A few lessons that I learnt about interprocedural analyses

Hongseok Yang (Univ. of Oxford)

Mayur Naik, Ravi Mangal, Xin Zhang (Georgia Tech),
Wonchan Lee, Hakjoo Oh, Kwangkeun Yi (Seoul National Univ.)

Scalable static analyses for Java, which use cheap finite abstract domains.

A few lessons that I learnt about interprocedural analyses

Hongseok Yang (Univ. of Oxford)

Mayur Naik, Ravi Mangal, Xin Zhang (Georgia Tech),

Wonchan Lee, Hakjoo Oh, Kwangkeun Yi (Seoul National Univ.)

A few lessons that I learnt about interprocedural analyses

Hongseok Yang (Univ. of Oxford)

Mayur Naik, Ravi Mangal, Xin Zhang (Georgia Tech),
Wonchan Lee, Hakjoo Oh, Kwangkeun Yi (Seoul National Univ.)

Static analyses for C based on infinite numerical abstract domains

A few lessons that I learnt about interprocedural analyses

Hongseok Yang (Univ. of Oxford)

Mayur Naik, Ravi Mangal, Xin Zhang (Georgia Tech),
Wonchan Lee, Hakjoo Oh, Kwangkeun Yi (Seoul National Univ.)

Lesson 1

- Handling procedures well is really important in static analysis.
- Hajoo: About 50% false alarms of Sparse Sparrow on make and tar are due to procedures.

Lesson 2

- Two popular approaches for analysing procedures are equivalent, even when analyses are not distributive.

Lesson 2

- Two popular approaches for analysing procedures are equivalent, even when analyses are not distributive.

```
0:    int x;
1:    void main() {
2:      x = 1; inc();
3:      x = 1; foo();
4:      x = -x;
5:      inc();
6:  }
7:
8:  void foo() { inc(); }
9:  void inc() { x++; }
```

Callstring-based summary of inc

[2]: x>0 [3,8]: x>0 [5]: x<=0

Input-output summary of inc

x>0 ↦ x>0 x<0 ↦ x<=0

Lesson 3

- Often we can effectively guess program points where procedures should be analysed precisely.

Outline

- Review of callstring and functional approaches.
- Equivalence between these approaches (Lesson 2).
- Partial context-sensitivity (Lesson 3).

Outline

with Naik, Mangal and Zhang

- Review of callstring and functional approaches.
- Equivalence between these approaches (Lesson 2).
- Partial context-sensitivity (Lesson 3).

Outline

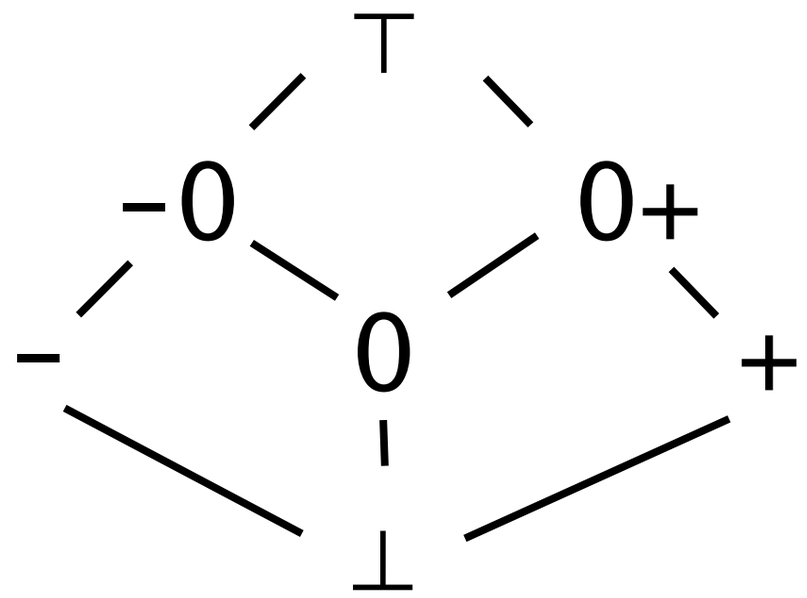
with Naik, Mangal and Zhang

- Review of callstring and functional approaches.
- Equivalence between these approaches (Lesson 2).
- Partial context-sensitivity (Lesson 3).

with Lee, Oh and Yi

Sign domain

- Abstract values:



- An abstract state is a map from variables to abstract values. E.g. `[x:0+, y:-]`.

Context-insensitive analysis

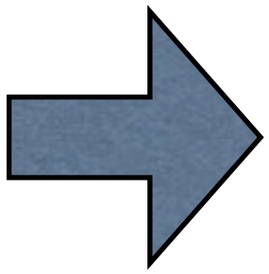
- It treats function calls and returns as gotos, and do not match call and return.
- Ignores calling contexts.
- Most imprecise but popular approach for handling procedures.

Context-insensitive analysis:
Treat call & return as gotos, and do not match them.

```
0:   int x;
1:   void main() {
2:     x = 0;
3:     while (*) {
4:       inc();
5:     }
6:     inc();
7:     assert(x > 0);
8:     x = -x;
9:     inc();
10:    assert(x >= 0);
11:  }
12:
13:  void inc() {
14:    x++;
15:  }
```

Context-insensitive analysis:
Treat call & return as gotos, and do not match them.

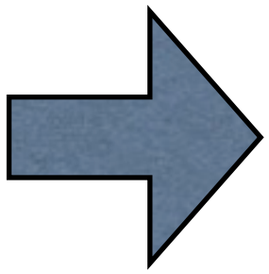
```
0:    int x;
1:    void main() {
2:      x = 0;
3:      while (*) {
4:        inc();
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {
14:     x++;
15:   }
```



```
0:    int x;
1:    void main() {
2:      x = 0;
3:      while (*) {
4:        goto 14;
5:      }
6:      goto 14;
7:      assert(x > 0);
8:      x = -x;
9:      goto 14;
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {
14:     x++; goto {5,7,10}
15:   }
```

Context-insensitive analysis:
Treat call & return as gotos, and do not match them.

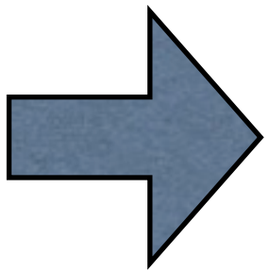
```
0:    int x;
1:    void main() {
2:      x = 0;
3:      while (*) {
4:        inc();
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {
14:     x++;
15:   }
```



```
0:    int x;
1:    void main() {
2:      x = 0;
3:      while (*) {
4:        goto 14;
5:      }
6:      goto 14;
7:      assert(x > 0);
8:      x = -x;
9:      goto 14;
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {
14:     x++; goto {5,7,10}
15:   }
```

Context-insensitive analysis:
Treat call & return as gotos, and do not match them.

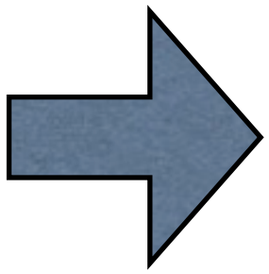
```
0:    int x;
1:    void main() {
2:      x = 0;
3:      while (*) {
4:        inc();
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {
14:     x++;
15:   }
```



```
0:    int x;
1:    void main() {
2:      x = 0;
3:      while (*) {
4:        goto 14;
5:      }
6:      goto 14;
7:      assert(x > 0);
8:      x = -x;
9:      goto 14;
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {
14:     x++; goto {5,7,10}
15:   }
```

Context-insensitive analysis:
Treat call & return as gotos, and do not match them.

```
0:    int x;
1:    void main() {
2:      x = 0;
3:      while (*) {
4:        inc();
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {
14:     x++;
15:   }
```



```
0:    int x;
1:    void main() {
2:      x = 0;
3:      while (*) {
4:        goto 14;
5:      }
6:      goto 14;
7:      assert(x > 0);
8:      x = -x;
9:      goto 14;
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {
14:     x++; goto {5,7,10}
15:   }
```

The analysis returns a false alarm at line 7.

Context sensitivity

- Distinguish calling contexts and analyse each procedure separately for different contexts.
- The functional approach does so based on input abstract states.
- The callstring approach uses call strings (i.e., sequences of call sites) instead.

Functional approach:
Separate analysis for each input abstract state.

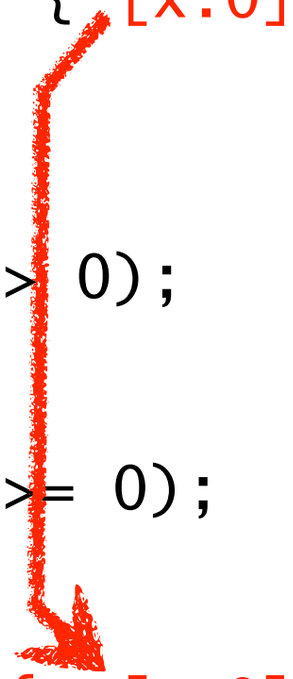
```
0:    int x;
1:    void main() {
2:      x = 0;
3:      while (*) {
4:        inc();
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {
14:     x++;
15:   }
```

Functional approach:
Separate analysis for each input abstract state.

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]
4:        inc();
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {
14:     x++;
15:   }
```

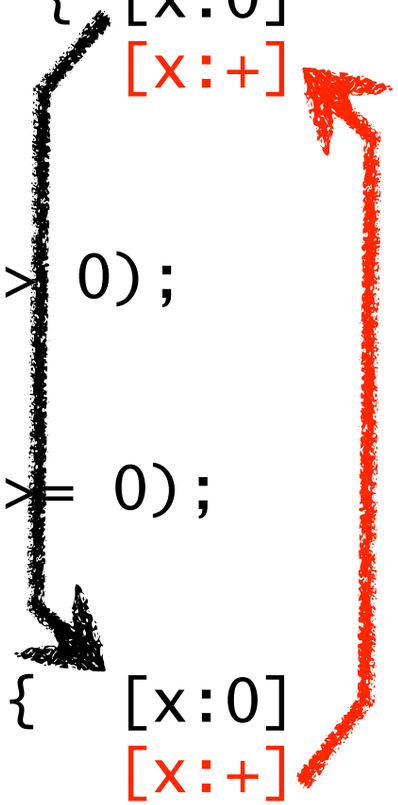
Functional approach:
Separate analysis for each input abstract state.

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]
4:        inc();
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  [x:0]
14:     x++;
15:   }
```



Functional approach:
Separate analysis for each input abstract state.

```
0:    int x;
1:    void main() {  [x:⊤]
2:      x = 0;       [x:0]
3:      while (*) {  [x:0]
4:        inc();     [x:+]
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {   [x:0]
14:     x++;         [x:+]
15:   }
```



Functional approach:
Separate analysis for each input abstract state.

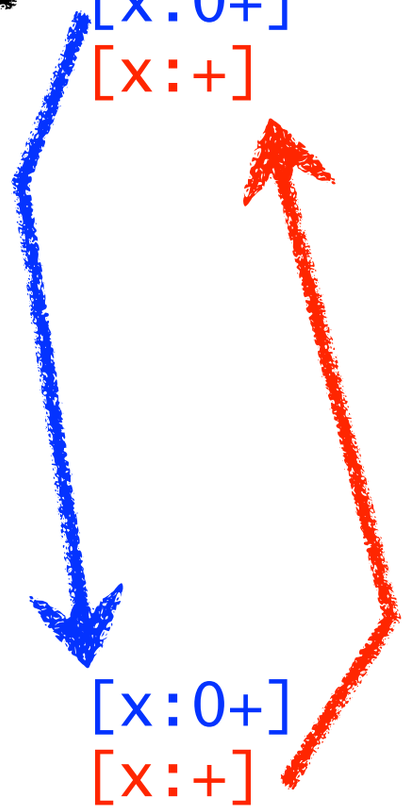
```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]
4:        inc();    [x:+]
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  [x:0]
14:     x++;        [x:+]
15:   }
```

Functional approach:
Separate analysis for each input abstract state.

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]  [x:0+]
4:        inc();    [x:+]
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  [x:0]
14:     x++;        [x:+]
15:   }
```

Functional approach:
Separate analysis for each input abstract state.

```
0:    int x;
1:    void main() {  [x:⊤]
2:      x = 0;       [x:0]
3:      while (*) {  [x:0]   [x:0+]
4:        inc();     [x:+]   [x:+]
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {   [x:0]   [x:0+]
14:     x++;         [x:+]   [x:+]
15:   }
```



Functional approach:
Separate analysis for each input abstract state.

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]   [x:0+]
4:        inc();    [x:+]   [x:+]
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  [x:0]   [x:0+]
14:     x++;        [x:+]   [x:+]
15:   }
```

Functional approach:
Separate analysis for each input abstract state.

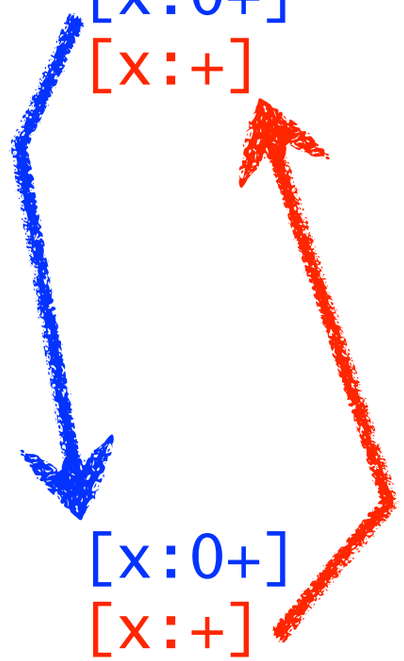
```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]   [x:0+]
4:        inc();    [x:+]   [x:+]
5:      }                   [x:0+]
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  [x:0]   [x:0+]
14:     x++;        [x:+]   [x:+]
15:   }
```

Functional approach:
Separate analysis for each input abstract state.

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]  [x:0+]
4:        inc();    [x:+]  [x:+]
5:      }                  [x:0+]
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  [x:0]  [x:0+]
14:     x++;        [x:+]  [x:+]
15:   }
```

Functional approach:
Separate analysis for each input abstract state.

```
0:    int x;
1:    void main() {  [x:⊤]
2:      x = 0;       [x:0]
3:      while (*) {  [x:0]   [x:0+]
4:        inc();     [x:+]   [x:+]
5:      }                    [x:0+]
6:      inc();               [x:+]
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {   [x:0]   [x:0+]
14:     x++;         [x:+]   [x:+]
15:   }
```



Functional approach:
Separate analysis for each input abstract state.

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]  [x:0+]
4:        inc();    [x:+]  [x:+]
5:      }                  [x:0+]
6:      inc();             [x:+]
7:      assert(x > 0);     [x:+]
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  [x:0]  [x:0+]
14:     x++;        [x:+]  [x:+]
15:   }
```

Functional approach:
Separate analysis for each input abstract state.

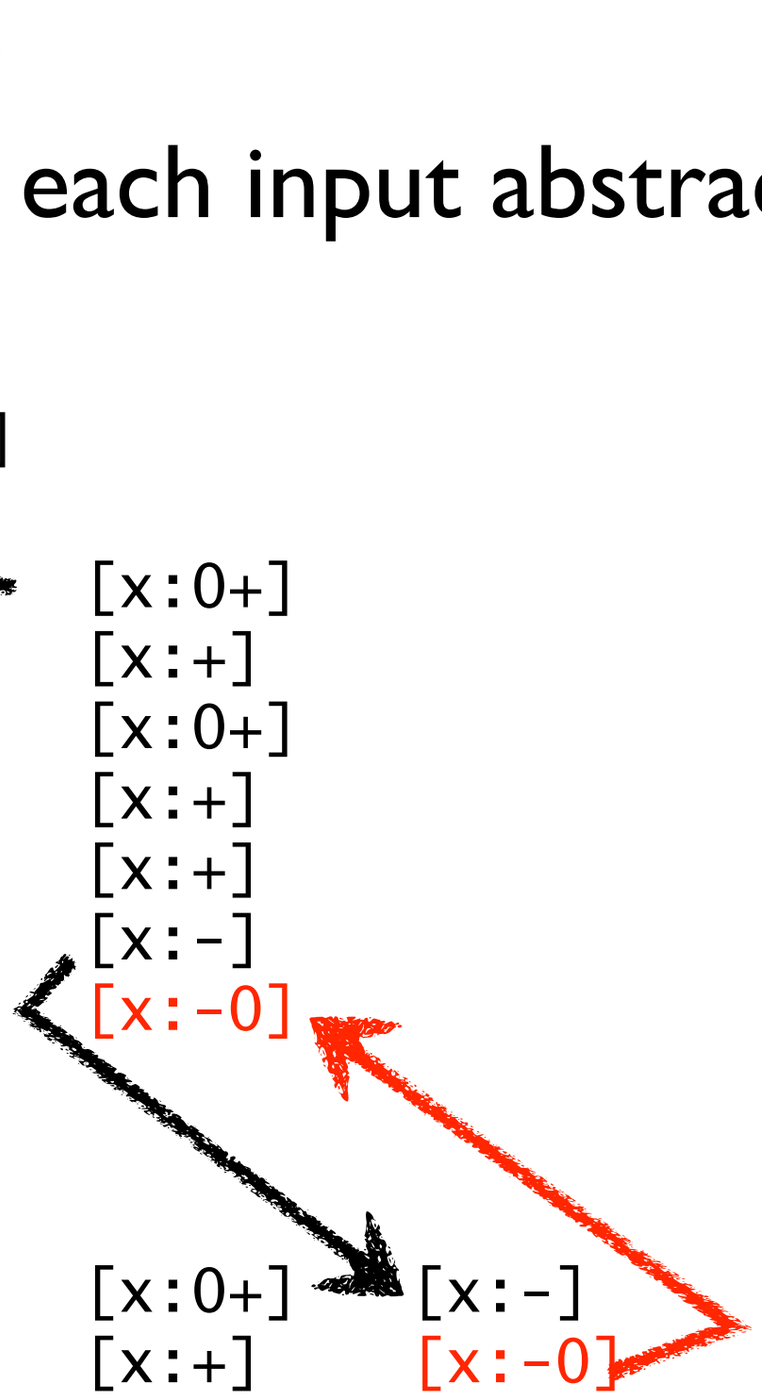
```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]  [x:0+]
4:        inc();    [x:+]  [x:+]
5:      }                  [x:0+]
6:      inc();             [x:+]
7:      assert(x > 0);     [x:+]
8:      x = -x;            [x:-]
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  [x:0]  [x:0+]
14:     x++;        [x:+]  [x:+]
15:   }
```

Functional approach:
Separate analysis for each input abstract state.

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;        [x:0]
3:      while (*) { [x:0]   [x:0+]
4:        inc();      [x:+]   [x:+]
5:      }                     [x:0+]
6:      inc();                [x:+]
7:      assert(x > 0);        [x:+]
8:      x = -x;               [x:-]
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {   [x:0]   [x:0+]   [x:-]
14:     x++;         [x:+]   [x:+]
15:   }
```

Functional approach:
Separate analysis for each input abstract state.

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;        [x:0]
3:      while (*) {   [x:0]   [x:0+]
4:        inc();      [x:+]   [x:+]
5:      }                     [x:0+]
6:      inc();                [x:+]
7:      assert(x > 0);        [x:+]
8:      x = -x;               [x:-]
9:      inc();                [x:-0]
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {    [x:0]   [x:0+]   [x:-]
14:     x++;          [x:+]   [x:+]    [x:-0]
15:   }
```



Functional approach:
Separate analysis for each input abstract state.

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;        [x:0]
3:      while (*) {   [x:0]   [x:0+]
4:        inc();      [x:+]   [x:+]
5:      }                     [x:0+]
6:      inc();                [x:+]
7:      assert(x > 0);        [x:+]
8:      x = -x;               [x:-]
9:      inc();                [x:-0]
10:     assert(x >= 0);       [x:-0]
11:   }
12:
13:   void inc() {   [x:0]   [x:0+]    [x:-]
14:     x++;         [x:+]   [x:+]     [x:-0]
15:   }
```

Functional approach:
Separate analysis for each input abstract state.

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]   [x:0+]
4:        inc();    [x:+]   [x:+]
5:      }                   [x:0+]
6:      inc();              [x:+]
7:      assert(x > 0);      [x:+]
8:      x = -x;             [x:-]
9:      inc();              [x:-0]
10:     assert(x >= 0);     [x:-0]
11:   }
12:
13:   void inc() {  [x:0]  [x:0+]   [x:-]
14:     x++;        [x:+]  [x:+]    [x:-0]
15:   }
```

Proved the first query.

Functional approach:
Separate analysis for each input abstract state.

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]   [x:0+]
4:        inc();    [x:+]   [x:+]
5:      }                   [x:0+]
6:      inc();              [x:+]
7:      assert(x > 0);      [x:+]
8:      x = -x;             [x:-]
9:      inc();              [x:-0]
10:     assert(x >= 0);     [x:-0]
11:   }
12:
13:   void inc() {  [x:0]  [x:0+]   [x:-]
14:     x++;        [x:+]  [x:+]    [x:-0]
15:   }
```

Proved the first query.

Computed a summary of inc().

Functional approach:
Separate analysis for each input abstract state.

```
0:    int x;
1:    void main() {  [x:⊤]
2:      x = 0;       [x:0]
3:      while (*) {  [x:0]   [x:0+]
4:        inc();     [x:+]   [x:+]
5:      }                    [x:0+]
6:      inc();               [x:+]
7:      assert(x > 0);       [x:+]
8:      x = -x;              [x:-]
9:      inc();               [x:-0]
10:     assert(x >= 0);      [x:-0]
11:   }
12:
13:   void inc() {   [x:0]   [x:0+]   [x:-]    [x:⊤]    [x:⊥]
14:     x++;         [x:+]   [x:+]    [x:-0]   ???      ???
15:   }
```

Lack of monotonicity.

Proved the first query.

Computed a summary of inc(). But something strange.

Functional approach:
Separate analysis for each input abstract state.

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]   [x:0+]
4:        inc();    [x:+]   [x:+]
5:      }                   [x:0+]
6:      inc();              [x:+]
7:      assert(x > 0);      [x:+]
8:      x = -x;             [x:-]
9:      inc();              [x:-0]
10:     assert(x >= 0);     [x:-0]
11:   }
12:
13:   void inc() {  [x:0]   [x:0+]   [x:-]
14:     x++;        [x:+]   [x:+]   [x:-0]
15:   }
```

Unused entry.
Difficult to see it as the least solution.

Proved the first query.

Computed a summary of inc(). But something strange.

Functional approach:
Separate analysis for each input abstract state.

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]   [x:0+]
4:        inc();    [x:+]   [x:+]
5:      }                   [x:0+]
6:      inc();              [x:+]
7:      assert(x > 0);      [x:+]
8:      x = -x;             [x:-]
9:      inc();              [x:-0]
10:     assert(x >= 0);     [x:-0]
11:   }
12:
13:   void inc() {  [x:0]   [x:0+]   [x:-]
14:     x++;        [x:+]   [x:+]   [x:-0]
15:   }
```

Unused entry.
Difficult to see it as the least solution.

Proved the first query.

Computed a summary of inc(). But something strange.

Callstring approach:
Separate analysis for each call string (call site here).

```
0:    int x;
1:    void main() {
2:      x = 0;
3:      while (*) {
4:        inc();
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {
14:     x++;
15:   }
```

Callstring approach:
Separate analysis for each call string (call site here).

```
0:   int x;
1:   void main() {
2:     x = 0;
3:     while (*) {
4:       inc();
5:     }
6:     inc();
7:     assert(x > 0);
8:     x = -x;
9:     inc();
10:    assert(x >= 0);
11:  }
12:
13:  void inc() {  3:          5:          8:
14:    x++;
15:  }
```

Callstring approach:
Separate analysis for each call string (call site here).

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]
4:        inc();
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  3:          5:          8:
14:     x++;
15:   }
```

Callstring approach:
Separate analysis for each call string (call site here).

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]
4:        inc();
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  3:[x:0]     5:          8:
14:     x++;
15:   }
```

Callstring approach:
Separate analysis for each call string (call site here).

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]
4:        inc();    [x:+]
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  3:[x:0]    5:          8:
14:     x++;          [x:+]
15:   }
```

Callstring approach:
Separate analysis for each call string (call site here).

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]
4:        inc();    [x:+]
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  3:[x:0]    5:         8:
14:     x++;          [x:+]
15:   }
```

Callstring approach:
Separate analysis for each call string (call site here).

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]  [x:0+]
4:        inc();    [x:+]
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  3:[x:0]    5:         8:
14:     x++;          [x:+]
15:   }
```

Callstring approach:
Separate analysis for each call string (call site here).

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]  [x:0+]
4:        inc();    [x:+]
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  3:[x:0]    5:         8:
14:     x++;          [x:+]
15:   }
```

Callstring approach:
Separate analysis for each call string (call site here).

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]  [x:0+]
4:        inc();    [x:+]
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  3:[x:0+]   5:          8:
14:     x++;          [x:+]
15:   }
```

Callstring approach:
Separate analysis for each call string (call site here).

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]   [x:0+]
4:        inc();    [x:+]   [x:+]
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  3:[x:0+]   5:          8:
14:     x++;          [x:+]
15:   }
```

Callstring approach:
Separate analysis for each call string (call site here).

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]   [x:0+]
4:        inc();    [x:+]   [x:+]
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  3:[x:0+]   5:          8:
14:     x++;          [x:+]
15:   }
```

Callstring approach:
Separate analysis for each call string (call site here).

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]   [x:0+]
4:        inc();    [x:+]   [x:+]
5:      }                   [x:0+]
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  3:[x:0+]   5:         8:
14:     x++;          [x:+]
15:   }
```

Callstring approach:
Separate analysis for each call string (call site here).

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]   [x:0+]
4:        inc();    [x:+]   [x:+]
5:      }                   [x:0+]
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  3:[x:0+]   5:[x:0+]   8:
14:     x++;          [x:+]
15:   }
```

Callstring approach:
Separate analysis for each call string (call site here).

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]   [x:0+]
4:        inc();    [x:+]   [x:+]
5:      }                   [x:0+]
6:      inc();              [x:+]
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  3:[x:0+]   5:[x:0+]  8:
14:     x++;          [x:+]      [x:+]
15:   }
```

Callstring approach:
Separate analysis for each call string (call site here).

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]   [x:0+]
4:        inc();    [x:+]   [x:+]
5:      }                   [x:0+]
6:      inc();              [x:+]
7:      assert(x > 0);      [x:+]
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  3:[x:0+]   5:[x:0+]  8:
14:     x++;          [x:+]      [x:+]
15:   }
```

Callstring approach:
Separate analysis for each call string (call site here).

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]   [x:0+]
4:        inc();    [x:+]   [x:+]
5:      }                   [x:0+]
6:      inc();              [x:+]
7:      assert(x > 0);      [x:+]
8:      x = -x;             [x:-]
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {  3:[x:0+]   5:[x:0+]  8:
14:     x++;          [x:+]      [x:+]
15:   }
```

Callstring approach:
Separate analysis for each call string (call site here).

```
0:    int x;
1:    void main() {  [x:⊤]
2:      x = 0;       [x:0]
3:      while (*) {  [x:0]   [x:0+]
4:        inc();     [x:+]   [x:+]
5:      }                    [x:0+]
6:      inc();               [x:+]
7:      assert(x > 0);       [x:+]
8:      x = -x;              [x:-]
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {   3:[x:0+]    5:[x:0+]   8:[x:-]
14:     x++;           [x:+]       [x:+]
15:   }
```

Callstring approach:
Separate analysis for each call string (call site here).

```
0:   int x;
1:   void main() { [x:⊤]
2:     x = 0;       [x:0]
3:     while (*) { [x:0]   [x:0+]
4:       inc();    [x:+]   [x:+]
5:     }                   [x:0+]
6:     inc();              [x:+]
7:     assert(x > 0);      [x:+]
8:     x = -x;             [x:-]
9:     inc();              [x:-0]
10:    assert(x >= 0);
11:  }
12:
13:  void inc() {  3:[x:0+]   5:[x:0+]   8:[x:-]
14:    x++;          [x:+]      [x:+]      [x:-0]
15:  }
```

Callstring approach:
Separate analysis for each call string (call site here).

```
0:    int x;
1:    void main() {  [x:⊤]
2:      x = 0;       [x:0]
3:      while (*) {  [x:0]   [x:0+]
4:        inc();     [x:+]   [x:+]
5:      }                    [x:0+]
6:      inc();               [x:+]
7:      assert(x > 0);       [x:+]
8:      x = -x;              [x:-]
9:      inc();               [x:-0]
10:     assert(x >= 0);      [x:-0]
11:   }
12:
13:   void inc() {   3:[x:0+]    5:[x:0+]   8:[x:-]
14:     x++;           [x:+]       [x:+]      [x:-0]
15:   }
```

Callstring approach:
Separate analysis for each call string (call site here).

```
0:   int x;
1:   void main() { [x:⊤]
2:     x = 0;       [x:0]
3:     while (*) { [x:0]   [x:0+]
4:       inc();    [x:+]   [x:+]
5:     }                   [x:0+]
6:     inc();              [x:+]
7:     assert(x > 0);      [x:+]
8:     x = -x;             [x:-]
9:     inc();              [x:-0]
10:    assert(x >= 0);     [x:-0]
11:  }
12:
13:  void inc() {  3:[x:0+]   5:[x:0+]  8:[x:-]
14:    x++;          [x:+]      [x:+]     [x:-0]
15:  }
```

Callstring approach:
Separate analysis for each call string (call site here).

```
0:    int x;
1:    void main() {  [x:⊤]
2:      x = 0;       [x:0]
3:      while (*) {  [x:0]   [x:0+]
4:        inc();     [x:+]   [x:+]
5:      }                    [x:0+]
6:      inc();               [x:+]
7:      assert(x > 0);       [x:+]
8:      x = -x;              [x:-]
9:      inc();               [x:-0]
10:     assert(x >= 0);      [x:-0]
11:   }
12:
13:   void inc() {   3:[x:0+]    5:[x:0+]   8:[x:-]
14:     x++;           [x:+]       [x:+]      [x:-0]
15:   }
```

Proved the first query. Same result for main as before.

Callstring approach:
Separate analysis for each call string (call site here).

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]   [x:0+]
4:        inc();    [x:+]   [x:+]
5:      }                   [x:0+]
6:      inc();              [x:+]
7:      assert(x > 0);      [x:+]
8:      x = -x;             [x:-]
9:      inc();              [x:-0]
10:     assert(x >= 0);     [x:-0]
11:   }
12:
13:   void inc() {   3:[x:0+]   5:[x:0+]   8:[x:-]
14:     x++;           [x:+]      [x:+]      [x:-0]
15:   }
```

Proved the first query. Same result for main as before.

A summary of inc() as a map from labels to results.

Callstring approach:
Separate analysis for each call string (call site here).

```
0:    int x;
1:    void main() { [x:⊤]
2:      x = 0;      [x:0]
3:      while (*) { [x:0]   [x:0+]
4:        inc();    [x:+]   [x:+]
5:      }                   [x:0+]
6:      inc();              [x:+]
7:      assert(x > 0);      [x:+]
8:      x = -x;             [x:-]
9:      inc();              [x:-0]
10:     assert(x >= 0);     [x:-0]
11:   }
12:
13:   void inc() {   3:[x:0+]   5:[x:0+]   8:[x:-]
14:     x++;           [x:+]      [x:+]      [x:-0]
15:   }
```

Nothing strange.
Least fixpoint.
But some redundancy.

Proved the first query. Same result for main as before.
A summary of inc() as a map from labels to results.

How are functional and callstring approaches related?

How are functional and callstring approaches related?

```
0:    int x;
1:    void main() {
2:      x = 0;
3:      while (*) {
4:        inc();
5:      }
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {
14:     x++;
15:   }
```

Analysis results for main by both approaches:

```
1:  [x:⊤]
2:  [x:0]
3:  [x:0+]
4:  [x:+]
5:  [x:0+]
6:  [x:+]
7:  [x:+]
8:  [x:-]
9:  [x:-0]
10: [x:-0]
```

Summary by fun. approach σ:

```
[x:0]    [x:0+]    [x:-]
[x:+]    [x:+]     [x:-0]
```

Summary by call. approach κ:

```
3:[x:+]    5:[x:+]    8:[x:-0]
```

Function η: 3:[x:0+] 5:[x:0+] 8:[x:-]

Relationship: κ = σ ∘ η

```
0:   int x;
1:   void main() {
2:     x = 0;
3:     while (*) {
4:       inc();
5:     }
6:     inc();
7:     assert(x > 0);
8:     x = -x;
9:     inc();
10:    assert(x >= 0);
11:  }
12:
13:  void inc() {
14:    x++;
15:  }
```

Analysis results for main by both approaches:

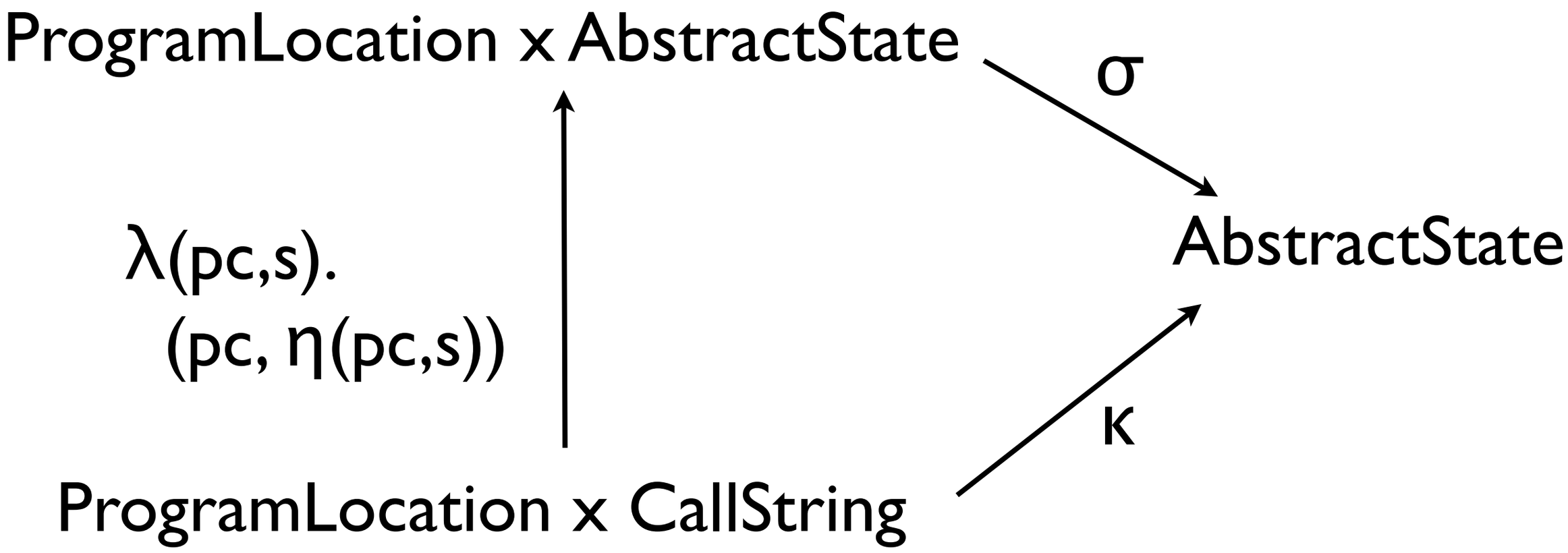
```
1:  [x:⊤]
2:  [x:0]
3:  [x:0+]
4:  [x:+]
5:  [x:0+]
6:  [x:+]
7:  [x:+]
8:  [x:-]
9:  [x:-0]
10: [x:-0]
```

Summary by fun. approach σ:

```
[x:0]    [x:0+]    [x:-]
[x:+]    [x:+]     [x:-0]
```

Summary by call. approach κ:

```
3:[x:+]    5:[x:+]    8:[x:-0]
```


ProgramLocation x AbstractState
σ
AbstractState
λ(pc,s).
(pc, η(pc,s))
κ
ProgramLocation x CallString

Functional approach

ProgramLocation x AbstractState

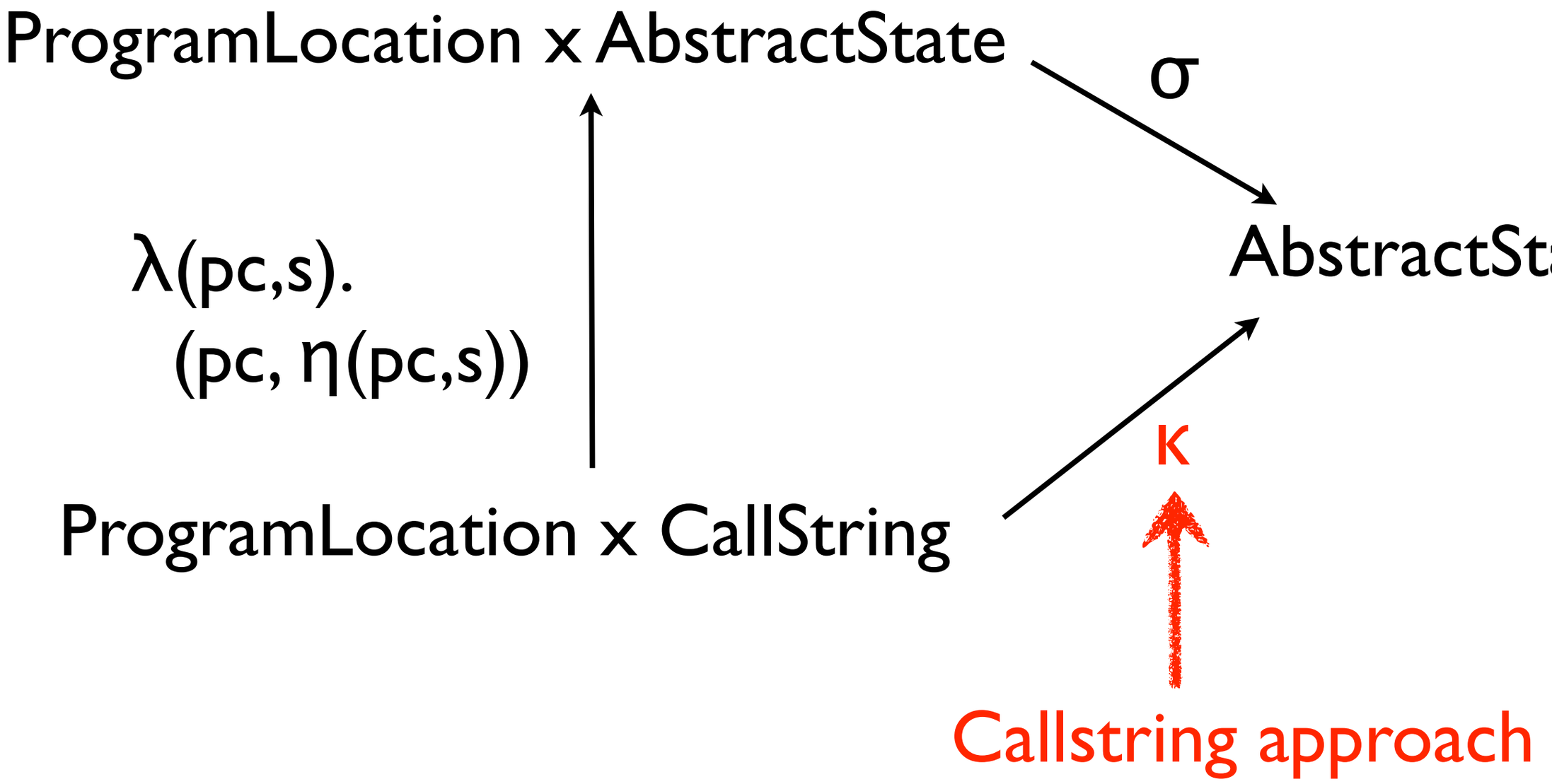
σ

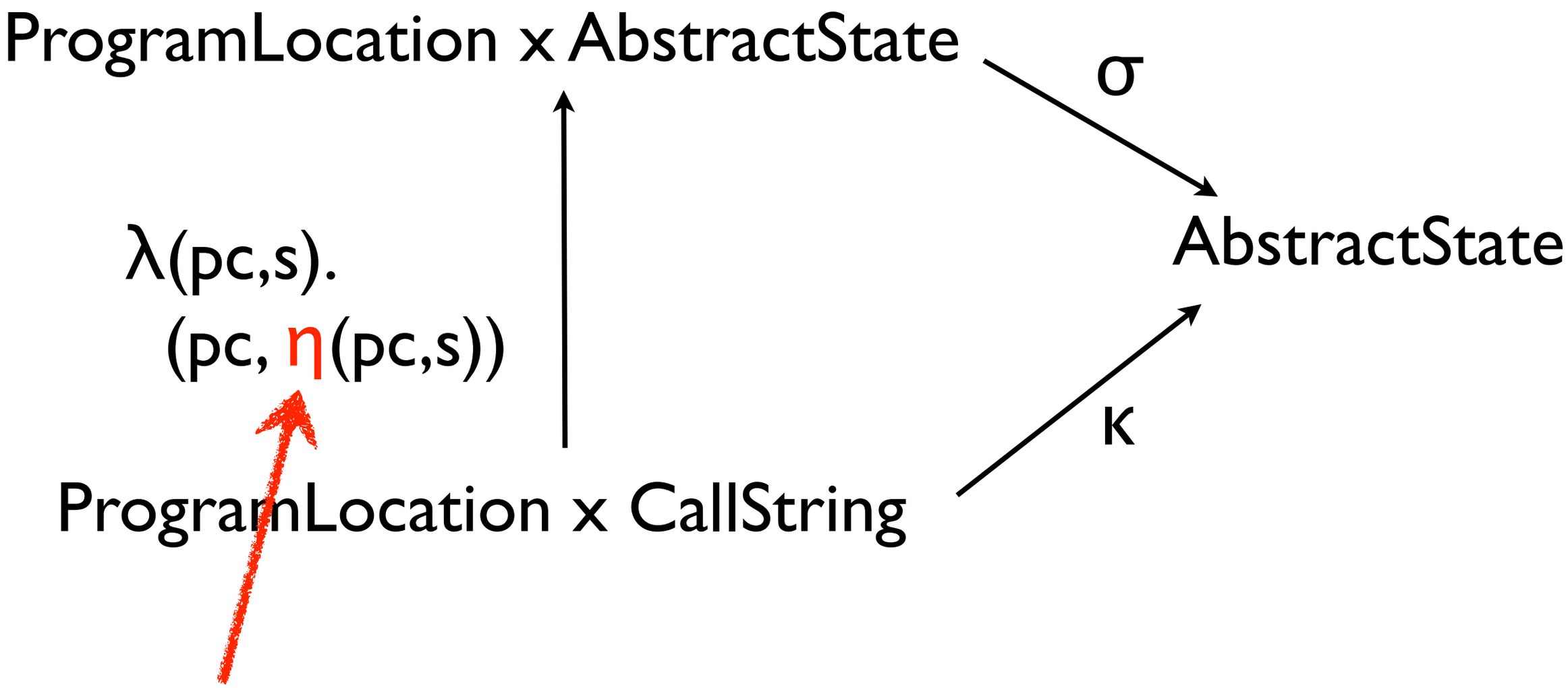
AbstractState

λ(pc,s).
(pc, η(pc,s))

κ

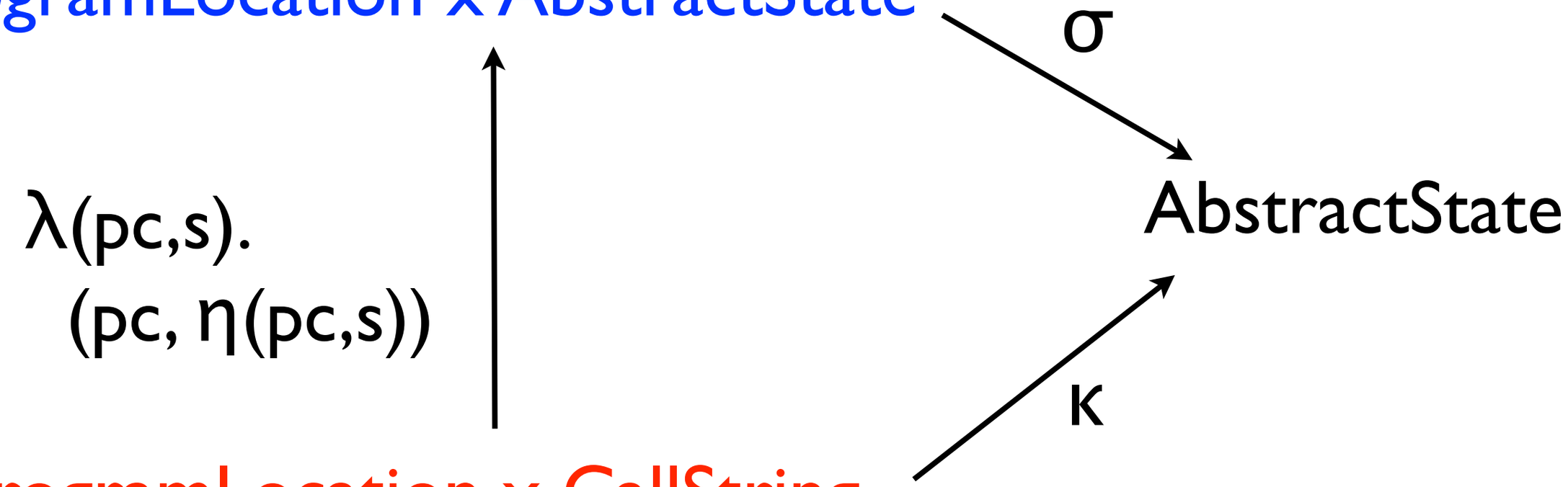
ProgramLocation x CallString


ProgramLocation x AbstractState
σ
λ(pc,s).
(pc, η(pc,s))
AbstractState
κ
ProgramLocation x CallString
Callstring approach


ProgramLocation x AbstractState
σ
AbstractState
λ(pc,s).
(pc, η(pc,s))
κ
ProgramLocation x CallString
Defined from σ by finding an
abstract state for each callstring.

Finite in many cases

ProgramLocation x AbstractState

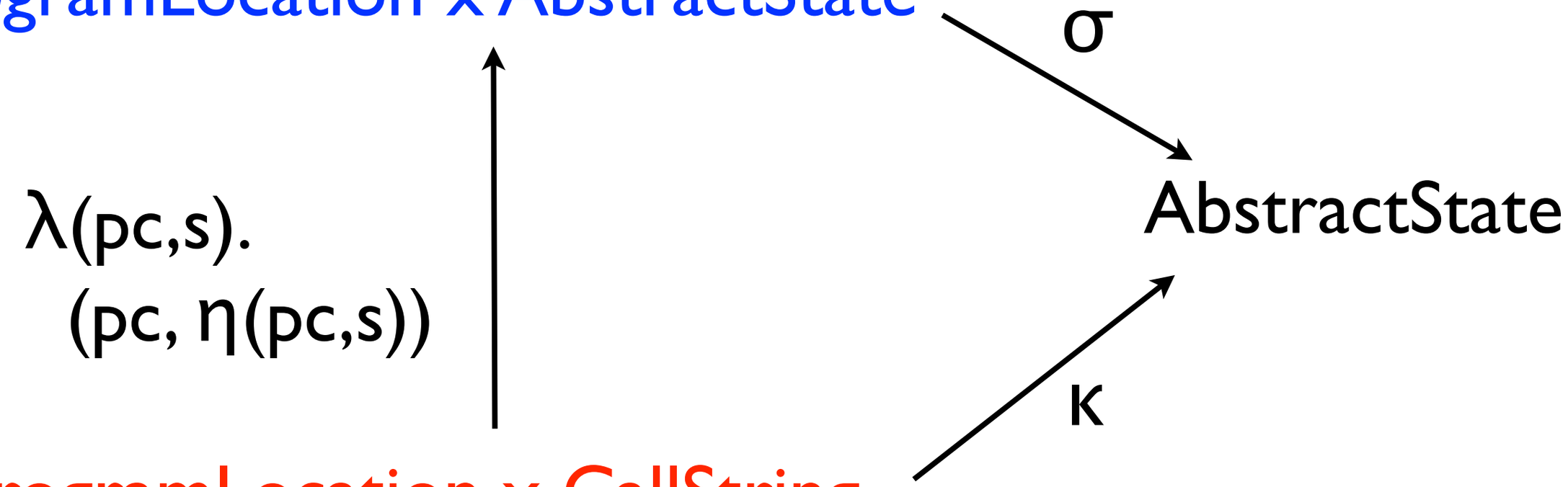


ProgramLocation x CallString

Always infinite

Finite in many cases

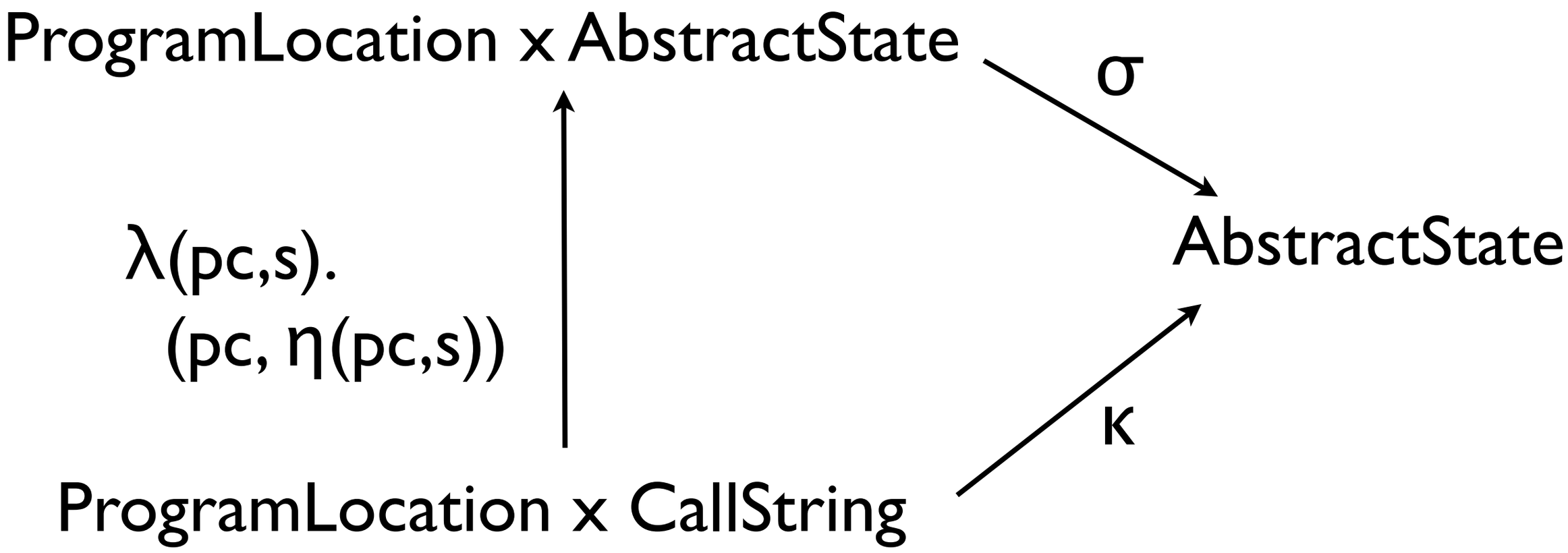
ProgramLocation x AbstractState



ProgramLocation x CallString

Always infinite

This gives an algorithm for doing callstring-based fully context-sensitive analysis.



Sharir & Pnueli proved for path-sensitive analyses.
We generalised it for non path-sensitive analyses.
Different proof techniques are needed.

Some further words

- Dominance relation in the result of function approach.
- Equivalence after garbage collection.

Experimental results

	0CFA$_I$	1CFA$_I$	2CFA$_I$	0CFA$_S$	SBA$_S$
antlr	1m45s	40m	72m	23m	21m
avrora	1m42s	38m	68m	26m	17m
bloat	3m10s	82m	239m	38m	60m
chart	4m40s	121m	256m	30m	51m
hsqldb	3m29s	74m	158m	34m	37m
luindex	2m34s	41m	83m	35m	27m
lusearch	2m22s	43m	80m	24m	16m
pmd	3m52s	61m	112m	34m	29m
sunflow	5m00s	148m	279m	58m	72m
xalan	2m32s	36m	82m	23m	16m

Table 3: Running times of the approaches from Table 1(b).

Experimental results

Functional approach

	0CFA$_I$	1CFA$_I$	2CFA$_I$	0CFA$_S$	SBA$_S$
antlr	1m45s	40m	72m	23m	21m
avrora	1m42s	38m	68m	26m	17m
bloat	3m10s	82m	239m	38m	60m
chart	4m40s	121m	256m	30m	51m
hsqldb	3m29s	74m	158m	34m	37m
luindex	2m34s	41m	83m	35m	27m
lusearch	2m22s	43m	80m	24m	16m
pmd	3m52s	61m	112m	34m	29m
sunflow	5m00s	148m	279m	58m	72m
xalan	2m32s	36m	82m	23m	16m

Table 3: Running times of the approaches from Table 1(b).

Experimental results

Functional approach

	0CFA$_I$	1CFA$_I$	2CFA$_I$	0CFA$_S$	SBA$_S$
antlr	1m45s	40m	72m	23m	21m
avrora	1m42s	38m	68m	26m	17m
bloat	3m10s	82m	239m	38m	60m
chart	4m40s	121m	256m	30m	51m
hsqldb	3m29s	74m	158m	34m	37m
luindex	2m34s	41m	83m	35m	27m
lusearch	2m22s	43m	80m	24m	16m
pmd	3m52s	61m	112m	34m	29m
sunflow	5m00s	148m	279m	58m	72m
xalan	2m32s	36m	82m	23m	16m

Approximate callstring approach

s of the approaches from Table 1(b).

```
0:   int x = -10;
1:   void main() {
2:     while (*) {
3:       inc();
4:     }
5:     x = 0;
6:     inc();
7:     assert(x > 0);
8:     x = -x;
9:     inc();
10:    assert(x >= 0);
11:  }
12:
13:  void inc() {
14:    x++;
15:  }
```

```
0:    int x = -10;
1:    void main() {
2:      while (*) {
3:        inc();
4:      }
5:      x = 0;
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {
14:     x++;
15:   }
```

[Q] Which one should we treat context-sensitively (i.e. inline)?

1. Line 6
2. Line 9
3. Lines 6,9
4. Lines 3,6,9

```
0:    int x = -10;
1:    void main() {
2:      while (*) {
3:        inc();
4:      }
5:      x = 0;
6:      inc();
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {
14:     x++;
15:   }
```

[Q] Which one should we treat context-sensitively (i.e. inline)?

1. Line 6
2. Line 9
3. Lines 6,9
4. Lines 3,6,9

```
0:    int x = -10;
1:    void main() {
2:      while (*) {
3:        inc();
4:      }
5:      x = 0;
6:      x++;
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {
14:     x++;
15:   }
```

[Q] Which one should we treat context-sensitively (i.e. inline)?

1. Line 6
2. Line 9
3. Lines 6,9
4. Lines 3,6,9

```
0:    int x = -10;
1:    void main() {
2:      while (*) {
3:        inc();
4:      }
5:      x = 0; // [x:0]
6:      x++;   // [x:+]
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {
14:     x++;
15:   }
```

[Q] Which one should we treat context-sensitively (i.e. inline)?

1. Line 6
2. Line 9
3. Lines 6,9
4. Lines 3,6,9

```
0:    int x = -10;
1:    void main() {
2:      while (*) {
3:        inc();
4:      }
5:      x = 0; // [x:0]
6:      x++;   // [x:+]
7:      assert(x > 0);
8:      x = -x;
9:      x++;
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {
14:     x++;
15:   }
```

[Q] Which one should we treat context-sensitively (i.e. inline)?

1. Line 6
2. Line 9
3. Lines 6,9
4. Lines 3,6,9

```
0:    int x = -10;
1:    void main() {
2:      while (*) {
3:        inc();
4:      }
5:      x = 0; // [x:0]
6:      x++;   // [x:+]
7:      assert(x > 0);
8:      x = -x;// [x:-]
9:      x++;   // [x:-0]
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {
14:     x++;
15:   }
```

[Q] Which one should we treat context-sensitively (i.e. inline)?

1. Line 6
2. Line 9
3. Lines 6,9
4. Lines 3,6,9

```
0:    int x = -10;
1:    void main() {
2:      while (*) {
3:        inc();
4:      }
5:      x = 0; // [x:0]
6:      x++;   // [x:+]
7:      assert(x > 0);
8:      x = -x;// [x:-]
9:      x++;   // [x:-0]
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {
14:     x++;
15:   }
```

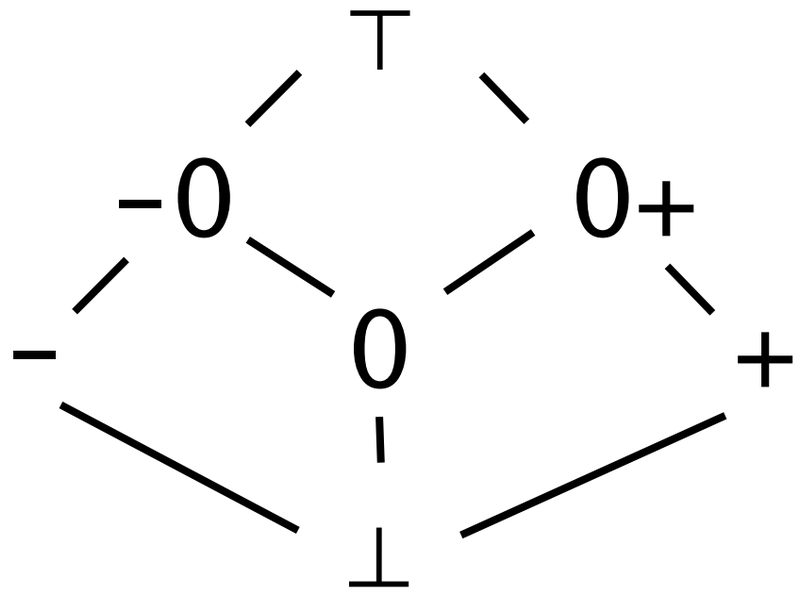
[Q] Which one should we treat context-sensitively (i.e. inline)?

1. Line 6
2. Line 9
3. Lines 6,9
4. Lines 3,6,9

We want to predict where context-sensitivity would help, without running the context-sensitive analysis.

Approximate a static analysis

- Further abstraction of abstract values.

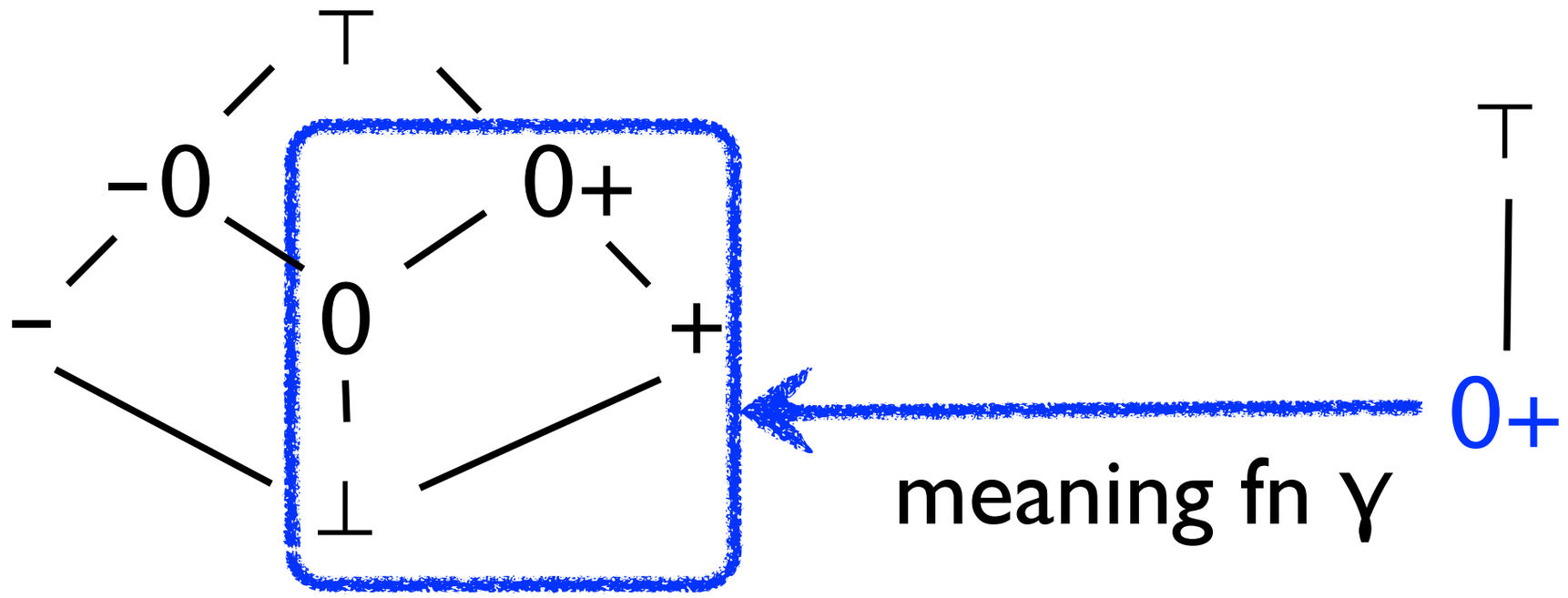


Old value domain

New value domain

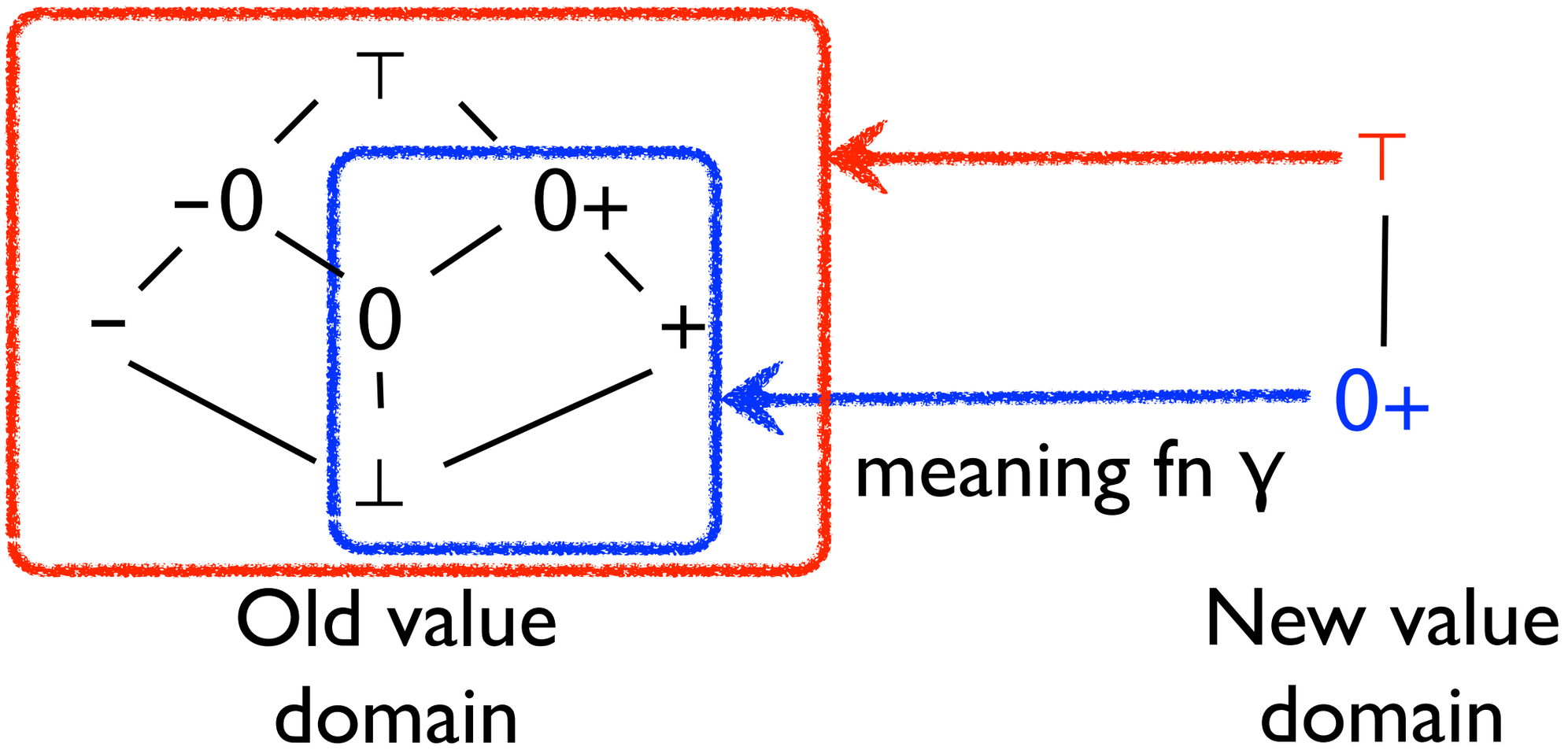
Approximate a static analysis

- Further abstraction of abstract values.



Approximate a static analysis

- Further abstraction of abstract values.



Approximate a static analysis

- Further abstraction of transfer functions:

$$(\gamma \circ [\![x{=}x{+}y]\!]_{new})a \;\sqsupseteq\; ([\![x{=}x{+}y]\!]_{old} \circ \gamma)a$$

- Use transfer functions of particular form only:

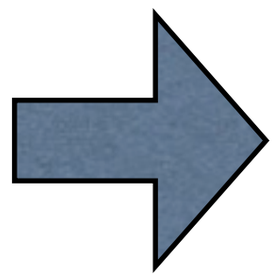
$$[\![x{=}x{+}y]\!]_{new}\, a = a[x{:}\, a(x) \sqcup a(y)]$$

- This restriction ensures an efficient algorithm for doing a fully context-sensitive analysis.

Approximate queries

- Express queries using new abstract states.
- Replace x>0 by x>=0 (i.e., a(x)=0+).

```
0:   int x = -10;
1:   void main() {
2:     while (*) {
3:       inc();
4:     }
5:     x = 0;
6:     inc();
7:     assert(x > 0);
8:     x = -x;
9:     inc();
10:    assert(x >= 0);
11:  }
```



```
0:   int x = -10;
1:   void main() {
2:     while (*) {
3:       inc();
4:     }
5:     x = 0;
6:     inc();
7:     assert(x >= 0);
8:     x = -x;
9:     inc();
10:    assert(x >= 0);
11:  }
```

```
0:   int x = -10;
1:   void main() {
2:     while (*) {      // [x:⊤]
3:       inc();         // [x:⊤]
4:     }                // [x:⊤]
5:     x = 0;           // [x:0+]
6:     inc();           // [x:0+]
7:     assert(x >= 0);
8:     x = -x;          // [x:⊤]
9:     inc();           // [x:⊤]
10:    assert(x >= 0);
11:  }
12:
13:  void inc() {       // [x:0+]      [x:⊤]
14:    x++;             // [x:0+]      [x:⊤]
15:  }
```

```
0:   int x = -10;
1:   void main() {
2:     while (*) {      // [x:⊤]
3:       inc();         // [x:⊤]
4:     }                // [x:⊤]
5:     x = 0;           // [x:0+]
6:     inc();           // [x:0+]
7:     assert(x >= 0);
8:     x = -x;          // [x:⊤]
9:     inc();           // [x:⊤]
10:    assert(x >= 0);
11:  }
12:
13:  void inc() {       // [x:0+]      [x:⊤]
14:    x++;             // [x:0+]      [x:⊤]
15:  }
```

```
0:   int x = -10;
1:   void main() {
2:     while (*) {      // [x:⊤]
3:       inc();         // [x:⊤]
4:     }                // [x:⊤]
5:     x = 0;           // [x:0+]
6:     inc();           // [x:0+]
7:     assert(x >= 0);
8:     x = -x;          // [x:⊤]
9:     inc();           // [x:⊤]
10:    assert(x >= 0);
11:  }
12:
13:  void inc() {       // [x:0+]      [x:⊤]
14:    x++;             // [x:0+]      [x:⊤]
15:  }
```

```
0:   int x = -10;
1:   void main() {
2:     while (*) {      // [x:⊤]
3:       inc();         // [x:⊤]
4:     }                // [x:⊤]
5:     x = 0;           // [x:0+]
6:     inc();           // [x:0+]
7:     assert(x >= 0);
8:     x = -x;          // [x:⊤]
9:     inc();           // [x:⊤]
10:    assert(x >= 0);
11:  }
12:
13:  void inc() {       // [x:0+]      [x:⊤]
14:    x++;             // [x:0+]      [x:⊤]
15:  }
```

```
0:    int x = -10;
1:    void main() {
2:      while (*) {       // [x:⊤]
3:        inc();          // [x:⊤]
4:      }                 // [x:⊤]
5:      x = 0;            // [x:0+]
6:      inc();            // [x:0+]
7:      assert(x >= 0);
8:      x = -x;           // [x:⊤]
9:      inc();            // [x:⊤]
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {        // [x:0+]      [x:⊤]
14:     x++;              // [x:0+]      [x:⊤]
15:   }
```

1) Collect all the proved approximate queries.

```
0:    int x = -10;
1:    void main() {
2:      while (*) {       // [x:⊤]
3:        inc();          // [x:⊤]
4:      }                 // [x:⊤]
5:      x = 0;            // [x:0+]
6:      inc();            // [x:0+]
7:      assert(x >= 0);
8:      x = -x;           // [x:⊤]
9:      inc();            // [x:⊤]
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {        // [x:0+]      [x:⊤]
14:     x++;              // [x:0+]      [x:⊤]
15:   }
```

1) Collect all the proved approximate queries.

```
0:    int x = -10;
1:    void main() {
2:      while (*) {       // [x:⊤]
3:        inc();          // [x:⊤]
4:      }                 // [x:⊤]
5:      x = 0;            // [x:0+]
6:      inc();            // [x:0+]
7:      assert(x >= 0);
8:      x = -x;           // [x:⊤]
9:      inc();            // [x:⊤]
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {        // [x:0+]      [x:⊤]
14:     x++;              // [x:0+]      [x:⊤]
15:   }
```

1) Collect all the proved approximate queries.

2) For such queries, find relevant procedure calls.

```
0:    int x = -10;
1:    void main() {
2:      while (*) {       // [x:⊤]
3:        inc();          // [x:⊤]
4:      }                 // [x:⊤]
5:      x = 0;            // [x:0+]
6:      x++;              // [x:0+]
7:      assert(x >= 0);
8:      x = -x;           // [x:⊤]
9:      inc();            // [x:⊤]
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {        // [x:0+]      [x:⊤]
14:     x++;              // [x:0+]      [x:⊤]
15:   }
```

1) Collect all the proved approximate queries.
2) For such queries, find relevant procedure calls.
3) All those calls should be treated context-sensitively.

```
0:    int x = -10;
1:    void main() {
2:      while (*) {
3:        inc();
4:      }
5:      x = 0;
6:      x++;
7:      assert(x > 0);
8:      x = -x;
9:      inc();
10:     assert(x >= 0);
11:   }
12:
13:   void inc() {
14:     x++;
15:   }
```

1) Collect all the proved approximate queries.
2) For such queries, find relevant procedure calls.
3) All those calls should be treated context-sensitively.
4) Run the original analysis.

Experimental results

Program	LOC	C.I.A.		Partially Context-Sensitive Analysis						Alarms↓	Time↑
		#alarm	time	#alarm	pre	main	total	#selected call-sites	depth		
spell-1.0	2K	58	0.6	30	0.3	0.9	1.2	25 / 124 (20.2%)	1.08 (3)	48.3%	2.0×
bc-1.06	13K	606	15.6	483	6.5	15.3	21.8	29 / 777 (3.7%)	1.16 (2)	20.3%	1.4×
tar-1.17	20K	940	43.8	799	11.8	43.5	55.3	56 / 1213 (4.6%)	1.02 (3)	15.0%	1.3×
less-382	24K	654	131.1	561	11.9	184.7	196.6	59 / 1522 (3.9%)	1.71 (4)	14.2%	1.5×
make-3.76.1	27K	1500	89.3	1002	20.3	124.2	144.5	87 / 1050 (8.3%)	1.20 (2)	33.2%	1.6×
grep-2.5	31K	1191	12.5	1182	5.8	15.4	21.2	37 / 530 (8.3%)	1.16 (3)	0.8%	1.7×
wget-1.9	35K	1307	72.0	905	29.9	126.1	156.0	111 / 1973 (5.6%)	1.39 (5)	30.8%	2.2×
bison-2.4	56K	2439	61.9	2249	52.6	37.5	69.93	165 / 1457 (11.3%)	1.53 (4)	7.8%	1.5×
a2ps-4.14	64K	3682	125.0	2004	205.3	343.6	548.9	263 / 2450 (10.7%)	2.20 (9)	45.6%	4.4×
lsh-2.0.4	111K	631	256.9	626	142.6	271.7	414.3	63 / 891 (7.7%)	2.96 (5)	0.8%	1.6×
Total	385K	13008	808.7	9841	486.9	1162.9	1629.7	764 / 13558 (5.6%)		**24.3%**	**2.0×**

We used an interval analysis for C programs.
Alarms are related to potential buffer overruns.