Abstract Semantic Differencing for Numerical Programs

Nimrod Partush
Eran Yahav

Technion, Israel

Semantic differencing

Characterize **semantic difference** between **similar programs**

Motivating example

```
1. if (input % 2 == 0) goto 2 else  goto 4
2. s := input+2
3. goto 5
4. s := input+3
5.
6. ptr := realloc(ptr,s)
7. // use ptr[0], ptr[1], … p
```

$2^{32} - 3 \leq \text{input} \leq 2^{32} - 1$

```
1. if (input % 2 == 0) goto 2
2. s := input+2
3. goto 5
4. s := input+3
5. + if (s>input) goto 6 else goto ERROR
6. ptr := realloc(ptr,s)
7. // use ptr[0], ptr[1], … ptr[input-1]
```

(Brumley et al. S&P 2008)

Abstract semantic differencing

- Use **abstract interpretation** to **prove equivalence** between two program versions
- Or **characterize their difference**
  - find (an abstraction of) **all inputs** that lead to **different output**
- **Sound**
  - never miss a difference
- **Precise**
  - report few false differences

Equivalence under abstraction

```
int sign(int x) {
  int sgn;
  if (x < 0)
    sgn = -1
  else
    sgn = 1


  return sgn
}
```

$$sign(x) = \begin{cases} -1 & , x<0 \\ 1 & , x\geq 0 \end{cases}$$

sgn ↦ [-1,-1]

sgn ↦ [1,1]

sgn ↦[-1,-1]⊔[1,1]

sgn ↦ [-1,1]

```
int sign(int x) {
  int sgn;
  if (x < 0)
    sgn = -1
  else
    sgn = 1
+ if (x == 0)
+    sgn = 0
  return sgn
}
```

$$sign(x) = \begin{cases} 1 & , x>0 \\ 0 & , x=0 \\ -1 & , x<0 \end{cases}$$

sgn ↦ [-1,-1]

sgn ↦ [1,1]

sgn ↦ [-1,-1]⊔[1,1]

sgn ↦ [0,0]

sgn ↦[0,0]⊔[-1,1]

sgn ↦ [-1,1]

Equivalence under abstraction does not entail equivalence between the concrete values it represents

Our approach

- Create a **correlating program** P ⋈ P’ which captures behaviors of P and P’
- Analyze P ⋈ P’ using a partially disjunctive **correlating abstract domain**
  - Track equivalence between variables of P and P’
  - Join states with the same equivalence relation (**partitioning**)

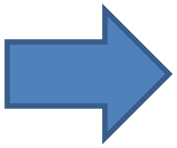
Correlating Program P ⋈ P’

- We create a new syntactic object that combines P and P’

```
int sign(int x) {
  int sgn;
  if (x < 0)
    sgn = -1
  else
    sgn = 1

  return sgn
}
```

```
int sign’(int x’) {
  int sgn’;
  if (x’ < 0)
    sgn’ = -1
  else
    sgn’ = 1
+ if (x’==0)
+  sgn’ = 0
  return sgn’
}
```



```
int sign⋈sign’(int x) {
  int x’ = x;
  int sgn, sgn’ = sgn;
  (x < 0) → sgn = -1
  (x' < 0) → sgn’ = -1
  (x ≥ 0) → sgn = 1
  (x’ ≥ 0) → sgn’ = 1
  (x'==0) → sgn’ = 0
}
```

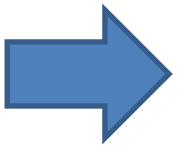
Correlating Program P ⋈ P’

- We create a new syntactic object that combines P and P’

```
int sign(int x) {
  int sgn;
  if (x < 0)
    sgn = -1
  else
    sgn = 1

  return sgn
}
```

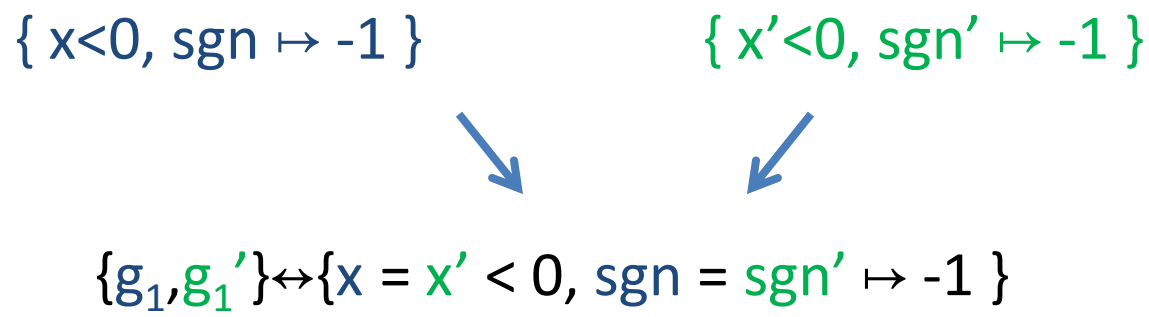
```
int sign’(int x’) {
  int sgn’;
  if (x’ < 0)
    sgn’ = -1
  else
    sgn’ = 1
+ if (x’==0)
+  sgn’ = 0
  return sgn’
}
```



```
int sign⋈sign’(int x) {
  int x’ = x;
  int sgn, sgn’ = sgn;
  guard g1 = (x < 0);
  guard g1’ = (x’ < 0);
  if (g1) sgn = -1;
  if (g1’) sgn’ = -1;
  if (!g1) sgn = 1;
  if (!g1’) sgn’ = 1;
  guard g2’ = (x’ == 0);
  if (g2’) sgn’ = 0;
  retval = sgn;
  retval’ = sgn’;
}
```

Correlating abstract domain

- Maintain **direct correlation** between values in the programs P and P'
  - Use an **relational** abstraction that **captures equivalences**



  - we use a **partially disjunctive domain** since we need to **delay joining**

Delay ⊔ to preserve equivalence

```
int sign⋈sign'(int x) {
  int x' = x;
  int sgn, sgn' = sgn;
  guard g1 = (x < 0);
  if (g1) sgn = -1;
  guard g1' = (x' < 0);
  if (g1') sgn' = -1;
   . . .
}
```

{ {}↔{x = x', sgn = sgn'} }

{ {g_1}↔{x = x'<0, sgn ↦ -1, sgn' ↦ ?},
 {¬g_1}↔{x = x'≥0, sgn = sgn'}

⊔ = { {}↔{x = x', sgn ≠ sgn'} }

{ {g_1,g_1'}↔{x = x'<0, sgn ↦ -1, sgn' ↦ -1},
 {¬g_1,¬g_1'}↔{x = x'≥0, sgn = sgn'} }

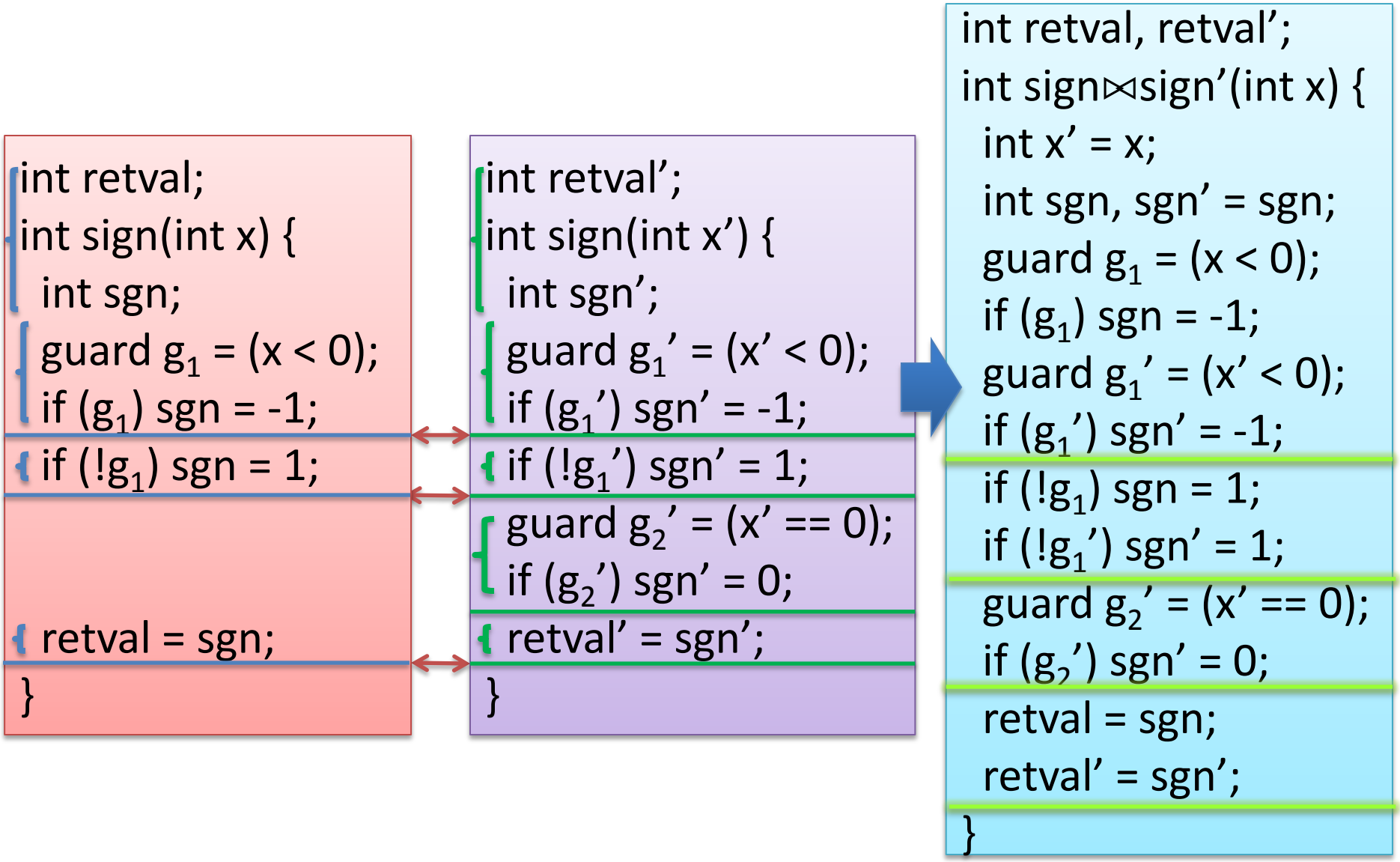
⊔ ={ {}↔{x = x', sgn = sgn'} }

Delay ⨅ to preserve equivalence

- We want to join at locations in P ⋈ P’ where equivalence is more likely to hold
  - After “matched” instructions both ran

- We call these **correlation points**
  - Part of P ⋈ P’ creation process

Picking correlation points in P ⋈ P’



Correlating Program P ⋈ P’

- P ⋈ P’ is a **reduction over P x P’** that’s better for tracking equivalences and finding differences
  - **Construct the program in a way that matches the abstraction**
  - **brings matched instructions closer together**

- In general, can search the space of potential correlating programs
  - As well as correlation points

Delay ⊔ to preserve equivalence

```
int sign⋈sign'(int x) {
  int x' = x;
  int sgn, sgn' = sgn;
  guard g1 = (x < 0);
  if (g1) sgn = -1;
  guard g1' = (x' < 0);
  if (g1') sgn' = -1;
  ⊔ =
   if (!g1) sgn = 1;
  if (!g1') sgn' = 1;
  guard g2' = (x' == 0);
  if (g2') sgn' = 0;
  retval = sgn;
  retval' = sgn';
}
```

{ {g_1}↔{x = x'<0, sgn ↦ -1, sgn' ↦ ?} ,
{¬g_1}↔{x = x'≥0, sgn = sgn' ↦ ?} }

{ {g_1,g_1'}↔{x = x'<0, sgn ↦ -1, sgn' ↦ -1} ,
{¬g_1,¬g_1'}↔{x = x'≥0, sgn = sgn' ↦ ?} }

⊔ ={ {}↔{x = x', sgn = sgn' ↦ ?} }

{ {g_1,g_1'}↔{x = x'<0, sgn ↦ 1, sgn' ↦ 1} ,
{¬g_1,¬g_1'}↔{x = x'≥0, sgn = sgn' ↦ ?} }

⊔ ={ {}↔{x = x', sgn = sgn' ↦ ?} }

{ {¬g_2'}↔{x = x', sgn = sgn' ↦ ?} ,
{g_2'}↔{x = x' = 0, sgn ↦ 1, sgn' ↦ 0} }

⊔ ={ {}↔{x = x', sgn ≠ sgn'} }

Partitioning based on equivalence

- Join abstract states based on the **equivalences they preserve**
  - the set of variables that hold equivalence
  - disjunction size bound at $2^{|VAR|}$
  - lose some information, but maintain what's important (equivalence)

Correlating Analysis for P⋈P’

```
int sign⋈sign’(int x) {
  int x’ = x;
  int sgn, sgn’ = sgn;
  guard g1 = (x < 0);
  if (g1) sgn = -1;
  guard g1’ = (x’ < 0);
  if (g1’) sgn’ = -1;
   if (!g1) sgn = 1;
  if (!g1’) sgn’ = 1;
  guard g2’ = (x’ == 0);
  if (g2’) sgn’ = 0;
  retval = sgn;
  retval’ = sgn’;
}
```

$\{ \{\neg g_2'\} \leftrightarrow \{x = x', sgn = sgn' \mapsto ?\}$,
$\{g_2'\} \leftrightarrow \{x = x' = 0, sgn \mapsto 1, sgn' \mapsto 0\}$ } eqv

$\bigsqcup_{\mathbf{eqv}} = \{ \{\neg g_2'\} \leftrightarrow \{x = x', sgn = sgn' \mapsto ?\}$,
$\{g_2'\} \leftrightarrow \{x = x' = 0, sgn \mapsto 1, sgn' \mapsto 0\} \}$ diff

Semantic differencing for **loops**

P

```
int foo(int x, int y, int z) {
  while (x > 0) {
    if (z > 0)
-     y++;
    x--;
  }
  return y;
}
```

P'

```
int foo'(int x', int y', int z') {
  while (x' > 0) {
    if (z' > 0)
+     y'--;
    x'--;
  }
  return y';
}
```

- These programs differ for cases where **z>0** and are otherwise equivalent

Semantic differencing for **loops**

```
int foo⋈foo'(int x, int y, int z) {
  int x' = x, y' = y , z' = z;
loop:
  guard g1 = (x > 0);
loop':
  guard g1' = (x' > 0);
  guard g2 = (z > 0);
  guard g2' = (z' > 0);
  if (g1 && g2) y++;
  if (g1' && g2') y'--;
  if (g1) x--;
  if (g1') x'--;
  if (g1) goto loop;
  if (g1') goto loop';
}
```

$$\{\ \{g_1,g_2\}\leftrightarrow\{z = z' > 0,\ x = x' > 0,\ y = y' + 1\},\ \{g_1,\neg g_2\}\leftrightarrow\{z = z' \le 0,\ x = x' > 0,\ y = y'\},\ \{\neg g_1\}\leftrightarrow\{z = z',\ x = x' \le 0,\ y = y'\}\ \}$$

$$\{\ \{g_1,g_1,g_2,g_2\}\leftrightarrow\{z = z' > 0,\ x = x' > 0,\ y = y' + 2\}\ \{g_1,g_1,\neg g_2,\neg g_2\}\leftrightarrow\{z = z' \le 0,\ x = x' > 0,\ y = y'\}\ \{\neg g_1,\neg g_1\}\leftrightarrow\{z = z',\ x = x' \le 0,\ y = y'\}\ \}$$

$$\sqcup_{\mathbf{eqv}} = \{\ \{\}\leftrightarrow\{z = z',\ x = x',\ y = y'\},\ \{g_1,g_1,g_2,g_2\}\leftrightarrow\{z = z' > 0,\ x = x' > 0,\ y = y' + 2\}\ \}$$

Semantic differencing for **loops**

```
int foo⋈foo'(int x, int y, int z) {
  int x' = x, y' = y , z' = z;
loop:
  guard g1 = (x > 0);
loop':
  guard g1' = (x' > 0);
  guard g2 = (z > 0);
  guard g2' = (z' > 0);
  if (g1 && g2) y++;
  if (g1' && g2') y'--;
  if (g1) x--;
  if (g1') x'--;
  if (g1) goto loop;
  if (g1') goto loop';
}
```

{ {}↔{z = z', x = x', y = y'} ,
{g1,g1,g2,g2}↔{z = z' > 0, x = x' > -1, y = y' + 2} }

{ {}↔{z = z', x = x', y = y'} ,
{g1,g1,g2,g2}↔{z = z' > 0, x = x' > -1, y = y' + 4}

• • •

?

Semantic differencing for **loops**

- We need to **widen in the powerset domain**, but **which sub-states** should be matched?
  - Strategy 1: Widen-by-equivalence
  - Strategy 2: Widen-by-guards

Widen-by-equivalence

```
int foo⋈foo'(int x, int y, int z) {
  int x' = x, y' = y , z' = z;
loop:
  guard g1 = (x > 0);
loop':
  guard g1' = (x' > 0);
  guard g2 = (z > 0);
  guard g2' = (z' > 0);
  if (g1 && g2) y++;
  if (g1' && g2') y'--;
  if (g1) x--;
  if (g1') x'--;
  if (g1) goto loop;
  if (g1') goto loop';
}
```

{ {}↔{z = z', x = x', y = y'} ,
{g1,g1,g2,g2}↔{z = z' > 0, x = x' > -1, y = y' + 2} }

{ {}↔{z = z', x = x', y = y'} ,
{g1,g1,g2,g2}↔{z = z' > 0, x = x' > -1, y = y' + 4}

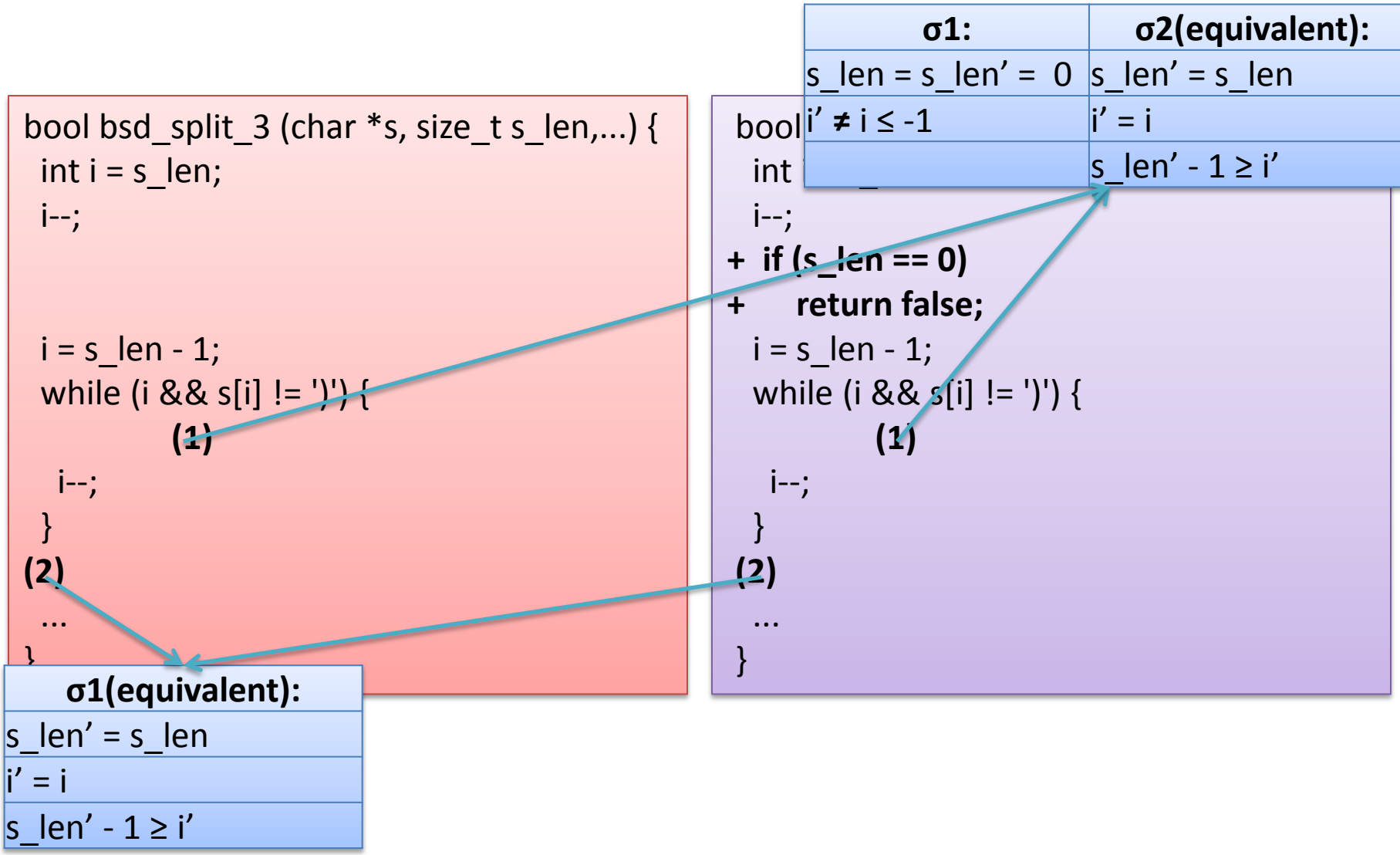
• • •

{ {}↔{z = z', x = x', y = y'} ,
{g1,g1,g2,g2}↔{z = z' > 0, x = x' > -1, y < y'} }

Results summary

Name	#LOC	#P	Widen	Octagon (Part-by-eqv)	Octagon (Part-by-guard)	Polyhedra (Part-by-eqv)	Polyhedra (Part-by-guard)
remove	16	4	No	✔	✔	✔	✔
copy	44	2	No	✔	✔	✔	✔
fmt	42	5	Yes	✘	*TO*	✔	✔
md5sum	40	3	Yes	✔	*TO*	✔	✔
pr	100	10	Yes	*TO*	*TO*	✔	*TO*
savewd	86	1	No	✔	✔	✔	✔
seq	23	15	Yes	✘	*TO*	✘	✘
addr	77	1	No	✔	*TO*	✔	*TO*
nsGDDN	47	11	No	✘	✘	✔	✔
sign	8	2	No	✔	✔	✔	✔
sum	7	5	Yes	✘	✘	✔	✔

Results



coreutils md5sum 6.10 vs. 6.11

Results

σ1	σ2	σ3
input_position0 = 0	input_position0 < -width	input_position0 < -width
chars' = 0	input_position0 < 0	input_position0 > 0
input_position = width	input_position' = 0	input_position' = 0
input_position < 0	input_position < width	input_position > width
input_position' = 0	width < 0	input_position <= 0

```
int i

boo
  int
    ...

  input_position += width;

    ...
  return chars;
}
```

```
    ...
+  if (width < 0 && input_position == 0) {
+      chars = 0;
+      input_position = 0;
+  } else if (width < 0 && input_position <= -width) {
+    input_position = 0;
+  } else {
       input_position += width;
+ }
    ...
  return chars;
}
```

coreutils pr 6.10 vs. 6.11

Summary

- Abstract semantic differencing
  - Characterize difference between similar programs
  - Many applications for computed difference
- Key ideas that make this work
  - Correlating program
  - Correlating abstract domain
    - Partially disjunctive based on equivalences
- Results over real-world patches

Proposed Questions

1. Why not use a correlating semantics instead of P⋈P'?
2. Partitioning
   1. are guard values considered in partitioning?
   2. isn't $2^{|VAR|}$ too big?
3. How do you handle other data types?
4. How did you handle procedures?
5. How are the variables matched?
6. Future work?

Backup slides

Procedures

- Establish equivalence of procedures bottom-up
- Use the equivalence of callees to establish equivalence of callers
  - No recursion

```
y = f(x);
y' = f'(x');
```

- Similar to Strichman et. al.

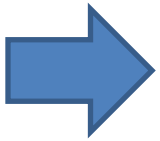
Related Work

- Brumley et al.
- Strichman et al. (SymDiff)
- DSE
- UC-KLEE

Sequential composition is bad

```
int sign(int x) {
  int sgn;
  if (x < 0)
    sgn = -1
  else
    sgn = 1
 retval = sgn
}
```

```
int sign'(int x') {
  int sgn';
  if (x' < 0)
    sgn' = -1
  else
    sgn' = 1
  if (x'==0)
    sgn' = 0
  retval' = sgn'
}
```



```
int sign;sign'(int x) {
  int x' = x;
  int sgn;
  if (x < 0)
    sgn = -1
  else
    sgn = 1
 retval = sgn
 int sgn';
  if (x' < 0)
    sgn' = -1
   else
    sgn' = 1
  if (x'==0)
    sgn' = 0
 retval' = sgn'
 assert(retval == retval')
}
```

P and P' use disjoint parts of memory

Any interleaving is sufficient (POR)

Can use sequential composition

Strichman et. al., Person et. al., Ramos & Engler

Challenge: P;P’ Inhibits Partial Disjunction

```
int sign;sign’(int x) {
  int x' = x;
  int sgn;
  if (x < 0)
    sgn = -1
  else
    sgn = 1
  retval = sgn
  int sgn';
  if (x' < 0)
    sgn' = -1
  else
    sgn' = 1
  if (x'==0)
    sgn' = 0
  retval’ = sgn‘
  assert(retval == retval’)
}
```

[{x = x’ < 0, sgn ↦ -1, sgn’ ↦ ?}]

[{x = x’ ≥ 0, sgn ↦ 1, sgn’ ↦ ?}]

states that hold equivalence for the same set of variables will be merged

$\sqcup_{eqv}$ =

[{x = x’ ≥ 0, sgn ↦ [-1,1], sgn’ ↦ ?}]

Challenge: How to compose P, P’ such that $\sqcup_{eqv}$ will maintain equivalence?

Correlating Analysis for P;P’

```
int sign;sign’(int x) {
 int x' = x;
 int sgn;
 if (x < 0)
   sgn = -1
 else
   sgn = 1
 retval = sgn
 int sgn';
 if (x' < 0)
   sgn' = -1
  else
   sgn' = 1
 if (x'==0)
   sgn' = 0
 retval’ = sgn‘
 assert(retval == retval’)
}
```

{x = x’ < 0, sgn ↦ -1, sgn’ ↦ ?}

{x = x’ ≥ 0, sgn ↦ 1, sgn’ ↦ ?}

⊔ =

{x = x’, sgn ↦ [-1,1], sgn’ ↦ ?}

Challenge: Performing a join operation in the correlating domain results in a **non-restorable loss of equivalence**

Solution: Disjunctive Correlating Domain

```
int sign;sign'(int x) {
 int x' = x;
 int sgn;
 if (x < 0)
   sgn = -1
 else
   sgn = 1
 retval = sgn
 int sgn';
 if (x' < 0)
   sgn' = -1
  else
   sgn' = 1
 if (x'==0)
   sgn' = 0
 retval' = sgn'
 assert(retval == retval')
}
```

[{x = x' < 0, sgn ↦ -1, sgn' ↦ ?}]

[{x = x' ≥ 0, sgn ↦ 1, sgn' ↦ ?}]

U =

[{x = x' < 0, sgn ↦ -1, sgn' ↦ ?}, {x = x' ≥ 0, sgn ↦ 1, sgn' ↦ ?}]

• • •

[{x = x' < 0, sgn = sgn' ↦ -1}, {x = x' > 0, sgn = sgn' ↦ 1},
{x = x' ↦ 0, sgn ↦ 1, sgn' ↦ 0}]

Challenge: Every condition splits the abstract state, does not scale

Patch

- (Why) Easier to spot the difference
  - Our approach's precision is based on syntactic similarity
- (What) A patch can be any syntactic change
  - can be fragments or compound patch
  - the smaller the better