Also Expand Intervals

- In addition to combining Intervals, also split small intervals into individual values if it can be done within the the fixed set size.
- For example $\{[1,5]\}$ will become $\{1,2,3,4,5\}$.
- Beneficial under certain operations such as multiplication For example $\{[1,5]\} * 2 = \{[2,10]\}$ while $\{1,2,3,4,5\} * 2 = \{2,4,6,8,10\}$

Bottom Not A Special Case

- Inside the transfer functions Bottom is represented as the interval containing the maximum range for the type.
- Some operations such as '%', '/', and backward comparisons functions can benefit from this.
- For example $\text{Bottom \% } 10 = [0,9]$

Transfer Functions

- Basic steps used in transfer functions
 - 1) Convert Bottom to Interval
 - 2) Apply Transfer function
 - 3) Fix Range of Intervals so that are in the range for the type
 - 4) Normalize resulting intervals, for example $\{[1,5], [4,8]\}$ becomes $\{[1,8]\}$
 - 5) Possible combine intervals if set size is to big, or expand small intervals if possible.
 - 6) If necessary convert back to bottom

Handling Overflow

- In order to avoid having to deal with overflow inside my domain I use arbitrary precision integers.
- Each interval is checked against the range of the type.
- For unsigned types, apply the mod transfer function, to handle out of range conditions.
- For signed integers, go to bottom if out of range

Unary Transfer Functions

- Apply unary operator to each interval in set and then combine.
- $-[x, y] = [-y, -x]$
- $\sim[x, y] = [\sim y, \sim x]$

Binary Transfer Functions

- Apply binary operator to all possible pairs and then combine.
- $[x, y] + [x', y'] = [x + x', y + y']$
- $[x, y] - [x', y'] = [x - y', y - x']$ (*x' and y' swapped*)
- $[x, y] * [x', y'] = [\min(xx', xy', yx', yy'), \max(\dots)]$
- $[x, y] / [x', y'] = [\min(x/x', x/y', y/x', y/y'), \max(\dots)]$
when 0 is not in $[x', y']$

More Binary Transfer Functions

- '%' is complicated
- $x \ll y = x / 2^y$
- $x \gg y = x * 2^y$
- Use machine generated transfer functions from the interval domain for '&', '^', and '|'.
- Maximum precise for all but '%'.

Comparisons

- Only have to implement two operators '==' and '<' since all others can be written in terms of those.
- Can never determine if two interval sets are equal unless each set contains a single element.
- Two sets are definitely not equal if the intersection is the empty set.

Less Than

- For '<' consider the set as one big interval, that is [minimal element in set, maximum element in set]
- If the two meta intervals don't overlap than can return a definite answer.
- Otherwise return bottom except for one special case when we can return false.
- Example of special case: $[5, 6] < [4, 5]$

Backwards Transfer Functions

- Backwards Transfer Functions can be used to learn something about the domain. For example consider this code:
 - `if (x < 10) {if (x > 100) {/* dead code*/} ...}`
- Implementation is tricky and took several tries to get right.

Widening

- Since the interval domain has a very tall lattice. Without some sort of forced widening the analysis will take forever.
- Original cXprop code widened just about everywhere, which made the interval domain pretty useless.
- Nathan greatly improved this. Which helped, but still lots of room for improvement.
- Main area of improvement is to do something other than just going to Bottom when widen is called.

Widening Cont.

- For example consider again the following code

```
int x[100]
for (int i = 0; i < 100; ++i) {
    assert(i >= 0 && i < 100)
    x[i] = i;
}
```

- In theory the assert should be eliminated, and with out widening it can be.
- However, due to widening only the “i < 100” check can be eliminated.

Widen Implementation

- The current implementations uses the simple heuristic of going to Bottom if any interval size is larger than a predetermined value.
- Can do better. One idea is to track the direction the domain changes for example in the previous example the interval will only grow in the positive direction. Than when widen is called first expand the interval only in one direction. When than fails go to bottom.

Other Widen Idea

- Another Idea is to store a set of hints on possible values to widen two with the domain. For example in the following code:

```
int main() {  
    int x = 0;  
    while (x < 2000) {  
        x++;  
    }  
    ... x ...  
}
```

- Store a hint with 'x' to widen to 2000.

Results

- When tested on a some TinyOS code it produced results no worse than the ValueSet domain.
- Unfortunately it isn't much better either. In fact in many cases the optimized code is identical to that using the Value Set Domain.
- Also It takes a 3-4 times more time to run.
- Note: The ValueSet maximum set size was 16 while mine was 8, with a max interval size of 16.

Results: Numbers

Test	Code Size (% smaller)	Execution Time (x longer)	Bits Known	
			VS	MI
ap	<i>same</i>	2.4	49.7	87.6
genericbase	<i>same</i>	4.2	73.8	74.5
hfs	1.20%	4.9	63.8	66.0
ident	<i>same</i>	4.1	70.9	71.5
osc	0.94%	3.1	70.3	70.8
rfmtoleds	0.00%	4.2	70.0	70.8
sensetorm	0.61%	4.2	69.1	69.8
surge	0.39%	4.0	74.8	75.5
testtimestamping	<i>same</i>	4.6	66.7	67.4
tinydb	0.15%	3.9	---	---

So Now What

- The Tiny OS Code didn't seem very well suited to the Multi-Interval domain.
- But, what code is, maybe code with lots of unnecessary bound checks.
- So I looked at Deputy, unfortunately it's checks are all on pointer values, in addition to be rather convoluted
- And even if the checks were integer bound checks, as previously stated I would only be able to get a few of the cases.

Possible Reasons For Poor Performance

- At least in theory, my domain *should* take longer since it is more aggressive. Thus cXprop needs to do more work to reach a fixed point.
- However, it could also be that my domain's transfer functions are just slow.
- Transfer functions not written with speed in mine.
- Use of arbitrary precession integers may have something to do with it.

Future Work

- Implement more careful use of Widening
- Find examples which are well suited to the interval domain.

What Type Of Analysis Will Help With This Code?

```
int x = 0;
int i = 0;

while (x < 100) {
    if (i >= 100) {
        printf("This should be dead code");
    }
    i++;
    x++;
}
```