

Widening as Abstract Domain

Bogdan Mihaila, Alexander Sepp and Axel Simon

Technical University Munich, Germany

May 23, 2013

$$x \nabla y$$

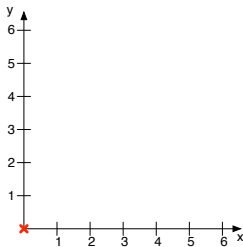
Widening in Abstract Interpretation

Static program analysis:

- ▶ use abstract domains to represent program states
- ▶ execute abstract semantics of program statements
- ▶ compute a fixpoint that over-approximates all possible program behaviors

```
1 int x = y = 0;  
2 while (x < 6) {  
3   p_0:  
4     x = x + 1;  
5     y = y + 1;  
6 }  
7 p_1:
```

State at p_0 :



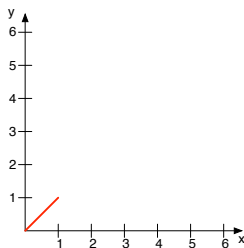
Widening in Abstract Interpretation

Static program analysis:

- ▶ use abstract domains to represent program states
- ▶ execute abstract semantics of program statements
- ▶ compute a fixpoint that over-approximates all possible program behaviors

```
1 int x = y = 0;  
2 while (x < 6) {  
3   p_0:  
4     x = x + 1;  
5     y = y + 1;  
6 }  
7 p_1:
```

State at p_0 :



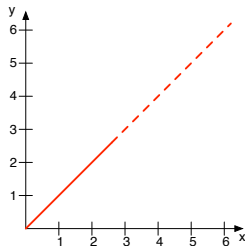
Widening in Abstract Interpretation

Static program analysis:

- ▶ use abstract domains to represent program states
- ▶ execute abstract semantics of program statements
- ▶ compute a fixpoint that over-approximates all possible program behaviors

```
1 int x = y = 0;  
2 while (x < 6) {  
3   p_0:  
4     x = x + 1;  
5     y = y + 1;  
6 }  
7 p_1:
```

State at p_0: (widened)



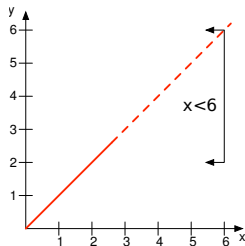
Widening in Abstract Interpretation

Static program analysis:

- ▶ use abstract domains to represent program states
- ▶ execute abstract semantics of program statements
- ▶ compute a fixpoint that over-approximates all possible program behaviors

```
1 int x = y = 0;  
2 while (x < 6) {  
3   p_0:  
4     x = x + 1;  
5     y = y + 1;  
6 }  
7 p_1:
```

State at p_0: (narrowed)



Widening in Abstract Interpretation

Idea of widening:

- ▶ some domains have infinite ascending chains:
[0, 0] [0, 1] [0, 2] ...
- ▶ widening is needed for *termination*

Widening in Abstract Interpretation

Idea of widening:

- ▶ some domains have infinite ascending chains:
[0, 0] [0, 1] [0, 2] ...
- ▶ widening is needed for *termination*

Definition:

Given a domain \mathcal{D} , define $\nabla : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$ such that $\forall x, y \in \mathcal{D}$:

$$x \sqsubseteq x \nabla y \quad \text{and} \quad y \sqsubseteq x \nabla y$$

and for all increasing chains $x_0 \sqsubseteq x_1 \sqsubseteq \dots$ the increasing chain $y_0 = x_0, \dots, y_{i+1} = y_i \nabla x_{i+1}$ is eventually stable.

Widening in Abstract Interpretation

Idea of widening:

- ▶ some domains have infinite ascending chains:
[0, 0] [0, 1] [0, 2] ...
- ▶ widening is needed for *termination*

Definition:

Given a domain \mathcal{D} , define $\nabla : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$ such that $\forall x, y \in \mathcal{D}$:

$$x \sqsubseteq x \nabla y \quad \text{and} \quad y \sqsubseteq x \nabla y$$

and for all increasing chains $x_0 \sqsubseteq x_1 \sqsubseteq \dots$ the increasing chain $y_0 = x_0, \dots, y_{i+1} = y_i \nabla x_{i+1}$ is eventually stable.

- ▶ widening *seems* to require a modified fixpoint computation
- ▶ cannot easily adapt widening strategies

Properties of Narrowing

Narrowing is often required after widening:

- ▶ widening introduces imprecision by overshooting the fixpoint
- ▶ *narrowing* can sometimes recover precision
- ▶ here: 2nd iter. $p_0 : x = y, x \in [0, \infty]$; $p_1 : x = y, x \in [6, \infty]$
3rd iter. $p_0 : x = y, x \in [0, 5]$; $p_1 : x = y, x \in [6, 6]$

```
1 int x = y = 0;
2 while (x < 6) {
3   p_0:
4     x = x + 1;
5     y = y + 1;
6 }
7 p_1:
```

Properties of Narrowing

Narrowing is often required after widening:

- ▶ widening introduces imprecision by overshooting the fixpoint
- ▶ *narrowing* can sometimes recover precision
- ▶ here: 2nd iter. $p_0 : x = y, x \in [0, \infty]$; $p_1 : x = y, x \in [6, \infty]$
3rd iter. $p_0 : x = y, x \in [0, 5]$; $p_1 : x = y, x \in [6, 6]$

Problems:

```
1 int x = y = 0;
2 while (x < 6) {
3   p_0:
4     x = x + 1;
5     y = y + 1;
6 }
7 p_1:
```

- ▶ need to refine states on all exit points of the loop
- ▶ what if the program contains `goto p_1` ?
- ▶ alternative: avoid propagating to p_1 until loop is stable
- ▶ complicates fixpoint engine and state management

Widening on Low Level Code

We analyze machine code:

- ▶ Control-Flow Graph (CFG) is reconstructed on-the-fly
- ▶ → loop entries and exits not known up front
- ▶ possibly irreducible CFGs: no best set of widening points
- ▶ → we need a very robust widening
- ▶ → need to try other heuristics
- ▶ → avoid narrowing altogether

Widening on Low Level Code

We analyze machine code:

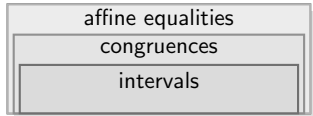
- ▶ Control-Flow Graph (CFG) is reconstructed on-the-fly
- ▶ → loop entries and exits not known up front
- ▶ possibly irreducible CFGs: no best set of widening points
- ▶ → we need a very robust widening
- ▶ → need to try other heuristics
- ▶ → avoid narrowing altogether

Our goal: keep fixpoint engine, implement widenings as plug-ins

Co-fibered Abstract Domains

A co-fibered domain $\langle \mathcal{D} \triangleright \mathcal{C}, \sqsubseteq_{\mathcal{D} \triangleright \mathcal{C}}, \sqcup_{\mathcal{D} \triangleright \mathcal{C}}, \sqcap_{\mathcal{D} \triangleright \mathcal{C}} \rangle$ tracks values of the form $\langle d, c \rangle \in \mathcal{D} \triangleright \mathcal{C}$ where:

- ▶ d is the internal information tracked by the domain itself
- ▶ c is the child domain
- ▶ all operations are defined on $\langle d, c \rangle$
- ▶ \rightarrow can execute multiple operations on the child or none at all
- ▶ can translate an operation on $\langle d, c \rangle$ into a different operation on the child
- ▶ example: congruence domain stores $x/4$ in child if x is multiple of 4



Widening as Co-fibered Domains

Idea:

implement widening + heuristics as co-fibered abstract domains.

Namely:

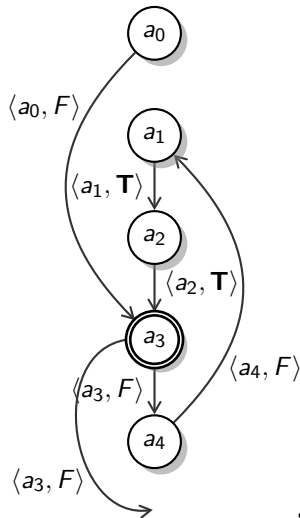
- ▶ \mathcal{W} : domain inferring widening points
- ▶ \mathcal{D} : delay widening domain
- ▶ \mathcal{T} : widening thresholds domain
- ▶ \mathcal{P} : guided static analysis domain



Finding Widening Points

Define domain $\mathcal{W} \triangleright \mathcal{C}$ where $\mathcal{W} = Lab \times \{T, F\}$ that applies widening instead of join on child \mathcal{C} .

- ▶ $l \in Lab$ is a program point and $f \in \{T, F\}$ is a Boolean flag
- ▶ for termination at least one widening point in each loop is needed
- ▶ use total order on the program points (instruction addresses) to detect back-edges
- ▶ simple heuristic: any back-edges is considered an edge to a loop head
- ▶ l is last program point, f is set if back-edge has been seen



Tracking Widening Thresholds

Define $\mathcal{T} \triangleright \mathcal{C}$ where $\mathcal{T} : Lab \times Pred \times \wp(Lab)$ that applies thresholds after widening to refine the state.

```
1 int x = y = 0;  
2 while (x < 6) {  
3   p_0:  
4     x = x + 1;  
5     y = y + 1;  
6 }  
7 p_1:
```

- ▶ $l \in Lab$ is the origin of test $p \in Pred$ and $a \in \wp(Lab)$ tracks application sites of p
- ▶ track redundant tests as *thresholds*
- ▶ thresholds are invariants for the current state (applying the test does not change the state)
- ▶ here $x < 6$ is a threshold at line 3
- ▶ thresholds are transformed by assignments, so that they stay invariant
- ▶ use thresholds after widening to immediately restrict the widened state

Tracking Widening Thresholds

```
1 int x = y = 0;
2 while (x < 6) {
3   p_0:
4     x = x + 1;
5     y = y + 1;
6 }
7 p_1:
```

- ▶ collect threshold from redundant test
3: $t = \langle 2, (x < 6), \{\} \rangle$
- ▶ transform thresholds with instructions
4: $t = \langle 2, (x < 6), \{\} \rangle$
5: $t = \langle 2, (x < 7), \{\} \rangle$
- ▶ apply thresholds only once per widening point (termination)
2': $t = \langle 2, (x < 7), \{\} \rangle$
3': $t = \langle 2, (x < 7), \{2\} \rangle$
 $\rightarrow p_0 : x=y, x \in [0, 5]; p_1 : x=y, x \in [6, 6]$
- ▶ when seeing a threshold again, keep the transformed one (termination)
- ▶ mark only the “smallest” thresholds that restrict widening (retain others)

No Widening after Constant Assignments

Define $\mathcal{D} \triangleright \mathcal{C}$ where $\mathcal{D} : \wp(\text{Lab})$ is a set of program points with constant assignments.

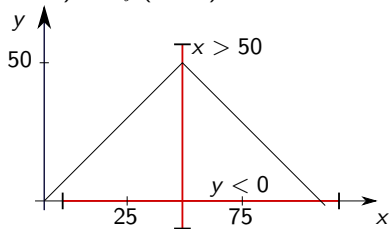
```
1 int x = 0;
2 int y = 0;
3 while (x < 100){
4     if (x > 5) {
5         y = 1;
6     }
7     x = x + 4;
8 }
```

- ▶ problem: widening of y yields $[0, 0] \nabla [1, 1] = [0, \infty]$
- ▶ common approach is to delay widening for the first n loop iterations (here: $n = 2$)
- ▶ slows down fixpoint computation unnecessarily if not needed
- ▶ better: do not widen if we have seen a new constant assignment
- ▶ we track program locations with constant assignments
- ▶ when widening $\mathcal{D} \triangleright \mathcal{C}$, compute a join on \mathcal{C} if there are new constant assignments

Guided Static Analysis as Abstract Domain

Define $\mathcal{P} \triangleright \mathcal{C}$ where $\mathcal{P} : \mathcal{C} \times (\text{Pred} \times \mathcal{P})^* \times \wp(\text{Pred})$.

```
1 int x = 0;
2 int y = 0;
3 while (true) {
4   if (x <= 50){
5     y++;
6   } else {
7     y--;
8   }
9   if (y < 0)
10    break;
11   x++;
12 }
```



- ▶ numeric domains usually are convex approximations
- ▶ \rightarrow precision loss when joining different states
- ▶ idea is to separate the states that belong to different *phases* of a loop to avoid convex approximation of widened states

Conclusion

- ▶ widening/narrowing is a challenge to implement for binary analysis
- ▶ especially to combine with interesting widening heuristics in the literature!
- ▶ co-fibered domains allow the modular combination of different strategies
- ▶ no adjustment to the fixpoint and state management necessary
- ▶ we successfully applied our domain stack to the problems in the literature
- ▶ our combined strategies were more efficient (fewer iterations) than the current state of the art