Inference of Fractional, Counting and Chalice Access Permissions via Abstract Interpretation

Pietro Ferrara and Peter Müller

ETH Zurich
Switzerland

Universidad Complutense de Madrid, Spain

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);
}
```

Behavioral subtyping

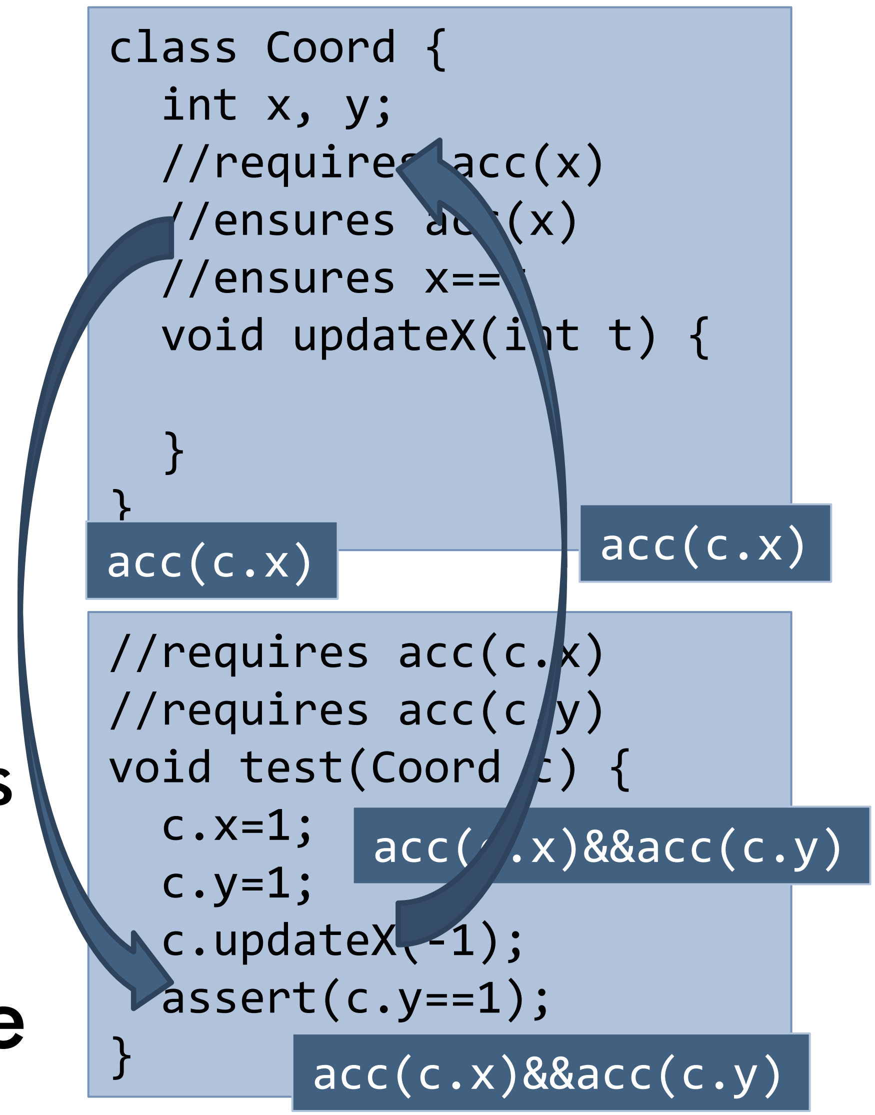
- Subtypes specialize the behavior of supertypes
  - > Weaker preconditions
    - Fewer accesses
  - > Stronger postconditions:
    - More accesses
- An overriding method
  - > Cannot access more locations
  - > Can provide access to more locations

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

```
class BadCoord ext Coord{
  //requires acc(x)
  //requires acc(y)
  //ensures acc(x)
  //ensures acc(y)
  void updateX(int t) {
    super(t);
    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



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)

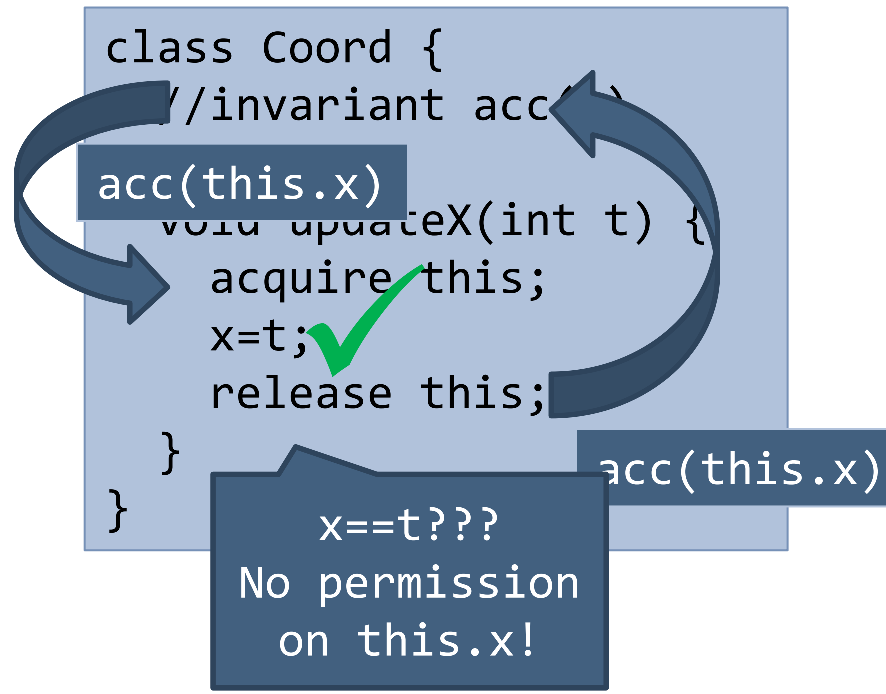
```
//requires acc(c.x)
//requires acc(c.y)
void test(Coord c) {
  c.x=1;
  c.y=1;
  fork c.updateX(-1);
  fork c.updateX(1);
}
```

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: inhale
- Remove: 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

Demo

Outline

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

Symbolic Permissions

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

```
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:

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

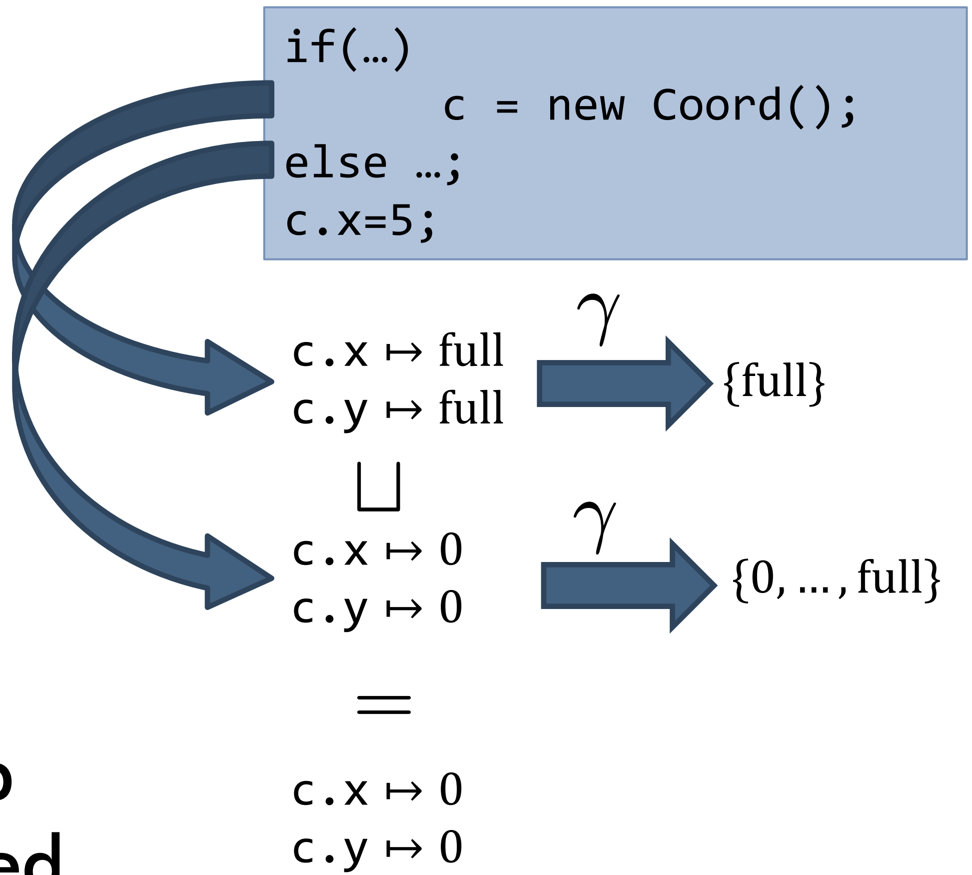
```
class Coord {
  int x, y;
  void updateX(int t) {
    acquire this;
    x=t;
    release this;
  }
}
```

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

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

Lattice Structure

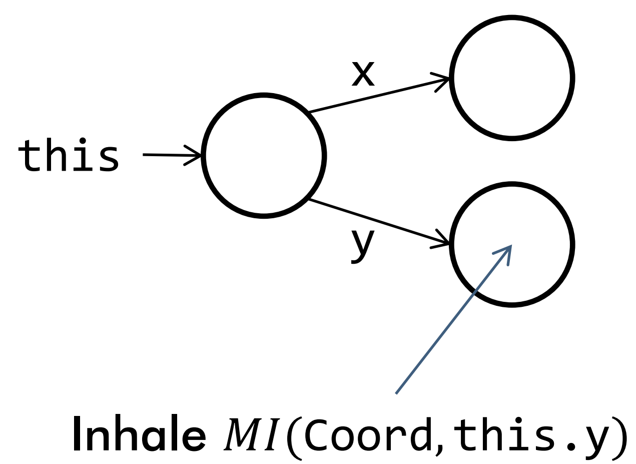
- **<u>Surely</u>** owned permissions
- Upper bound
  - > Minimum
- Goal
  - > Infer enough permissions to perform the heap accesses contained in the code



What could be specified?

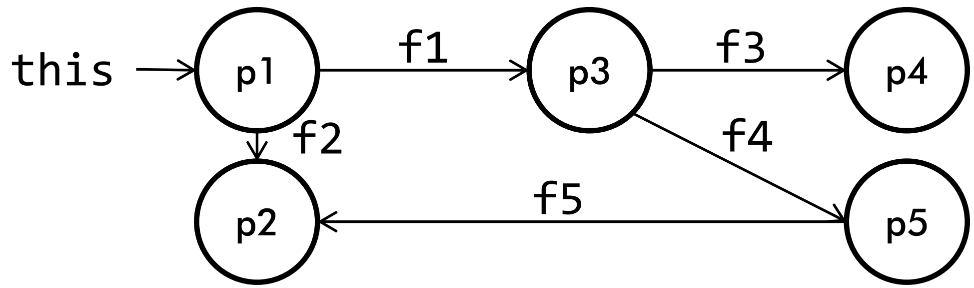
- Initial assumption
  - > No contracts
- Infer what permissions *could* be specified
  - > Anything reachable
    - On the abstract heap
- Collect the path to reach that location
  - > Build up a symbolic value for that

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



Algorithm

- 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;
  }
}
```

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

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: $\bar{s} \leq \text{full}$
- Exhale: $\bar{s} \geq 0$
- $\forall \bar{s} \in \overline{SV} : 0 \leq \bar{s} \leq \text{full}$
- Post(C,m,...)==exit

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

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

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

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

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

Permission Systems

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

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

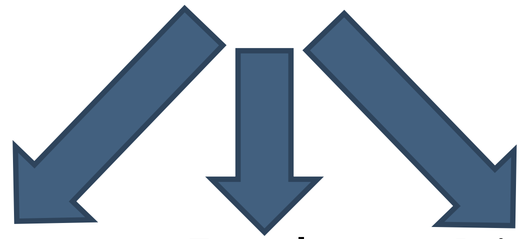
- Linear programming
  - > Solve our system
- Objective function:
  - > $\sum_i n_i * s_i$ where
    - n: integer coefficient
    - s: symbolic value
  - > Minimize it
    - Goal: permissions as weak as possible
  - > Higher coefficient $\Rightarrow$ lower priority

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

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

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

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

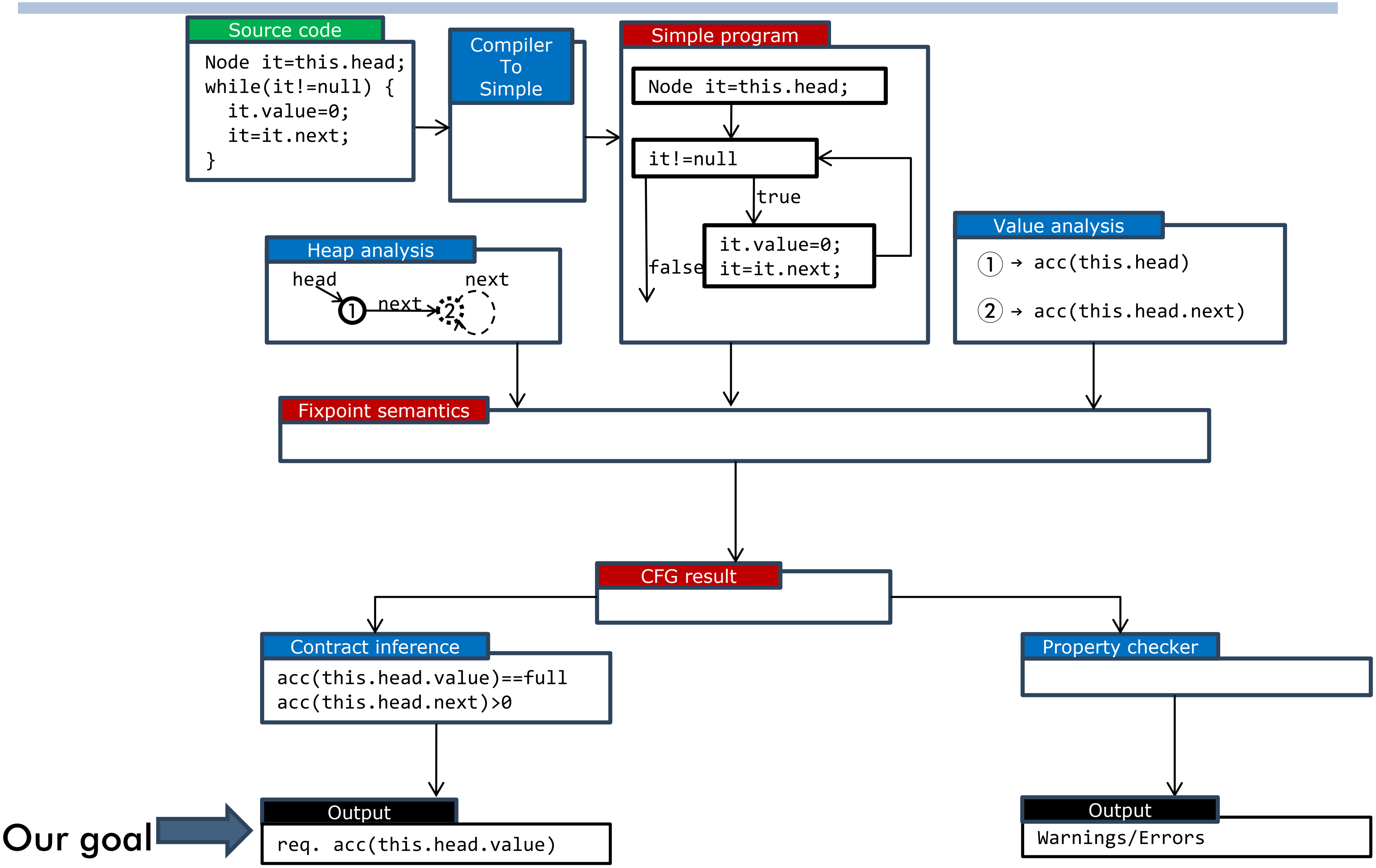


Priority on preconditions	Equal priority	Priority on invariants
$Pre = 1$ $Post = 1$	$Pre = 0.5$ $Post = 0.5$ $MI = 0.5$	$MI = 1$
on `this.x`	on `this.x`	on `this.x`

Outline

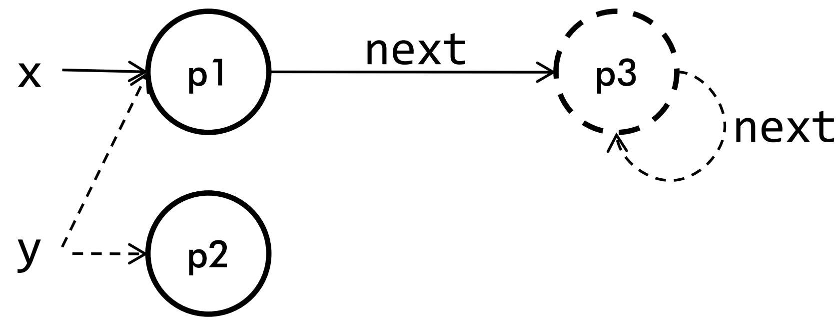
1. Introduction
2. Symbolic permissions
3. Annotation inference
4. Experimental results & Conclusion

Sample's Structure

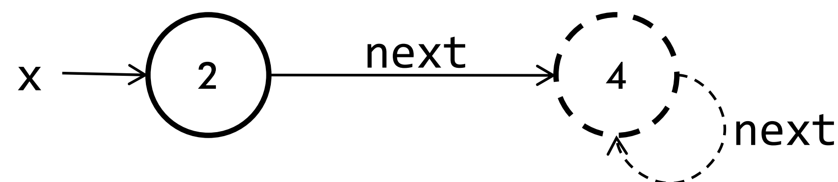


Heap Abstraction

- Generic approach
  - > Heap access
    - Heap identifier
  - > Assignments
- Standard analysis
  - > Allocation-site abstraction
  - > Quite rough!
    - Recursive data structures



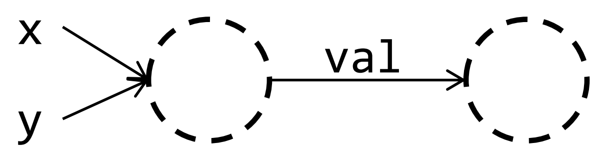

```
1: List l = new Node();
2: List it = l;
3: for(i <- 0 to 2) {
4:   it.next=new Node();
5:   it=it.next;
6:}
```



Unsoundness

- Method parameters
  > Not aliases
    - Unsound
  > Common approach
    - CodeContracts
    - Monoidics

```
void swap(Cell x, Cell y) {
  int temp = x.val;
  x.val=y.val;
  y.val=temp;
}
```

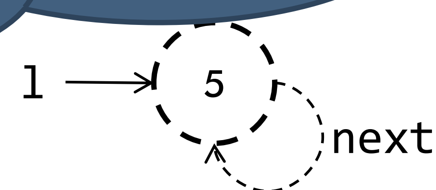


Less soundness ⇒ more contracts! ☺

More precise heap analysis ⇒ more contracts! ☺

More precise ⇒ slower! ☹

> … but (relatively) effective



Experimental Results

- 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