# Lecture Notes on Abstract Interpretation

15-411: Compiler Design
André Platzer

Lecture 28

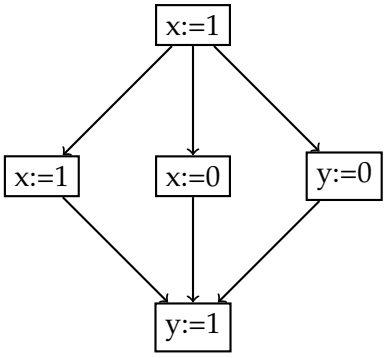
## 1 Introduction

More information on abstract interpretation can be found in [CC92, CC77, CC79] and [WM95, Chapter 10]. Simple examples of abstract interpretation type ideas in more classical situations include sign abstraction of values into $\{-, 0, +, ?\}$ or abstraction of values by remainders mod 4 [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.

## 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.

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