Automatic Inference of Fractional, Counting and Chalice Access Permissions

Pietro Ferrara and Peter Müller

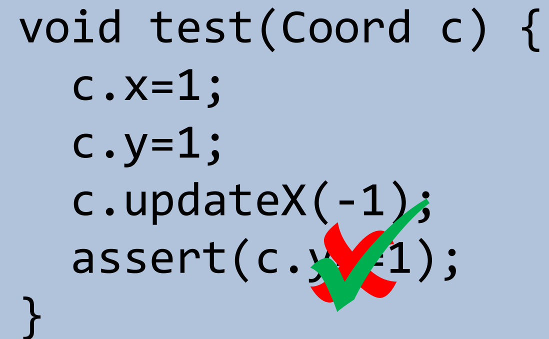
ETH Zurich
Switzerland

IBM Watson Research Center, Hawthorne, NY, U.S.A.

Access permissions

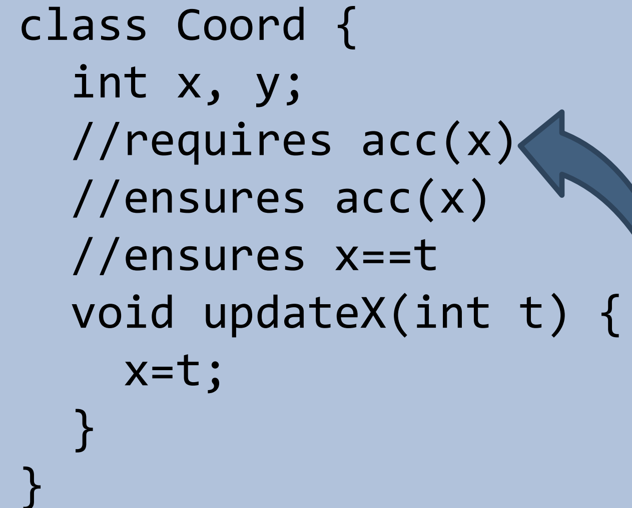
- OO programs
- Modular reasoning
  > Design by Contracts
- Side effects
  > Problem for modular reasoning!
- Restrict this scenario:
  > Access a location iff we have the permission

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



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 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

```
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

```
class Coord {
  //invariant acc(
  void updateX(int t) {
    acquire this;
    x=t;
    release this;
  }
}
```

acc(this.x)

acc(this.x)

x==t???
No permission on this.x!

```
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

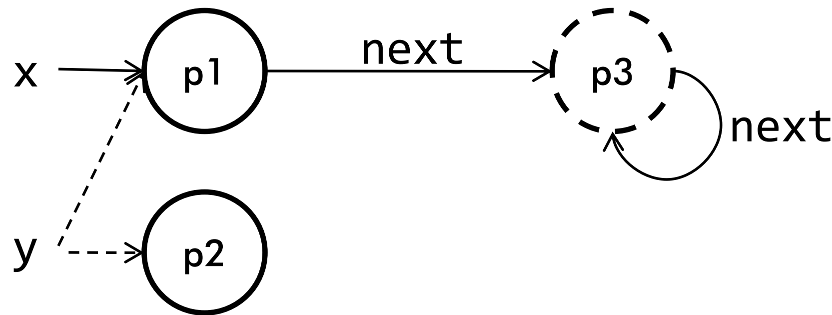
- Annotation overhead of permissions
  - > Quite verbose
  - > The code already contains all the accesses
- Start with a program without annotation
- Apply static analysis
  - > Based on abstract interpretation
  - > To infer the permissions that could be specified
    - Strong enough to perform the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

Outline

1. Introduction
2. Symbolic permissions
3. Annotation inference
4. A case study: the Owicki-Gries example
5. 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;
  }
}
```

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

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

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;
```

$\texttt{c.x} \mapsto \text{full}$
$\texttt{c.y} \mapsto \text{full}$ $\xrightarrow{\gamma}$ $\{\text{full}\}$

$\sqcup$

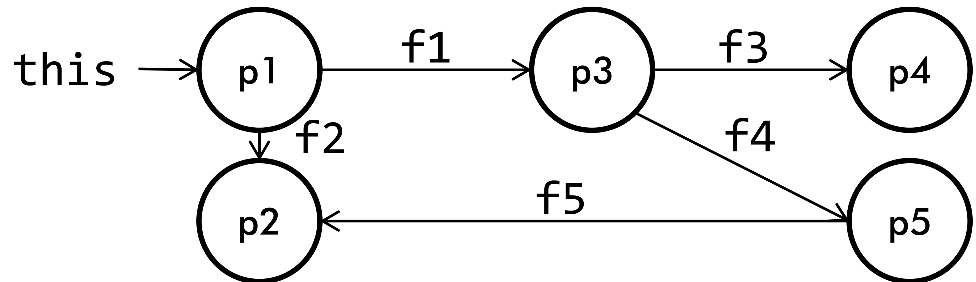
$\texttt{c.x} \mapsto 0$
$\texttt{c.y} \mapsto 0$ $\xrightarrow{\gamma}$ $\{0, \ldots, \text{full}\}$

$=$

$\texttt{c.x} \mapsto 0$
$\texttt{c.y} \mapsto 0$

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;
  }
}
```

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

Outline

1. Introduction
2. Symbolic permissions
3. Annotation inference
4. A case study: the Owicki-Gries example
5. 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 ≥ 1
Chalice	100	✓	✓	p $\geq \epsilon$

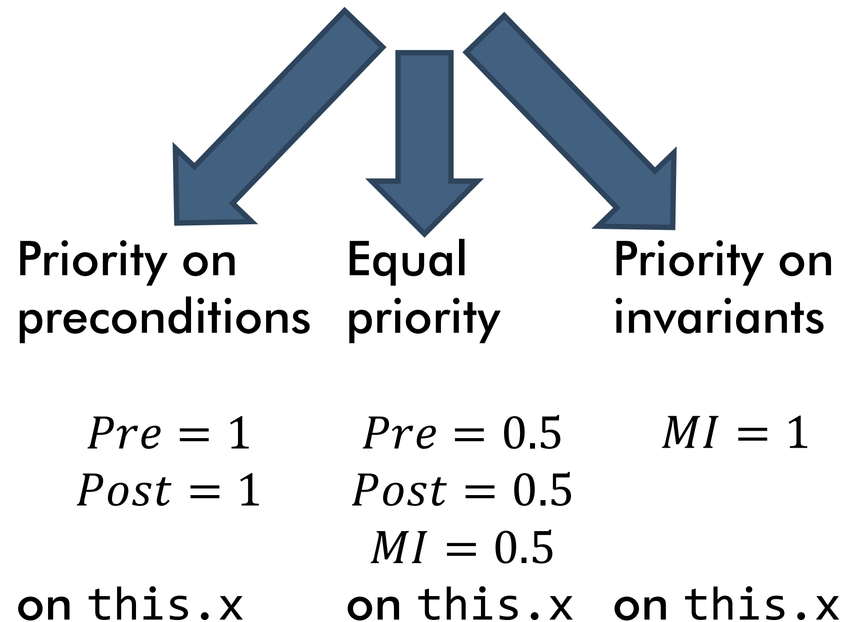
- Parameters
  - > Full perm.
  - > Fractional perm.
  - > Infinitesimal perm.
  - > Read perm.

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

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

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



Outline

1. Introduction
2. Symbolic permissions
3. Annotation inference
4. A case study: the Owicki-Gries example
5. Experimental results & Conclusion

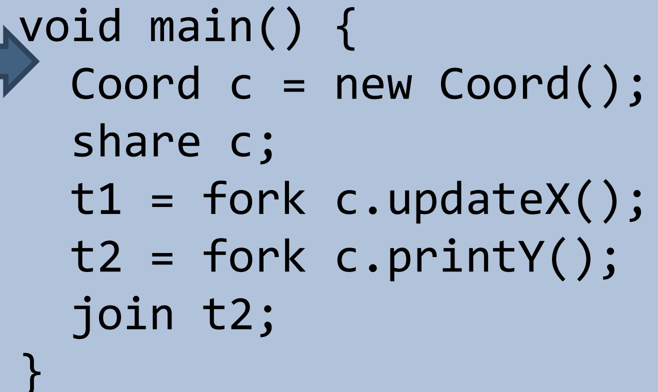
Chalice

- Experimental language
  - > Verification purposes
    - Pre- and post-conditions
    - Class (monitor) invariants
    - Loop invariants
    - Abstract predicates
- Focused on multithreading features:
  - > Join/fork of threads
  - > Sharing objects
  - > Acquire/release monitors

Chalice permissions

- `acc(x)`: 100% on x
- `acc(x, n)`: n% on x
  - > `acc(x)=acc(x,100)`
- `rd(x)`: ϵ on x
  - > ϵ infinitesimal (smallest) permission
- `share o`
  - > The monitor invariant of o is exhaled

```
class Coord {
  //invariant acc(y, 50)
  int x, y;
  //requires acc(x)
  //ensures acc(x)
  void updateX(int t) {…}
  //requires rd(y)
  //ensures rd(y)
  void printY() {…}
}
```



$\texttt{c.x} \mapsto 100\% \; -100\%$

$\texttt{c.y} \mapsto 100\% \; -50\% \; -\epsilon$

Discussion

```
class Cell {
  int x, c1, c2;
  invariant x==c1+c2;
}
```

rd(x) && rd(c1) && rd(c2)

- Can discharge acc(x) in the invariant of Cell
- Cannot discharge acc(c1) or acc(c2) in the invariant of Cell
- precondition of Inc() in W1 plus invariant in Cell == 100% for c1
- precondition of Inc() in W2 plus invariant in Cell == 100% for c2

```
  acquire c;
  assert c.x==2;
}
```

```
class W1 {
  Cell c;
  void Inc()
  ens c.c1==old(c.c1)+1
    acquire c;
    c.x=c.x+1;
    c.c1=c.c1+1;
    release c;
  }
}
```

rd(c.c1)

acc(c.c1) && acc(c.x)

```
class W2 {
  Cell c;
  void Inc()
  ens c.c2==old(c.c2)+1
    acquire c;
    c.x=c.x+1;
    c.c2=c.c2+1;
    release c;
  }
}
```

rd(c.c2)

acc(c.c2) && acc(c.x)

Abstract Semantics

```
class Cell {
  int x, c1, c2;
  invariant x==c1+c2
}

void main() {
  Cell c = new Cell();
  share c;
  W1 w1 = new W1();
  w1.c=c;
  W2 w2 = new W2();
  w2.c=c;
  t1 = fork w1.Inc();
  t2 = fork w2.I
  join t1;
  join t2;
  acquire c;
  assert c.x==2;
}
```

```
class W1 {
  Cell c;

  ens c.c1==old(c.c1)+1 {
    acquire c;
    c.x=c.x+1;
    c.c1=c.c1+1
    release c;
  }
}
```

```
class W2 {
  Cell c;

  ens c.c2==old(c.c2)+1 {
    acquire c;
    c.x=c.x+1;
    c.c2=c.c2+1
    release c;
  }
}
```

→MI(Cell,)

c.*→100

c.*→100-MI(Cell,*)-Pre(W1,Inc(),c.*)-Pre(W2,Inc(),c.*)

c.c1→Post(W1,Inc(),c.c1)

c.*→Pre(W1,Inc(),c.*)+MI(Cell,*)

c.c2→Post(W2,Inc(),c.c2)

c.*→Pre(W2,Inc(),c.*)+MI(Cell,*)

Constraint Inference

```
class Cell {
  int x, c1, c2;
  invariant x==c1+c2
}

void main() {
  Cell c = new Cell();
  share c;
  W1 w1 = new W1();
  w1.c=c;
  W2 w2 = new W2();
  w2.c=c;
  t1 = fork w1.Inc();
  t2 = fork w
  join t1;
  join t2;
  acquire c;
  assert c.x==2;
}
```

→MI(Cell,)

c.x→100-MI(Cell,*)-Pre(W1,Inc(),c.*)-Pre(W2,Inc(),c.*)

```
class W1 {
  Cell c;
    ens c.c1==old(c.c1)+1 {
      acquire c;
      c.x=c.x+1;
      c.c1=c.c1+1
      release c;
    }
}
```

c.c1→Post(W1,Inc(),c.c1)

c.*→Pre(W1,Inc(),c.*)+MI(Cell,*)

$$1 * Post(\text{W1}, \text{Inc}, \text{c.c1}) \geq \epsilon$$

$$1 * Pre(\text{W1}, \text{Inc}, \text{c.x}) + 1 * MI(\text{Cell}, \text{x}) = 100$$

$$1 * Pre(\text{W1}, \text{Inc}, \text{c.c1}) + 1 * MI(\text{Cell}, \text{c1}) = 100$$

$$1 * MI(\text{Cell}, *) \geq \epsilon$$

$$100 - 1 * MI(\text{Cell}, *) - 1 * Pre(\text{W1}, \text{Inc}, \text{c.}*) - 1 * Pre(\text{W2}, \text{Inc}, \text{c.}*) \geq 0$$

Solutions

$$1 * Post(\mathsf{W1}, \mathsf{Inc}, \mathsf{c.c1}) \geq \epsilon$$
$$1 * Pre(\mathsf{W1}, \mathsf{Inc}, \mathsf{c.x}) + 1 * MI(\mathsf{Cell}, \mathsf{x}) = 100$$
$$1 * Pre(\mathsf{W1}, \mathsf{Inc}, \mathsf{c.c1}) + 1 * MI(\mathsf{Cell}, \mathsf{c1}) = 100$$
$$1 * MI(\mathsf{Cell}, *) \geq \epsilon$$
$$100 - 1 * MI(\mathsf{Cell}, *) - 1 * Pre(\mathsf{W1}, \mathsf{Inc}, \mathsf{c.}*) - 1 * Pre(\mathsf{W2}, \mathsf{Inc}, \mathsf{c.}*) \geq 0$$

- Three possible scenarios:
  - > Maximize monitor invariants

Symbolic value	Numerical permission
Pre(W1, Inc, c.x)=Post(W1, Inc, c.x)	0
Pre(W1, Inc, c.c1)=Post(W1, Inc, c.c1)	ϵ
MI(Cell, c1)	100-ϵ
MI(Cell, x)	100

Solutions

$$1 * Post(\text{W1}, \text{Inc}, \text{c.c1}) \geq \epsilon$$

$$1 * Pre(\text{W1}, \text{Inc}, \text{c.x}) + 1 * MI(\text{Cell}, \text{x}) = 100$$

$$1 * Pre(\text{W1}, \text{Inc}, \text{c.c1}) + 1 * MI(\text{Cell}, \text{c1}) = 100$$

$$1 * MI(\text{Cell}, *) \geq \epsilon$$

$$100 - 1 * MI(\text{Cell}, *) - 1 * Pre(\text{W1}, \text{Inc}, \text{c.}*) - 1 * Pre(\text{W2}, \text{Inc}, \text{c.}*) \geq 0$$

- Three possible scenarios:
  - > Maximize pre- and post- conditions

Symbolic value	Numerical permission
Pre(W1, Inc, c.x)=Post(W1, Inc, c.x)	0
Pre(W1, Inc, c.c1)=Post(W1, Inc, c.c1)	100-ϵ
MI(Cell, c1)	ϵ
MI(Cell, x)	100

Solutions

$$1 * Post(\mathsf{W1}, \mathsf{Inc}, \mathsf{c.c1}) \geq \epsilon$$
$$1 * Pre(\mathsf{W1}, \mathsf{Inc}, \mathsf{c.x}) + 1 * MI(\mathsf{Cell}, \mathsf{x}) = 100$$
$$1 * Pre(\mathsf{W1}, \mathsf{Inc}, \mathsf{c.c1}) + 1 * MI(\mathsf{Cell}, \mathsf{c1}) = 100$$
$$1 * MI(\mathsf{Cell}, *) \geq \epsilon$$
$$100 - 1 * MI(\mathsf{Cell}, *) - 1 * Pre(\mathsf{W1}, \mathsf{Inc}, \mathsf{c.}*) - 1 * Pre(\mathsf{W2}, \mathsf{Inc}, \mathsf{c.}*) \geq 0$$

- Three possible scenarios:
  - > Distribute equally the permissions

Symbolic value	Numerical permission
Pre(W1, Inc, c.x)=Post(W1, Inc, c.x)	0
Pre(W1, Inc, c.c1)=Post(W1, Inc, c.c1)	50
MI(Cell, c1)	50
MI(Cell, x)	100

Outline

1. Introduction
2. Symbolic permissions
3. Annotation inference
4. A case study: the Owicki-Gries example
5. Experimental results & Conclusion

Sample's Structure

Source code

```
var x, y : Int = 0;
while(x<=i) {
  j=j*2;
  x++;
}
return j;
```

Compiler To Simple

Simple program

```
int x, j = 0;
x.>(i);
false
j=j.*(2);
x.++();
true
return j;
```

Heap analysis

head ① next ② next

getfieldId(1, “next”) → 2

createObject(Node) → 3

Semantic analysis

x → [0..+∞]

i → [-∞..+∞]

① → [1..3]

assign(y, 1) → {y→[1..1]}

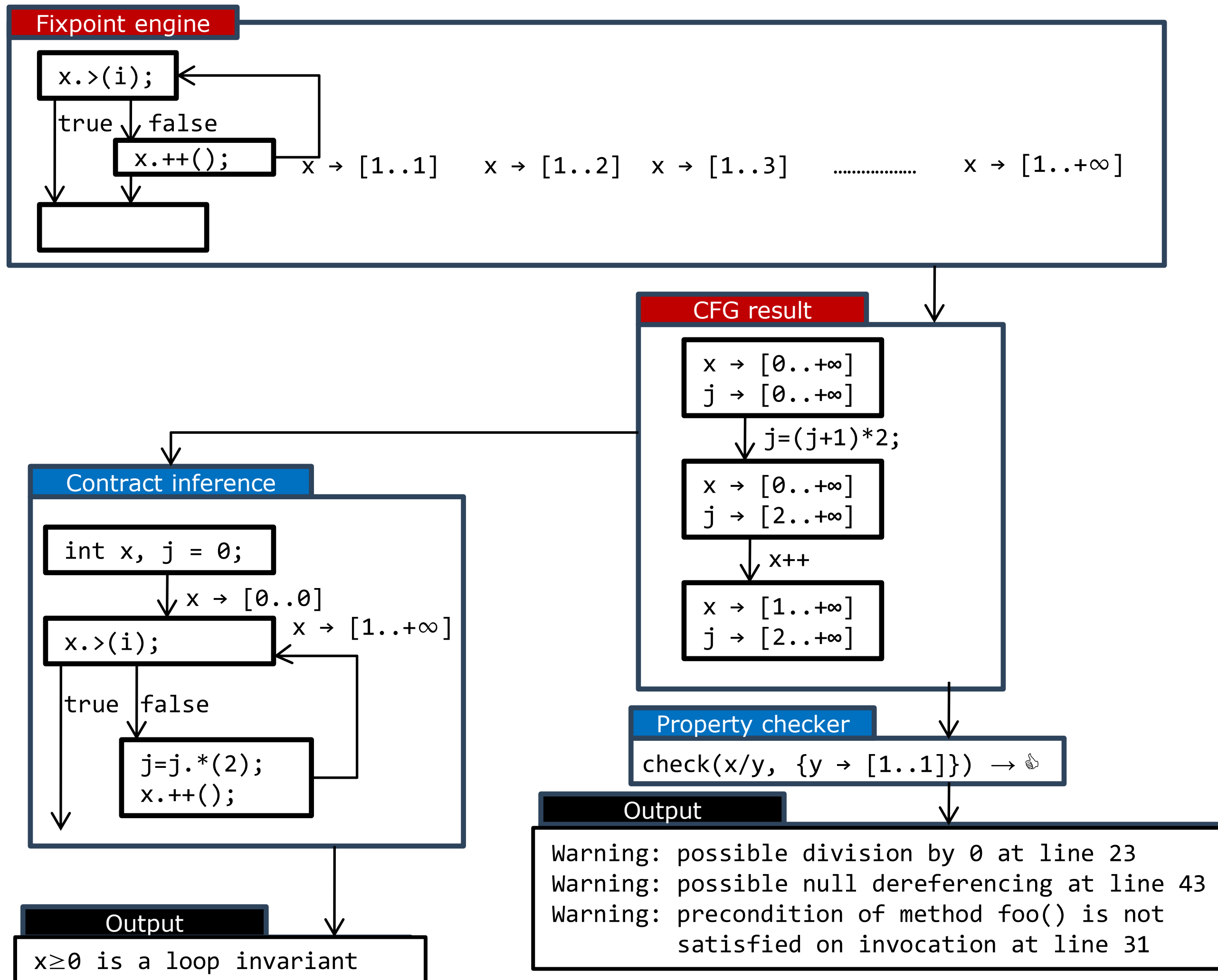
testTrue(① ==y+1) → {y → [1..1],① → [2..2]}

Fixpoint engine

```
x.>(i);
true  false
x.++();
```

x → [1..1] x → [1..2] x → [1..3] ……………… x → [1..+∞]

Sample's Structure



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