Lecture Notes on Abstract Interpretation

15-411: Compiler Design
André Platzer

Lecture 28

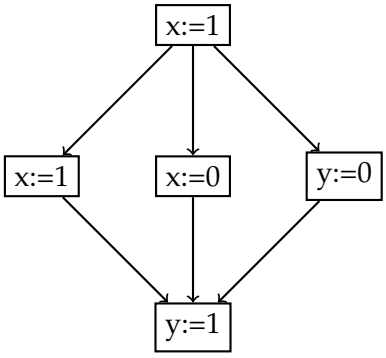
1 Introduction

More information on abstract interpretation can be found in [CC92, CC77, CC79] and [WM95, Chapter 10].

2 Abstract Interpretation

Abstract interpretation generalizes the theory of monotone frameworks and dataflow analysis to a general principle of analyzing programs by defining an abstract semantics for it [CC92, CC77, CC79, WM95]. In order to show the principle of abstract interpretation, without having to dig too much into the details, we consider an example where we abstractly interpret a program but still keep using monotone frameworks.

Suppose we want to check the property whether a variable x may be 0, which is a principle that can be useful for null pointer exception tests. As domain L for this we just choose the Boolean lattice $\{true, , false\}$. The operator $\bigsqcup$ is just logical disjunction ($\vee$). The flow relation is the forward control flow. Initialization is $false$, say. Transfer functions at the nodes make sense to choose from the constant functions $true$, $false$ and the identity function id.



By fixed-point iteration on the above example we find that $x = 1$ is possible after the program terminates. For a must analysis, instead, we would get that $x = 1$ is not necessary.

For multiple variables, we can choose a cartesian product $\{true, false\}^n$ of the Boolean lattice and use projections to coordinates as further transfer functions for copying the value for y over to x at a move $x := y$.

Another example is an abstract interpretation that performs general analysis for constant propagation. The property space has the form $\{x = \bot, x =?\} \cup \{x = v : v \in \mathbb{Z}\}$, where $\bot$ means is the bottom of the semilattice for undefined, $x =?$ means that x has nondeterministic values and $x = v$ for a number v means that we can be certain that x will always have value v at this program point. Let's look at an example. We initialize with no information ($\bot$) at all points, except the program init block, where we start with a nondeterministic initial value $i =?$:

```
{i=?,j=?,k=?}
i = 5; j = 0; k = 0;
{i=⊥,j=⊥,k=⊥}
while (j <= i) {
  {i=⊥,j=⊥,k=⊥}
  i = i + 2; k = k + j; j = j + 1
  {i=⊥,j=⊥,k=⊥}
  i = i − 2
  {i=⊥,j=⊥,k=⊥}
}
{i=⊥,j=⊥,k=⊥}
```

Now we can execute the first line in the abstract semantics and then enter the loop in the abstract semantics and execute the loop body once

```
{i=?,j=?,k=?}
```

```
i = 5; j = 0; k = 0;
{i=5,j=0,k=0}
while (j <= i) {
  {i=5,j=0,k=0}
  i = i + 2; k = k + j; j = j + 1
  {i=7,j=1,k=0}
  i = i − 2
  {i=5,j=1,k=0}
}
{i=⊥,j=⊥,k=⊥}
```

With those abstract values, we will repeat the loop, but we have to merge the previous information {i=5,j=0,k=0} with the current information {i=5,j=1,k=0} and find a joint representation in the property space lattice by the $\bigsqcup$ operator, giving {i=5,j=?,k=0}. Then we execute the loop body

```
{i=?,j=?,k=?}
i = 5; j = 0; k = 0;
{i=5,j=0,k=0}
while (j <= i) {
  {i=5,j=?,k=0}
  i = i + 2; k = k + j; j = j + 1
  {i=7,j=?,k=?}
  i = i − 2
  {i=5,j=?,k=?}
}
{i=⊥,j=⊥,k=⊥}
```

Again, merging the property values by the $\bigsqcup$ operator and executing the loop body gives

```
{i=?,j=?,k=?}
i = 5; j = 0; k = 0;
{i=5,j=0,k=0}
while (j <= i) {
  {i=5,j=?,k=?}
  i = i + 2; k = k + j; j = j + 1
  {i=7,j=?,k=?}
  i = i − 2
  {i=5,j=?,k=?}
}
{i=5,j=?,k=?}
```

Here the property value at the loop entry didn't change, so we can propagate to the loop exit and the analysis terminates. Now we know, as good as our abstract semantics could represent, what values the variables can have at the various program points.

3 Abstract Interpretation by Example

Consider the following simple program

```
0
1  x = 1
2
3  while (x<1000) {
4
5      x = x + 1
6
7  }
8
9  y = x
```

A run in the concrete semantics of the above program would start with the concrete state $x = \bot, y = \bot$ where the initial value of x, y in line 0 is unknown. The program would do 999 iterations through the loop after which it terminates with the state $y = x = 1000$. Concrete execution just does not help much for static analysis of programs in general, because we won't know the dynamic data until runtime.

Instead, let us consider an abstract run in an abstract semantics where variables take on intervals as values (due to Cousot and Cousot [CC77]):

$$L = \{[a, b] \ : \ a, b \in \mathbb{N} \cup \{+\infty, -\infty\}\}$$

To unify notation, we write $[-\infty, 5]$ for the left-open interval $(-\infty, 5]$ here. Now a run of the above program in the interval abstract domain gives after 1 iteration

```
0  {x = [−∞, ∞], y = [−∞, ∞]}
1  x = 1
2  {x = [1, 1], y = [−∞, ∞]}
3  while (x<1000) {
4      {x = [1, 1], y = [−∞, ∞]}
5      x = x + 1
6      {x = [2, 2], y = [−∞, ∞]}
```

```
7  }
8
9  y = x
```

and after 2 iterations

```
0  {x = [−∞, ∞], y = [−∞, ∞]}
1  x = 1
2  {x = [1, 1], y = [−∞, ∞]}
3  while (x<1000) {
4        {x = [1, 2], y = [−∞, ∞]}
5        x = x + 1
6        {x = [2, 3], y = [−∞, ∞]}
7  }
8
9  y = x
```

and after 3 iterations

```
0  {x = [−∞, ∞], y = [−∞, ∞]}
1  x = 1
2  {x = [1, 1], y = [−∞, ∞]}
3  while (x<1000) {
4        {x = [1, 3], y = [−∞, ∞]}
5        x = x + 1
6        {x = [2, 4], y = [−∞, ∞]}
7  }
8
9  y = x
```

We could keep on iterating, but this takes an awfully large number of iterations to figure out, since the loop count is 1000. If the bound is not computable statically, we do not even know how often to iterate. But we can iterate until we reach a fixedpoint. And we can also speed up convergence by jumping ahead in the lattice using a widening operator $\nabla : L \times L \to L$. For intervals let us jump ahead to $\pm\infty$ whenever our interval bounds are not inclusive:

$$[a, b] \nabla [a', b'] := \left[\begin{cases} a & \text{if } a \leq a' \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b' \leq b \\ +\infty & \text{otherwise} \end{cases} \right]$$

So in the 4th iteration, instead of doing a standard iteration, let us widening for computing line 4 from the previous two values $[1, 3] \nabla [1, 4]$:

```
0  {x = [−∞, ∞], y = [−∞, ∞]}
1  x = 1
2  {x = [1, 1], y = [−∞, ∞]}
3  while (x<1000) {
4        {x = [1, ∞], y = [−∞, ∞]}
5        x = x + 1
6        {x = [2, ∞], y = [−∞, ∞]}
7  }
8
9  y = x
```

In iteration 5, we obtain precise information by intersection with the guards

```
0  {x = [−∞, ∞], y = [−∞, ∞]}
1  x = 1
2  {x = [1, 1], y = [−∞, ∞]}
3  while (x<1000) {
4        {x = [1, 999], y = [−∞, ∞] since x = [1, ∞] ∩ [1, 999] = [1, 999]}
5        x = x + 1
6        {x = [2, 1000], y = [−∞, ∞]}
7  }
8  {x = [1000, 1000], y = [−∞, ∞] since x = [2, 1000] ∩ [1000, ∞] = [1000, 1000]}
9  y = x
10 {x = [1000, 1000], y = [1000, 1000]}
```

What we want the widening operator ∇ to satisfy is that it is like a union ($\cup$) but could be a bigger element of the lattice:

$$x \leq x \nabla y \quad y \leq x \nabla y$$

We also want iterated uses of the widening operator to become a fixedpoint eventually. That is

$$x_0 \nabla x_1 \nabla x_2 \nabla x_3 \nabla \ldots$$

is a finite sequence, for any $x_i \in L$.

When widening was too aggressive, a dual operator called narrowing $\Delta : L \times L \to L$ can be used as well. It is supposed to be like an intersection ($\cap$) but could be bigger:

$$x \cap y \leq x \Delta y$$

We also want iterated uses of the widening operator to become a fixedpoint eventually. That is

$$x_0 \Delta x_1 \Delta x_2 \Delta x_3 \Delta \ldots$$

is a finite sequence, for any $x_i \in L$.

This seems very powerful and it is, as a framework for static program analysis. The particular abstract domain of intervals alone, however, is insufficient. A simple variation of the above example shows that the example is misleading and real programs more complicated:

```
0  {x = [−∞, ∞], y = [−∞, ∞]}
1  x = 1
2  {x = [1, 1], y = [−∞, ∞]}
3  y = 1
4  {x = [1, 1], y = [1, 1]}
5  while (x<1000) {
6        {x = [1, 999], y = [1, ∞] since x = [1, ∞] ∩ [1, 999] = [1, 999]}
7        x = x + 1
8        {x = [2, 1000], y = [1, ∞]}
9        y = y + 1
10       {x = [2, 1000], y = [2, ∞]}
11 }
12 {x = [1000, 1000], y = [1, ∞] since x = [2, 1000] ∩ [1000, ∞] = [1000, 1000]}
```

But the abstract interpretation framework still applies. Abstract domains that can handle the above example need correlations of variables, i.e, they need to capture variable correlations like $0 \leq y - x \leq 1$. Difference-bounds matrix [Min01] are a fast abstract domain for this purpose. General convex polyhedra can be useful too. This is possible but out of scope for this lecture.

References

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13(2&3):103–179, 1992.

[Min01] Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In Olivier Danvy and Andrzej Filinski, editors, *PADO*, volume 2053, pages 155–172. Springer, 2001.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.