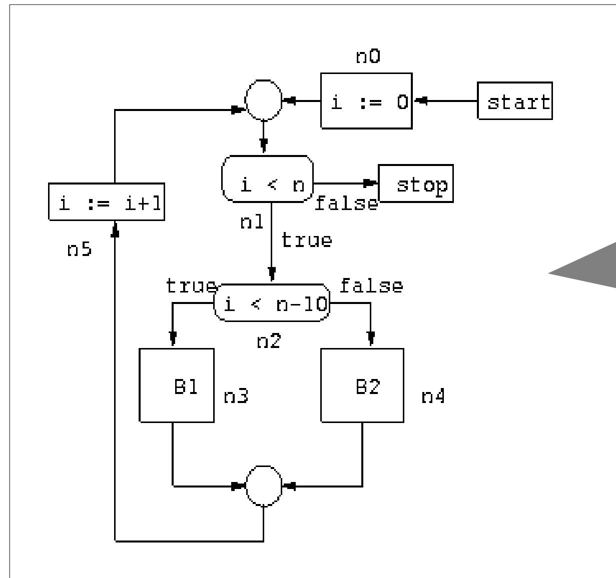
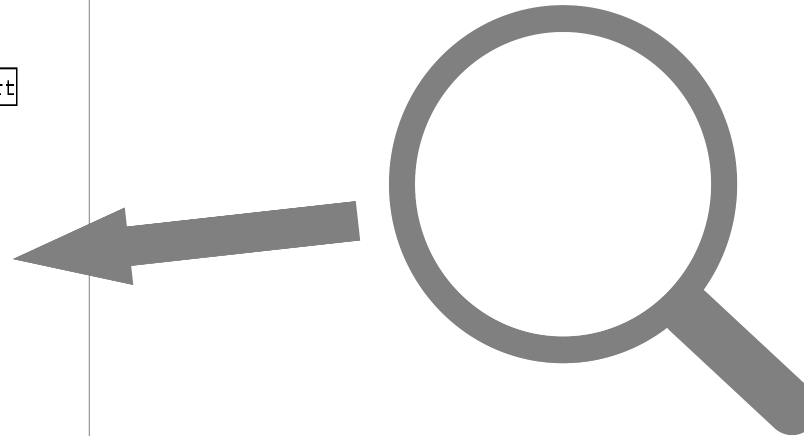
Static Program Analysis
Lecture 7: Tuning Value Analysis





Fine-Tuning Value Analysis

Getting the most out of a static program analysis tool can require turning some knobs

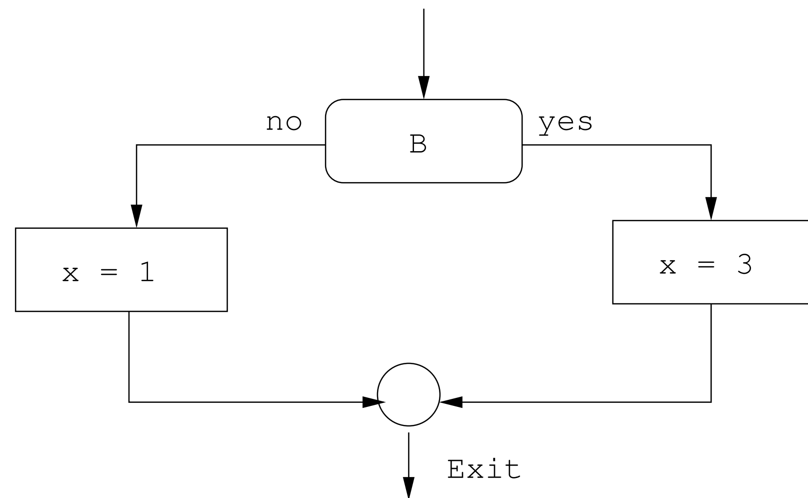
Find the right tradeoff precision vs. analysis time

We'll describe some ways to adjust the precision of value analysis:

- Selection of merge points
- Array smashing
- Adjusting the context-sensitivity
- Selection of abstract domain

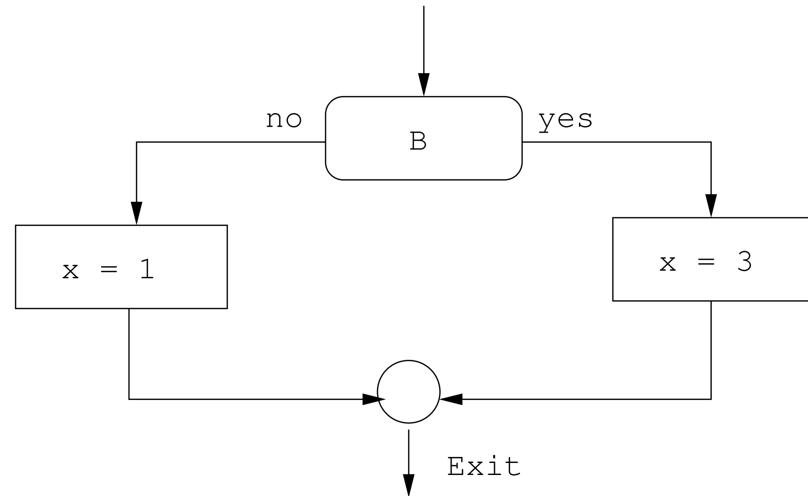
Selection of Merge Points

Merge at join nodes can cause imprecision:



Can we have $\mathtt{x} = 2$ at $\mathtt{Exit}$? Of course not. But merging abstract states at the join nodes yields the interval $[1, 3]$ for $\mathtt{x}$, indicating $\mathtt{x} = 2$ is possible

Selection of Merge Points (II)



Some value analysis tools allow turning off merge at some join nodes. This yields *two* abstract states at `Exit`: one with `x`: $[1, 1]$ and one with `x`: $[3, 3]$

This shows that $\mathtt{x} = 2$ is impossible, at the cost of handling several abstract states

Array Smashing

Array smashing: treat an array as a single variable rather than as one individual variable per element (`a[i]` $\rightarrow$ `a`)

Why do this?

1. Arrays can be large. Treating each element individually can give very large abstract states
2. Aliasing: `a[i]` may refer to different elements depending on the value of `i`. Can be hard to handle
3. Many array programs treat different array elements very much alike

An Example

Code with individual array elements:

```
s = 0;
for i = 0 to 99 do
  s = s + a[i];
```

Code after array smashing:

```
s = 0;
for i = 0 to 99 do
  s = s + a;
```

Typical candidate for array smashing. We're likely more interested in `i` not indexing outside the range of `a` than in the values of the elements of `a`

Context-Sensitivity

The ability to distinguish different abstract states for different *contexts*

A context can be, for instance:

- A loop iteration (or set of iterations)
- a specific function call

A context-sensitive analysis will be more precise, but at a higher cost

Finding the right tradeoff can be important

Semantic Loop Unrolling

To treat different loop iterations individually. Example:

```
int main()
{ static int no_init = 0;
  int i=0;
  float x=0.0, div=0.0;
  while (i<10) {
    if (no_init) { x+=x/div; }
    else { no_init = 1; x = 1.0; div = 2.0; }
    i++; }
}
```

Can we have division by zero?

Semantic Loop Unrolling (II)

A context-insensitive interval analysis yields `no_init`: $[0, 1]$, `div`: $[0.0, 2.0]$ just before the condition

Indicates that the true-branch can be taken with `div` $= 0.0$, possible division by zero!

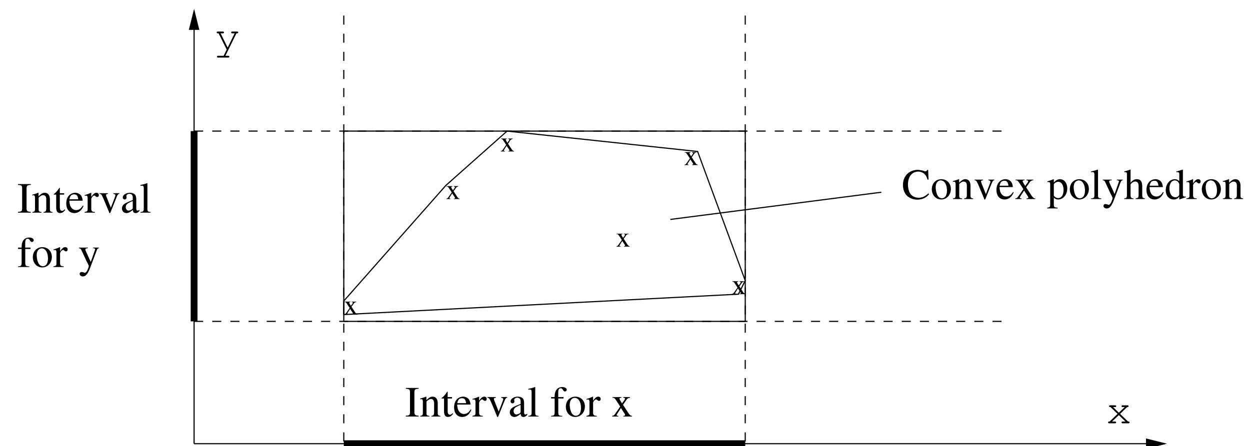
But this is a false positive! The true-branch can not be taken for the first iteration

Applying semantic loop unrolling to distinguish the first iteration from the rest will yield `no_init`: $[0, 0]$, `div`: $[0.0, 0.0]$ for the first iteration and `no_init`: $[1, 1]$, `div`: $[2.0, 2.0]$ for the rest

This eliminates the false positive!

Selection of Abstract Domain

The interval domain is fast but somewhat imprecise:



Many other abstract domains, with different precision/analysis time tradeoffs

For instance, the *polyhedral* domain. More precise, but costly

Selecting the right one can be important

Wrapping up

Tuning the value analysis can be very important, to obtain few false positives with reasonable analysis time

We have reviewed a few ways to adjust this tradeoff

Most of them apply to static program analysis in general, not just value analysis

Making the best use of them requires some knowledge of static program analysis, as well as of the code to be analysed

Thus, static program analysis is an advanced tool for skilled users

This course module has aimed to take you some of the way to there