Automatic Inference of Access Permissions

Pietro Ferrara and Peter Müller

ETH Zurich
Switzerland

VMCAI 2012, Philadelphia, U.S.A.

Access permissions

- OO programs
- Modular reasoning
  > Design by Contracts
- Side effects
  > Problem for modular reasoning!
- Restrict this scenario:
  > Access a location iff we have the permission

```
class Coord {
  int x, y;
  //requires acc(x)
  //ensures acc(x)
  //ensures x==t
  void updateX(int t) {
    x=t;
  }
}
```

```
void test(Coord c) {
  c.x=1;
  c.y=1;
  c.updateX(-1);
  assert(c.y==1);
}
```

Permission Transfer

- Permissions may be transferred between
  - > methods
  - > threads
    - e.g., acquire/release
- A method should
  - > require what it needs
  - > give back what it owns
- Add perm.: inhale
- Remove perm.: exhale

```
class Coord {
  int x, y;
  //requires acc(x)
  //ensures acc(x)
  //ensures x==t
  void updateX(int t) {

  }
}
```

acc(c.x)

acc(c.x)

```
//requires acc(c.x)
//requires acc(c.y)
void test(Coord c) {
  c.x=1;
  c.y=1;
  c.updateX(-1);
  assert(c.y==1);
}
```

acc(c.x)&&acc(c.y)

acc(c.x)&&acc(c.y)

Permission Transfer

- Permissions may be transferred between
  > methods
  > threads
    - e.g., acquire/release
- A method should
  > require what it needs
  > give back what it owns
- Add perm.: inhale
- Remove perm.: exhale

```
class Coord {
  int x, y;
  //requires acc(x)
  //ensures acc(x)
  //ensures x==t
  void updateX(int t) {

  }
}
```

acc(c.x)

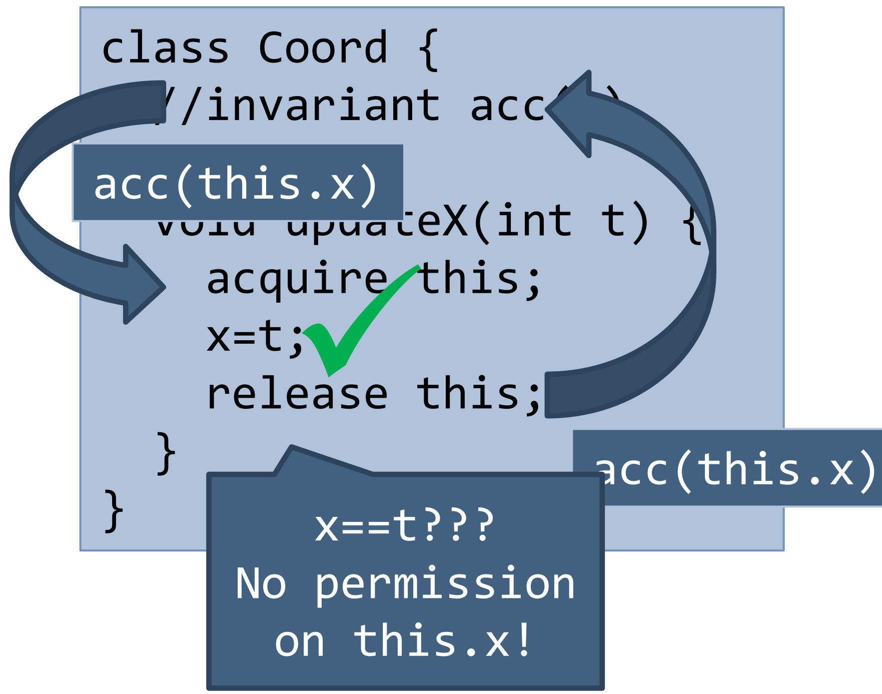
```
//requires acc(c.x)
//requires acc(c.y)
void test(Coord c) {
  c.x=1;
  c.y=1;
  fork c.updateX(-1);
  fork c.updateX(1);
}
```

acc(c.x)&&acc(c.y)

acc(c.y)

Permission Transfer

- Permissions may be transferred between
  - methods
  - threads
    - e.g., acquire/release
- A method should
  - require what it needs
  - give back what it owns
- Add perm.: inhale
- Remove perm.: exhale




```
void test(Coord c) {
  fork c.updateX(-1);
  fork c.updateX(1);
}
```

Fine-grained Permissions

- Full permission:
  - > Read&write access
- Partial permission:
  - > Read access
- No permission:
  - > No access
- Fine-grained:
  - > Fractional, counting, Chalice, …

```
class Coord {
  //invariant acc(x, 50%)
  int x, y;
  //requires acc(x, 50%)
  //ensures acc(x, 50%)
  //ensures x==t
  void updateX(int t) {
    acquire this;
    x=t;
    release this;
  }
}
```

this.x ↦ 50%
+50%

Read access on this.x

Motivations and goals

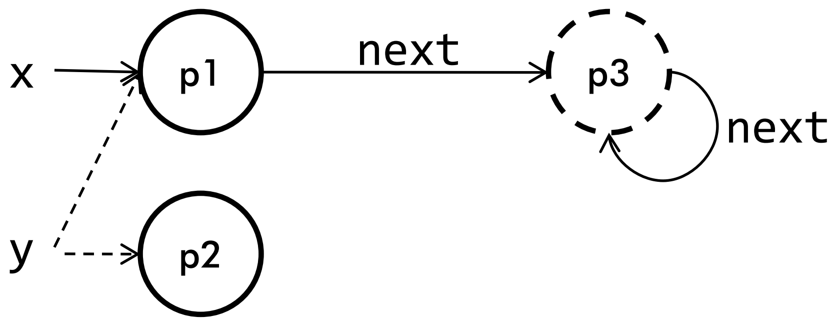
- Annotation overhead of permissions
  - > Quite verbose
  - > The code already contains all the accesses
- Start with a program without annotation
- Apply static analysis
  - > Based on abstract interpretation
  - > To infer the permissions that could be specified
    - Strong enough to perform the heap accesses
    - As weak as possible

Outline

1. Introduction
2. Symbolic permissions
3. Annotation inference
4. Experimental results & Conclusion

Heap Abstraction

- Not a contribution of this work
- Generic approach
  > Given a heap access, it returns a heap id



- Implemented a simple standard analysis
  > Abstract heap nodes with program points
  > Apply other heap analyses, e.g., TVLA

Symbolic Permissions

- Many ways to specify permissions
- Symbolic values
  - > Pre-conditions:
    - $Pre(\mathrm{C}, \mathrm{m}, \mathrm{p.f})$
    - Method `m` in class `C` over path `p.f`
  - > Post-conditions
  - > Monitor invariants
- Symbolic values: $\overline{\mathrm{SV}}$

```
class Coord {
  int x, y;
  void updateX(int t) {
    acquire this;
    x=t;
    release this;
  }
}
```

`this.x` $\mapsto$ Pre(`Coord`, `updateX`, `this.x`)

`this.y` $\mapsto$ Pre(`Coord`, `updateX`, `this.y`)

???

Precondition+Monitor invariant!

Symbolic Levels

- Inhale and exhale
  - > several times
  - > on the same location
- Sum of symbolic values
  - > At a given pp
  - > For each location
- Values:

$$\overline{\mathsf{AV}} = \{\sum_{\mathsf{i}} \mathsf{a_i} * \mathsf{s}_i + \mathsf{c} \text{ where } \mathsf{a_i}, \mathsf{c} \in \mathbb{Z}, \mathsf{s}_i \in \overline{\mathsf{SV}}\}$$

```
class Coord {
  int x, y;
  void updateX(int t) {
    acquire this;
    x=t;
    release this;
  }
}
```

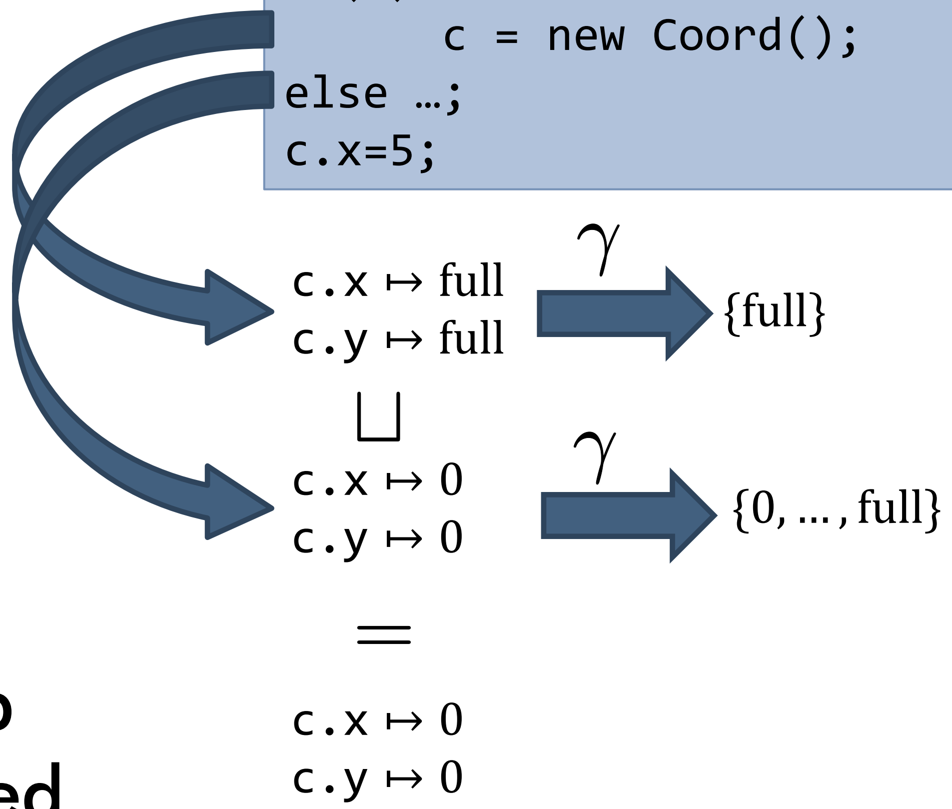
$\mathtt{this.x} \mapsto$
$\quad Pre(\mathtt{Coord}, \mathtt{updateX}, \mathtt{this.x})$
$\quad\quad + MI(\mathtt{Coord}, \mathtt{this.x})$

$\mathtt{this.y} \mapsto$
$\quad Pre(\mathtt{Coord}, \mathtt{updateX}, \mathtt{this.y})$
$\quad\quad + MI(\mathtt{Coord}, \mathtt{this.y})$

Lattice Structure

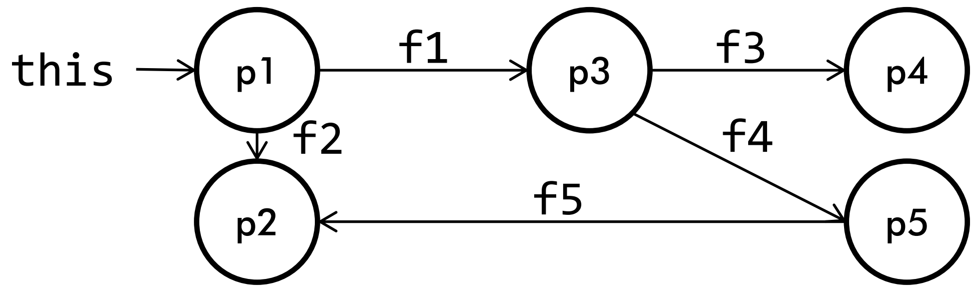
- **<u>Surely</u>** owned permissions
- Upper bound
  > Minimum
- Goal
  > Infer enough permissions to perform the heap accesses contained in the code

```
if(…)
        c = new Coord();
else …;
c.x=5;
```



What could be specified?

- Reachable($\overline{reach}$)
  1. Start from local variables
  2. Fields of added nodes
  3. Until we visit all reachable nodes
     - They are finite
     - We compute *one* of the shortest paths
  4. Rename the paths



1st step:
p1 reachable through `this`

2nd step:
p2 reachable through `this.f2`
p3 reachable through `this.f1`

3rd step:
p4 reachable through `this.f1.f3`
p5 reachable through `this.f1.f4`

Abstract Semantics

- Based on inhale (+) and exhale (-)
  - > Method call
    - Exhale precondition
    - Inhale postcondition
  - > Acquire a monitor
    - Inhale invariant
  - > Release a monitor
    - Exhale invariant
- Rely on $\overline{reach}$

```
class Coord {
  int x, y;
  void updateX(int t) {
    acquire this;
    x=t;
    release this;
  }
}
```

$$\texttt{this.x} \mapsto \mathit{Pre}(\texttt{Coord}, \texttt{updateX}, \texttt{this.x}) + \mathit{MI}(\texttt{Coord}, \texttt{this.x})$$

$$\texttt{this.y} \mapsto \mathit{Pre}(\texttt{Coord}, \texttt{updateX}, \texttt{this.y}) + \mathit{MI}(\texttt{Coord}, \texttt{this.y})$$

Outline

1. Introduction
2. Symbolic permissions
3. Annotation inference
4. Experimental results & Conclusion

Constraint Inference

- Heap accesses impose constraints:
  - Write: full
  - Read: non-zero
- Inhale: $\bar{s} \leq$ full
- Exhale: $\bar{s} \geq 0$
- $\forall \bar{s} \in \overline{SV} : 0 \leq \bar{s} \leq$ full
- Post(C,m,...)==exit

```
class Coord {
  int x, y;
  void updateX(int t) {
    acquire this;
    x=t;
    release this;
  }
}
```

$Pre(\text{Coord}, \text{updateX}, \text{this.\_}) + MI(\text{Coord}, \text{this.\_}) \leq \text{full}$

$Pre(\text{Coord}, \text{updateX}, \text{this.x}) + MI(\text{Coord}, \text{this.x}) == \text{full}$

$Pre(\text{Coord}, \text{updateX}, \text{this.\_}) \geq 0$

$Pre(\text{Coord}, \text{updateX}, \text{this.\_}) == Post(\text{Coord}, \text{updateX}, \text{this.\_})$

Permission Systems

- Various systems
  - > Fractional
  - > Counting
  - > Chalice
    - Combination

System	full	fract	infin	ensRd(p)
Fractional	1	✓	✗	p >0
Counting	MAX	✗	✗	p ≥ 1
Chalice	100	✓	✓	p $\geq \epsilon$

- Parameters
  - > Full perm.
  - > Fractional perm.
  - > Infinitesimal perm.
  - > Read perm.

- $Pre(\texttt{Coord}, \texttt{updateX}, \texttt{this.\_}) + MI(\texttt{Coord}, \texttt{this.\_}) \leq 1$
- $Pre(\texttt{Coord}, \texttt{updateX}, \texttt{this.x}) + MI(\texttt{Coord}, \texttt{this.x}) == 1$
- $Pre(\texttt{Coord}, \texttt{updateX}, \texttt{this.\_}) \geq 0$
- $Pre(\texttt{Coord}, \texttt{updateX}, \texttt{this.\_}) == Post(\texttt{Coord}, \texttt{updateX}, \texttt{this.\_})$

Linear Programming

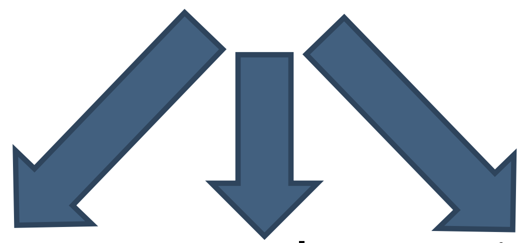
- Linear programming
  - > Solve our system
- Objective function:
  - > $\sum_i n_i * s_i$ where
    - n: integer coefficient
    - s: symbolic value
  - > Minimize it
    - Goal: permissions as weak as possible
  - > Higher coefficient $\Rightarrow$ lower priority

$$Pre(\texttt{Coord}, \texttt{updateX}, \texttt{this.\_}) + MI(\texttt{Coord}, \texttt{this.\_}) \leq 1$$

$$Pre(\texttt{Coord}, \texttt{updateX}, \texttt{this.x}) + MI(\texttt{Coord}, \texttt{this.x}) == 1$$

$$Pre(\texttt{Coord}, \texttt{updateX}, \texttt{this.\_}) \geq 0$$

$$Pre(\texttt{Coord}, \texttt{updateX}, \texttt{this.\_}) == Post(\texttt{Coord}, \texttt{updateX}, \texttt{this.\_})$$



Priority on preconditions	Equal priority	Priority on invariants
$Pre = 1$ $Post = 1$	$Pre = 0.5$ $Post = 0.5$ $MI = 0.5$	$MI = 1$
on `this.x`	on `this.x`	on `this.x`

Outline

1. Introduction
2. Symbolic permissions
3. Annotation inference
4. Experimental results & Conclusion

Experimental Results

- Implemented in Sample
  - > Generic static analyzer
- Intel Core 2 Quad CPU 2.83 Ghz
  - > 4 GB RAM
  - > Windows 7
- Fast
- Precise

Program	Time (msec)	% inf. contr.	
Fig1	45	100%	Chalice's tutorial
Fig2	12	100%	
Fig3	9	100%	
Fig4	3	100%	
Fig5	143	100%	
Fig6	53	100%	
Fig11	15	100%	
Fig12	15	100%	
Fig13	706	100%	
OwickiGries	164	100%	Chalice's distribution
Cell	115	100%	
Linkedlist	78	100%	
Swap	10	100%	
AssociationList	668	36%	
HandOverHand	564	36%	
Master	76	100%	Vericool
CellLib	148	100%	
CompositePattern	1217	71%	
Spouse	221	100%	Verifast
Account	12	100%	
Stack	76	67%	
Iterator	46	100%	

Conclusion

- Static analysis to infer symbolic permissions
- System of linear constraints
  - > Imposed by the semantics
- Solved using linear programming
  - > Many possible solutions
    - Priorities through the objective function
- Implemented
  - > Fast and precise