Abstract Semantic Differencing via Speculative Correlation

NIMROD PARTUSH

ERAN YAHAV

TECHNION (HAIFA, ISRAEL)

[OOPSLA'14]

European Research Council
Established by the European Commission
Supporting top researchers from anywhere in the world

Problem: Semantic Differencing

- For two procedures P,P':
  - Prove that P and P' are equivalent
    - Produce the **same output for the same input**
  - Otherwise, produce a *useful* description of the difference

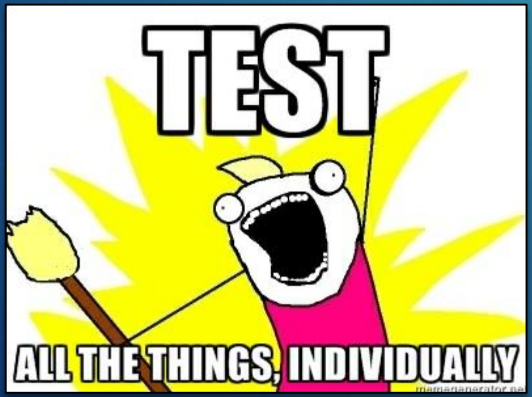
Motivation

- *"Successful software always gets changed."*

  – Frederick P. Brooks

- Program understanding & Debugging

- Test generation & pruning



- "Big Code"

Example: Sequence Printing

$first = 1 \quad last = 13$

```
static void
print_numbers_v6_9(long first, long last, ...)
{
  long i;
  for (i = 0; /* empty */; i++) {
    long x = first + i * STEP;
    if (last < x) break;
    if (i) fputs (separator, stdout);
    printf (fmt, x);
  }
  if (i)
    fputs (terminator, stdout);
}
```

$STEP = 2$

1,3,5,7,9,11,13

- Prints a sequence of numbers from $first$ to $last$ in increments of $STEP$
  - Simplified code from *Coreutils* seq.c version 6.9
- For instance for the input $\boldsymbol{first = 1, last = 13, STEP = 2}$ the procedure will print the sequence $\mathbf{1, 3, 5, 7, 9, 11, 13}$

Example: Sequence Printing

```
1  static void
2  print_numbers_v6_9(long first, long last, ...)
3  {
4    long i;
5    for (i = 0; /* empty */; i++) {
6      long x = first + i * STEP;
7      if (last < x) break;
8      if (i) fputs (separator, stdout);
9      printf (fmt, x);
10   }
11   if (i)
12     fputs (terminator, stdout);
13 }
```

```
1  static void
2  print_numbers_v6_10(long first, long last, ...)
3  {
4    bool out_of_range = (last < first);
5    if (!out_of_range) {
6      long x = first;
7      long i;
8      for (i = 1; /* empty */ ; i++) {
9        long x0 = x;
10       printf (fmt, x);
11       x = first + i * STEP;
12       out_of_range = (last < x);
13       if (out_of_range) break;
14       fputs (separator, stdout);
15     }
16     fputs (terminator, stdout);
17   }
18 }
```

- Does version 6.10 print the same sequence?
- Syntactic difference is vast
- But the output is the same: x is **equivalent** at the point of printing (9,10')

Test Equivalence

- Run the procedures with the same inputs and check outputs for equivalence

$first = 1, last = 3, STEP = 1$

```
static void
print_numbers_v6_9(long first, long last, ...)
{
  long i;
  for (i = 0; /* empty */; i++) {
    long x = first + i * STEP;
    if (last < x) break;
    if (i) fputs (separator, stdout);
    printf (fmt, x);
  }
  if (i)
    fputs (terminator, stdout);
}
```

1, 2, 3

$first = 1, last = 3, STEP = 1$

```
static void
print_numbers_v6_10(long first, long last, ...)
{
  bool out_of_range = (last < first);
  if (!out_of_range) {
    long x = first;
    long i;
    for (i = 1; /* empty */ ; i++) {
      long x0 = x;
      printf (fmt, x);
      x = first + i * STEP;
      out_of_range = (last < x);
      if (out_of_range) break;
      fputs (separator, stdout);
    }
    fputs (terminator, stdout);
  }
}
```

1, 2, 3

How do we check equivalence over *all* inputs?

- Can only show cases where procedures differ
  - Modulo cost and coverage (if you are lucky)
- Cannot **prove** equivalence for most programs

Abstract Semantic Differencing

- Use **abstract interpretation** to **prove equivalence** between two program versions
- Or **characterize their difference**
  - Find (an abstraction of) **differing programs states** that come from the same input
- **Sound**
  - Equivalence is guaranteed
- **Precise**
  - Report few false differences

Equivalence Under Abstraction

- Analyzing the programs separately may result in false equivalence
  - For instance, interpreting with a numerical relational abstraction:

$$\underbrace{\texttt{while (i < n) \{ i += 1; print(i); \}}}_{\{i \leq n\}} \neq \underbrace{\texttt{while (i < n) \{ i += 2; print(i); \}}}_{\{i' \leq n'\}}$$

```
while (i < n) {
  i += 1;
  print(i);
}
```

```
while (i < n) {
  i += 2;
  print(i);
}
```

$\{i \leq n\}$ $\neq$ $\{i' \leq n'\}$

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

Our Approach

- Analyze P and P' **together**
  - Define a *correlating semantics* that interprets the programs together
  - Interpretation is done in some **interleaving** of their steps
  - **Abstract** the correlating semantics to handle infinite-state programs
- **Search** for the interleaving that allows the abstraction to best capture equivalence

Correlating Semantics

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

```
while (i < n) {
  i += 1;
  print(i);
}
```

```
while (i < n) {
  i += 2;
  print(i);
}
```

$$\sigma = \{ i \le n,\ i \ne i',\ i' \le n' \}$$

Order Matters!

- Analysis order determines the abstract correlating semantics' ability to track equivalence
  - For example, in **sequential** order:
    - By the time one program's analysis is finished, the values have been abstracted and equivalence is lost

```
while (i < n) {
  i += 1;
  print(i);
}
```

;

Loop invariant:

iter #0: $\{i' = i\}$

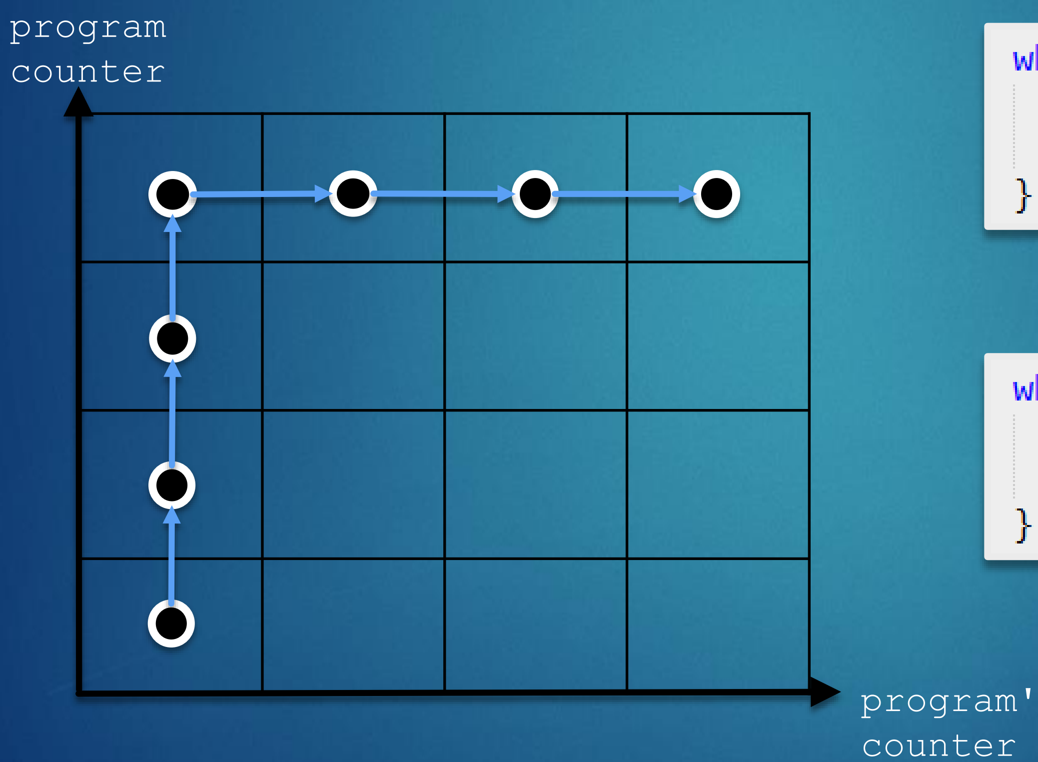
iter #1: $\{i' = i + 1\}$

iter #2: $\{i' = i + 2\}$

Abstraction: $\{i \leq n, i' = ?\}$ ☹

Choosing Program Interleaving

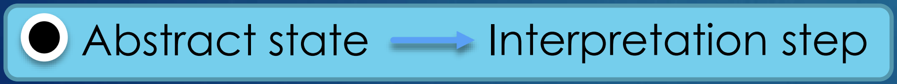
- In which order should the programs be analyzed?
  - Sequential?




```
while (i < n) {
  i += 1;
  print(i);
}
```

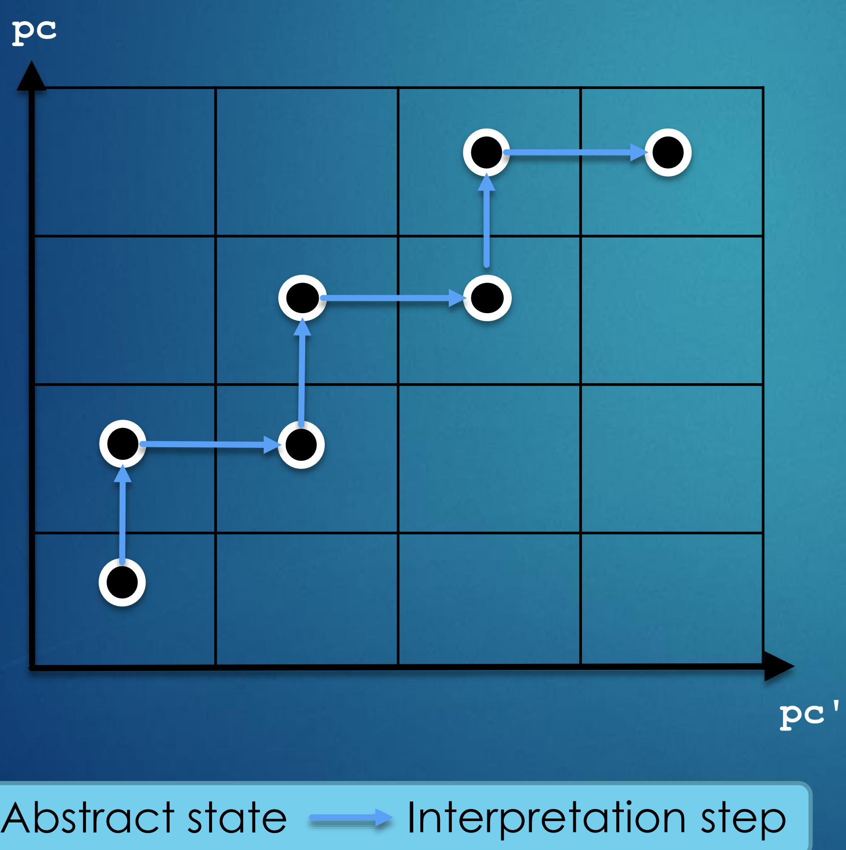
;

```
while (i < n) {
  i += 1;
  print(i);
}
```



Choosing Program Interleaving

- In which order should the programs be analyzed?
  - Lock-Step?

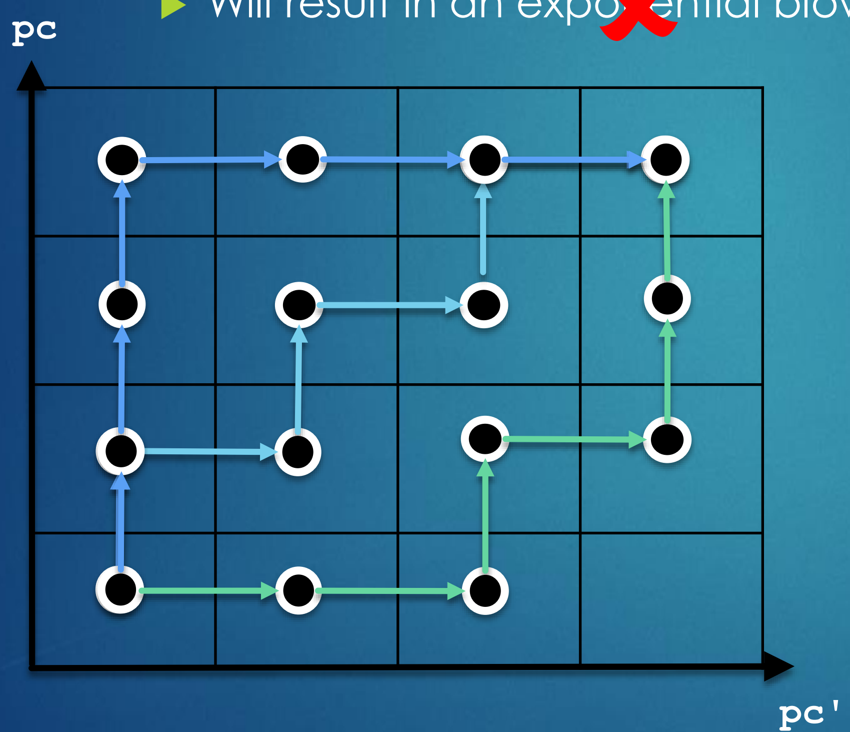



```
static void
print_numbers_v6_10(long first, long last, ...)
{
  bool out_of_range = (last < first);
  if (!out_of_range) {
    long x = first;
    long i;
    for (i = 1; /* empty */ ; i++) {
      long x0 = x;
      printf (fmt, x);
      x = first + i * STEP;
      out_of_range = (last < x);
      if (out_of_range) break;
      fputs (separator, stdout);
    }
    fputs (terminator, stdout);
  }
}
```

```
static void
print_numbers_v6_9(long first, long last, ...)
{
  long i;
  for (i = 0; /* empty */; i++) {
    long x = first + i * STEP;
    if (last < x) break;
    if (i) fputs (separator, stdout);
    printf (fmt, x);
  }
  if (i)
    fputs (terminator, stdout);
}
```

Choosing Program Interleaving

- In which order should the programs be analyzed?
  - All possible interleavings?
    - Will result in an exponential blow-up



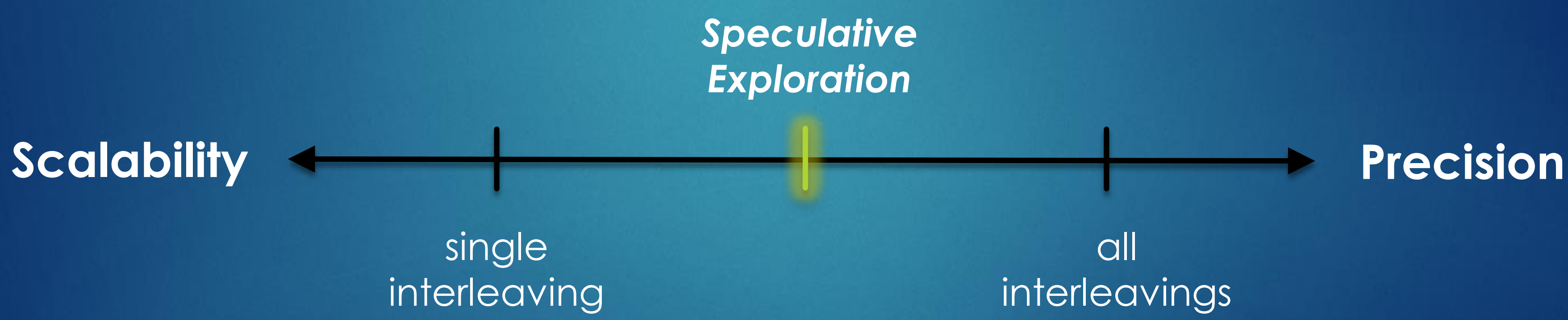
Abstract state Interpretation step

```
static void
print_numbers_v6_10(long first, long last, ...)
{
  bool out_of_range = (last < first);
  if (!out_of_range) {
    long x = first;
    long i;
    for (i = 1; /* empty */ ; i++) {
      long x0 = x;
      printf (fmt, x);
      x = first + i * STEP;
      out_of_range = (last < x);
      if (out_of_range) break;
      fputs (separator, stdout);
    }
    fputs (terminator, stdout);
  }
}
```

```
static void
print_numbers_v6_9(long first, long la[illegible], ...)
{
  long i;
  for (i = 0; /* empty */; i++) {
    long x = first + i * STEP;
    if (last < x) break;
    if (i) fputs (separator, stdout);
    printf (fmt, x);
  }
  if (i)
    fputs (terminator, stdout);
}
```

Choosing Program Interleaving

- The challenge is to find an interleaving that allows <u>maintaining equivalence under abstraction</u>
  - While avoiding exponential blow-up



Speculative Exploration

- The search for an interleaving is part of the fixed-point abstract in
  - The search driv
- The algorithm is c

Speculatively explore interpreting k steps, distributed over both programs
- 0 over P & k over P'
- 1 over P & $k-1$ over P'
- Etc.

k is the speculation window
- a parameter to the algorithm

- **while** $(worklist \neq \emptyset)$
  - $results \leftarrow$ `Speculate`$(\boldsymbol{P}, \boldsymbol{P'}, worklist, state_{\bowtie}, k)$
  - $(worklist, state_{\bowtie}) \leftarrow$ `Find_Minimal_Diff_Result`$(P, P', results)$

At the end of each speculative step, compare the k+1 results and pick the one with minimal difference
- A greedy approach

Speculative Exploration: Example

- Speculate $(k = 7)$

```c
static void
print_numbers_v6_9(long first, long last, ...)
{
  long i;
  for (i = 0; /* empty */; i++) {
    long x = first + i * STEP;
    if (last < x) break;
    if (i) fputs (separator, stdout);
    printf (fmt, x);
  }
  if (i)
    fputs (terminator, stdout);
}
```

```c
static void
print_numbers_v6_10(long first, long last, ...)
{
  bool out_of_range = (last < first);
  if (!out_of_range) {
    long x = first;
    long i;
    for (i = 1; /* empty */ ; i++) {
      long x0 = x;
      printf (fmt, x);
      x = first + i * STEP;
      out_of_range = (last < x);
      if (out_of_range) break;
      fputs (separator, stdout);
    }
    fputs (terminator, stdout);
  }
}
```

0 steps over v6.9, 7 steps over v6.10	…	3 steps over v6.9, 4 steps over v6.10	…	7 steps over v6.9, 0 steps over v6.10
$(4,11') \mapsto \{i' = 1, i = ?, x' = first' + STEP', x = ?, \ldots\}$	…	$(7,7') \mapsto \{i' = 1, i = 0, \boxed{x' = x} = first, \ldots\}$	…	$(6,4') \mapsto \{i' = ?, i = 1, x' = ?, x = first, \ldots\}$

Comparing Abstractions to find Minimal Difference

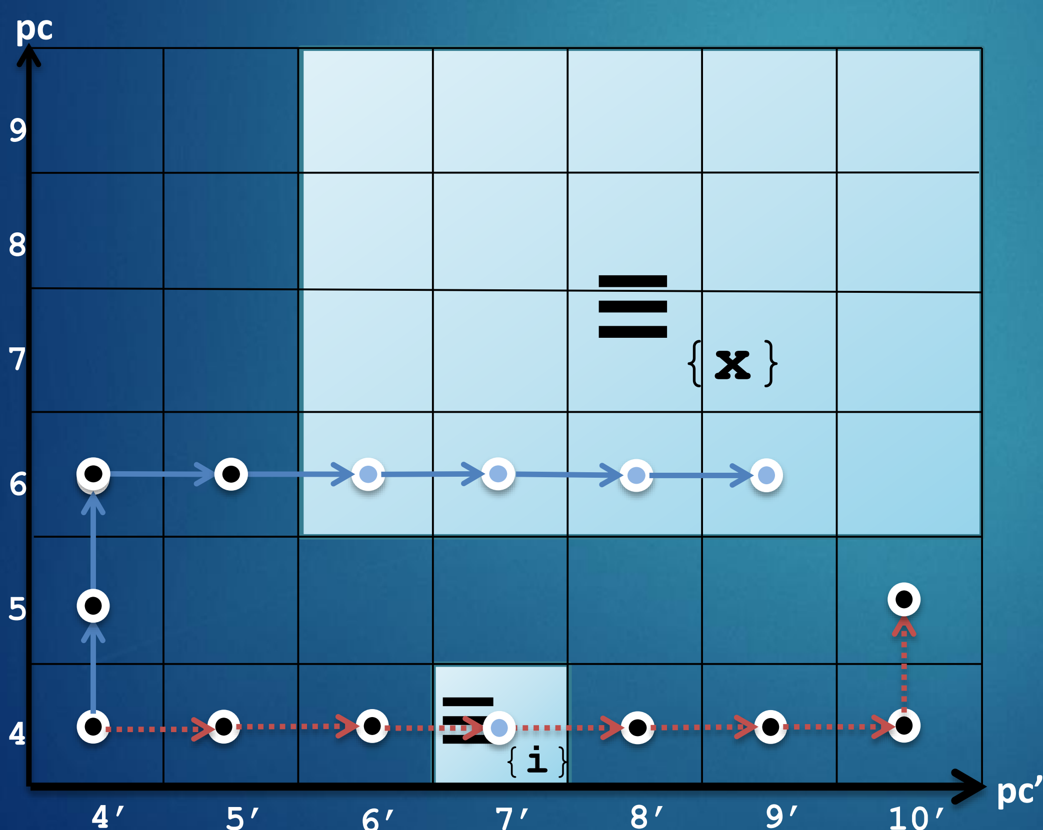
- We explored two strategies
  1. Scale-oriented: count the number of equivalent variables (denoted $\equiv_{\{v\}}$) per state
  2. Precision-oriented: Use the abstract domain (APRON Polyhedra) geometrical representation to compute difference inclusion

$$\sigma_{\bowtie 1} = \begin{array}{c} (9,4') \mapsto \{i = 0, i' = ?, x = first, x' = ?, x \le last, x_0' = x'\} \\ (6,7') \mapsto \{0 \le i, i' = 1, first \le x \le last, x' = first\} \\ \dots \end{array}$$

$$\sigma_{\bowtie 2} = \begin{array}{c} (5,5') \mapsto \{i = i' = 1, x = ?, x' = first + STEP, x_0' = ?\} \\ (7,9') \mapsto \{0 \le i = i', x \le last, x' = {x_0}'\} \\ \dots \end{array}$$

$\dots$

Speculative Exploration: Visualization ($k = 7$)




```c
static void
print_numbers_v6_9(long first, long last, ...)
{
  long i;
  for (i = 0; /* empty */; i++) {
    long x = first + i * STEP;
    if (last < x) break;
    if (i) fputs (separator, stdout);
    printf (fmt, x);
  }
  if (i)
    fputs (terminator, stdout);
}
```

```c
static void
print_numbers_v6_10(long first, long last, ...)
{
  bool out_of_range = (last < first);
  if (!out_of_range) {
    long x = first;
    long i;
    for (i = 1; /* empty */ ; i++) {
      long x0 = x;
      printf (fmt, x);
      x = first + i * STEP;
      out_of_range = (last < x);
      if (out_of_range) break;
      fputs (separator, stdout);
    }
    fputs (terminator, stdout);
  }
}
```

Producing Description of Difference

- The speculative algorithm is geared towards finding minimal difference (and not **strictly** equivalence)
  - A *usable description of difference* is produced even if full equivalence does not hold

```
static void
print_numbers_
{
  long i;
  for (i = 0;
    long x = f
    if (last <
    if (i) fpu
    printf (fm
  }
  if (i)
    fputs (terminator, stdout);
}
```

$$(9,10') \mapsto$$
$$\{print\_extra\_number, x' = x, i' = i + 1, x \leq last\}$$
$$\vee$$
$$\{\neg print\_extra\_number, x' = x + STEP, i' = i + 1, x \leq last\}$$

```
static void
print_numbers_v6_10(long first, long last, ...)
{
  bool out_of_range = (last < first);
  if (!out_of_range) {
                                   i++) {
                                   ;
                                   x);
      (out_of_range){
        bool print_extra_number = Check(x0, x);
        if (!print_extra_number) break;
      }
      fputs (separator, stdout);
    }
    fputs (terminator, stdout);
  }
}
```

Evaluation

	Function	#LOC	#Patch	#Loops	Time
Coreutils	print_numbers	23	7-,13+	1	00:11 (k=2)
	cache_fstatat	17	2-,4+	0	00:03 (k=1)
	set_owner	51	2-,4+	0	00:02 (k=2)
	fmt	42	5-,5+	1	00:22 (k=2)
	md5sum	40	0-,3+	3	13:31 (k=2)
	char_to_clump	111	2-,12+	3	19:09 (k=2)
	savewd	86	0-,1+	0	00:46 (k=2)
Linux, Firefox	addr	77	1-,2+	0	00:17 (k=1)
	SetTextInternal	47	0-,3+	1	11:28 (k=3)
git	get_sha1_basic v1	145	3-,10+	2	118:01 (k=2)
	get_sha1_basic v2	149	2-,20+	2	*TO* (2H)
	get_path_prefix	22	2-,3+	1	29:12 (k=3)
	boot_attr v1	77	7-,4+	0	08:08 (k=4)
	boot_attr v2	74	5-,7+	0	06:04 (k=4)
	read_attr	32	1-,4+	1	05:42 (k=2)
	ll_binary_merge	37	8-,24+	1	00:53 (k=1)
	write_zip_entry	340	1-,4+	3	07:32 (k=2)
related	DDEC	10	3-,3+	1	00:13 (k=1)
	DSE	7	2-,3+	1	00:09 (k=1)
	RegVer	10	4-,4+	1	00:07 (k=1)
	SymDiff	32	5-,4+	0	00:04 (k=1)

Conclusion

Contact us!
{nimi,yahave}@cs.technion.ac.il

- The interleaving\matching problem is closely coupled with semantic diff & equivalence
  - Previous approach mainly use syntactic cues
- New approach: search for an interleaving
  - With an equivalence criteria to drive the search
- A description of difference is produced, instead of a binary yes/no for equivalence
  - Useful for program understanding, debugging etc.
- Available on github

Proposed Questions

1. Shouldn't there (still) be some exponential blow-up here??
2. Where does the variable matching come from?
3. Can you talk about how related work find their interleaving?
4. How do you handle function calls?
5. How do you handle heap\array data?
6. Why were the k's in your evaluation so small?
7. Can you elaborate on the precise method for comparing abstractions?
8. You seem to have used a disjunctive domain, how did you $\sqcup$ and ∇ it?
9. Won't you miss interleavings?
10. How does the approach scale (inter-procedurally)?
11. Is the analysis proportional to the size of change\program\both?

Un-interpreted Functions

- Function calls are handled modularly

```
x = foo(y,z)
```

  - If foo was proven to be equivalent, and so are y & z, the result will be equivalent
    - Otherwise, anything (T) is ossible for x

$$\frac{\equiv_{foo} \quad \equiv_{\{v_1,v_2\}}}{\equiv_{foo(v1,v2)}}$$

- Array and heap access are modeled similarly

```
a = array[idx]
```

$$\frac{\equiv_{array} \quad \equiv_{idx}}{\equiv_{array[idx]}}$$

Related work: program matching

- Symbolic Execution based approaches [DSE, UC-KLEE]:
  - Bound loop iteration
  - try to explore all bound paths
  - Check equivalence on each path
- Recursion-Rule based approaches [SymDiff]:
  - Transform loops to function calls
  - Use function calls as matching locations
    - i.e. try and prove the inputs to each function call are equivalent
- Data\Trace based approaches [DDEC]:
  - Use run traces variable values to infer a bi-simulation relation of the two programs

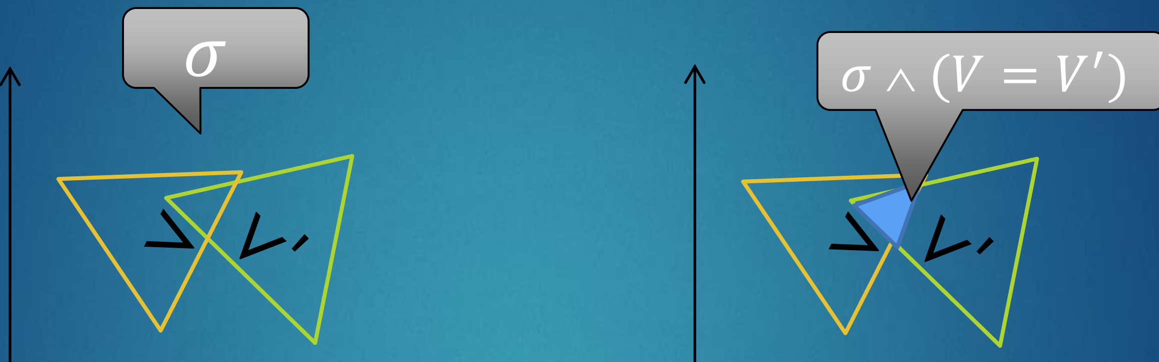
Equivalence-based Partitioning

- We use a disjunctive domain to allow separating **equivalent paths** from **differing paths**

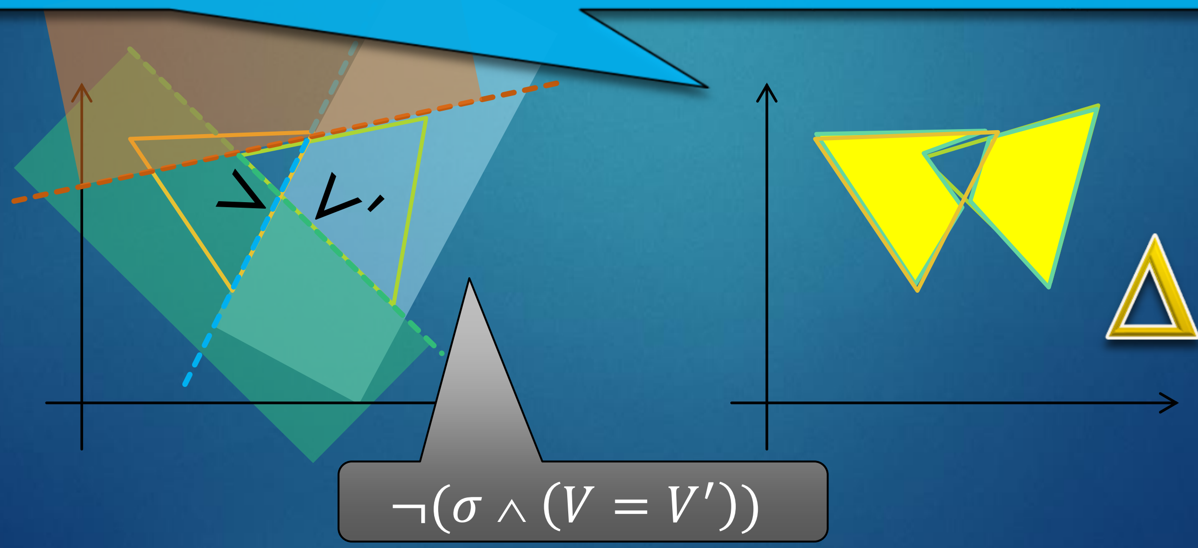
$$\{print\_extra\_number, x' = x, i' = i + 1, x \leq last\}$$
$$\vee$$
$$\{\neg print\_extra\_number, x' = x + STEP, i' = i + 1, x \leq last\}$$

- How do you Join such a domain?
  - which sub-states are joined with which?
- We Join and Widen 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)

Finding Minimal Difference



The abstract differences Δ_1, Δ_2 is computed for both states, and then containment $\Delta_1 \sqsubseteq \Delta_2$ (or vice-versa) is checked



Partial Order Reduction

- Exploring **all interleavings** over two programs within k steps would yield 2^k results
  - Since each step can be performed on either P or P'
- For the most part, a single representative is sufficient
  - Meaning foreach $0 < i \leq k$ we explore first interpreting all i steps over P and then $k - i$ over P'
    - No alternation in-between
    - Resulting in $k + 1$ results
  - Since the domain is commutative and join\widen is only performed after each speculative step
- This does mean we miss some interleavings
  - But did not pose an issue in our benchmarks
  - A small sacrifice to make for scaling

Full Program Equivalence Checking

- For full program equivalence, we currently use the modular approach
  - And in the case the callee differs among versions, we assume T
    - Future: use the description of difference instead, somehow
- This is the state of the art for equivalence checking
  - The only full-program approaches (known to this presenter :) are symbolic execution based ones, that try to pin-point differing paths
    - Usually get poor coverage
    - Generally unable to prove equivalence

Interesting Diff

```
int foo(int x, int y) {
    int z = 0,w,o;
    z++;

    ...

    for (x = 0; x < 2*y ; ++x)
        w += 2;

    ...

    if (y > 42) {
        z -= 13;
        o = z + zoo(3);
        x--;
    }

    ...

}
```

```
int foo`(int x`, int y`) {
    int z` = 1,w`,o`;
    z`--;

    ...

    for (x` = 0; x` < y` ; ++x`)
        w` += 4;

    ...

    if (y` > 47) {
        z` -= 12;
        o` = zoo(3);
        x`++;
    }

    ...

}
```

$$\{\equiv_{\{x,y,w,o\}}, z' = z - 1\}$$

$$\{\equiv_{\{y,w,o\}}, x = 2x', z' = z - 1\}$$

$$\{\equiv_{\{y,w,o\}}, y \leq 42, x = 2x', z' = z - 1\}$$

$$\vee$$

$$\left\{\begin{matrix} \equiv_{\{y,w\}}, 47 \geq y \geq 42, z' = z - 14, \\ o' = z' + zoo(3), \ldots \end{matrix}\right\}$$

$$\vee$$

$$\{\equiv_{\{y,w,z\}}, y > 47, o = o' + z, x = 2x' - 2\}$$