Fluid Updates: Beyond Strong vs. Weak Updates

Isil Dillig, Thomas Dillig, Alex Aiken
Stanford University

Strong and Weak Updates

- In *static analysis*, there is a distinction between two kinds of updates to *abstract* memory locations:

Strong vs *weak*

- Consider points-to edge from abstract location A to B:

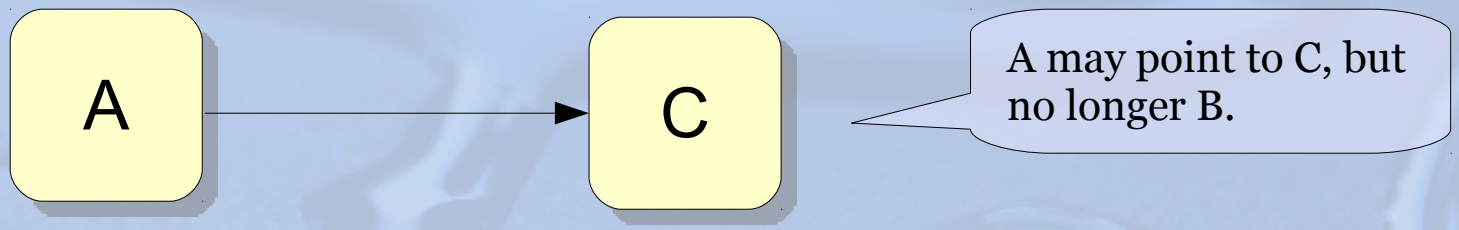
- A *strong update* removes existing points-to edges from location A and adds new edge to C.

Strong and Weak Updates

- In *static analysis*, there is a distinction between two kinds of updates to *abstract* memory locations:

Strong vs *weak*

- Consider points-to edge from abstract location A to B:



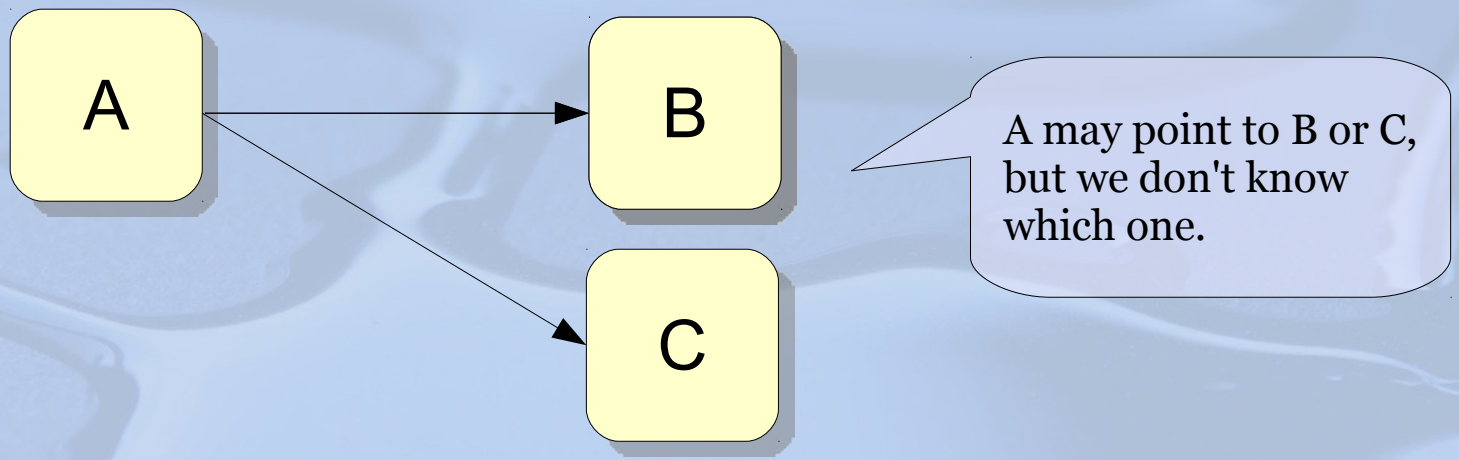
- A *strong update* removes existing points-to edges from location A and adds new edge to C.

Strong and Weak Updates

- In *static analysis*, there is a distinction between two kinds of updates to *abstract* memory locations:

Strong vs *weak*

- Consider points-to edge from abstract location A to B:



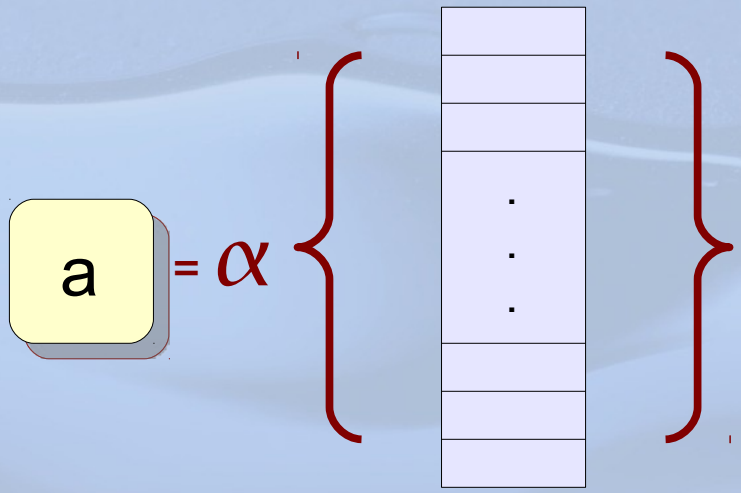
- A *weak update* adds a new edge without removing existing edges.

When are Strong Updates Applicable?

- In general, it is preferable to apply strong updates because it is more precise.
- However, we can only apply strong updates if the abstract location corresponds to *one concrete location*.
  - This makes it difficult to apply strong updates to elements of unbounded data structures, such as arrays.

Statically unknown size n.

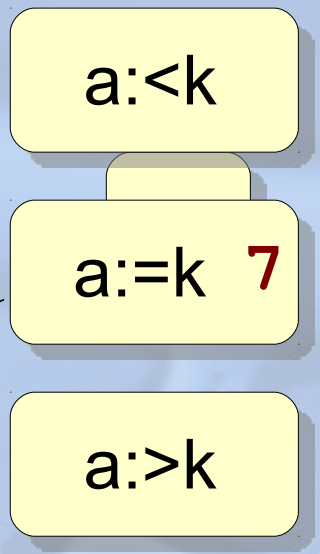
```
int* a = malloc(n*sizeof(int))
```



Strong Updates to Arrays

- Many approaches overcome this difficulty by creating *partitions* of the array.

```
a[k]=7;
```



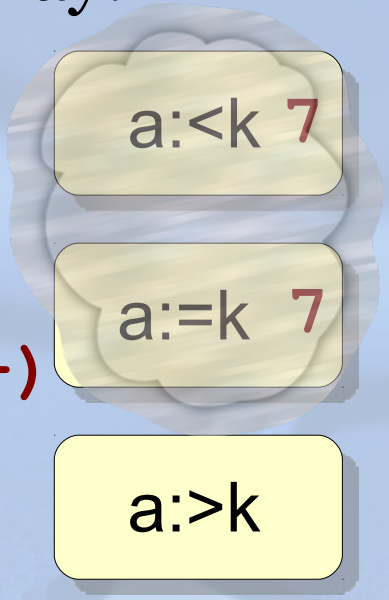
This abstract location contains only the k'th element of the array.

Strong Updates to Arrays

- Many approaches overcome this difficulty by creating *partitions* of the array.

```
for(int k=0;
    k < size; k++)
  a[k] = 7;
```

We need to avoid creating an unbounded number of partitions.



Strong Updates to Arrays

- Many approaches overcome this difficulty by creating *partitions* of the array.

```
for(int k=0;
    k < size; k++)
  a[k] = 7;
```

a:<=k 7

a:>k

Finitizes the number of partitions.

Drawbacks

- This approach has some drawbacks:
  - Can create *large number* of explicit partitions
    - ⇨ Limits scalability
  - *Heuristics* to decide when to focus/blur
    - ⇨ Can be difficult to automate
  - *„Shape"* of partitions are *fixed a priori*
    - ⇨ Typically contiguous ranges
- The rest of this talk is about a *heap abstraction* and *update mechanism* that does not have these drawbacks.

Symbolic Heap Abstraction

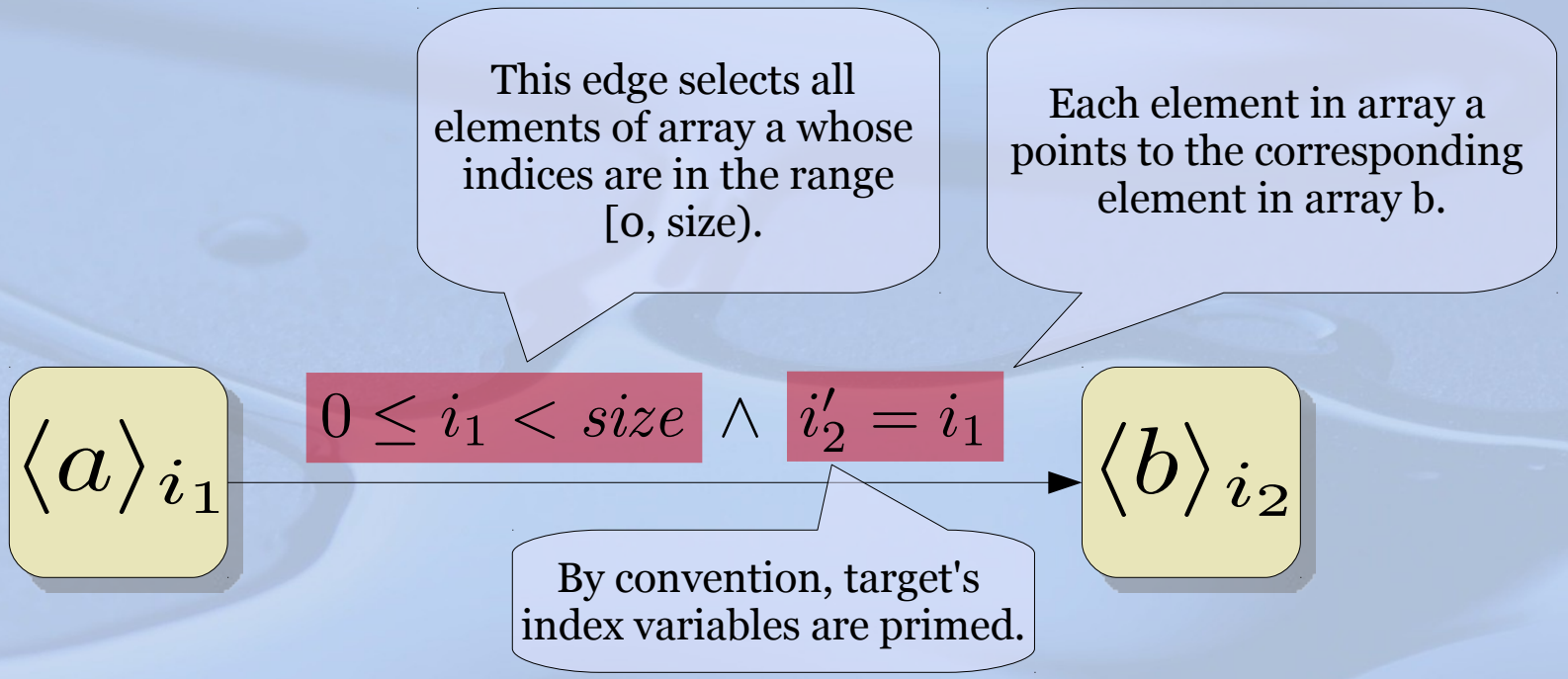
- Key Idea #1:
  - Arrays are represented by abstract locations that are qualified by *index variables*.

$$\langle a \rangle_i$$

  - These index variables range over *possible indices* of the array.

Symbolic Heap Abstraction

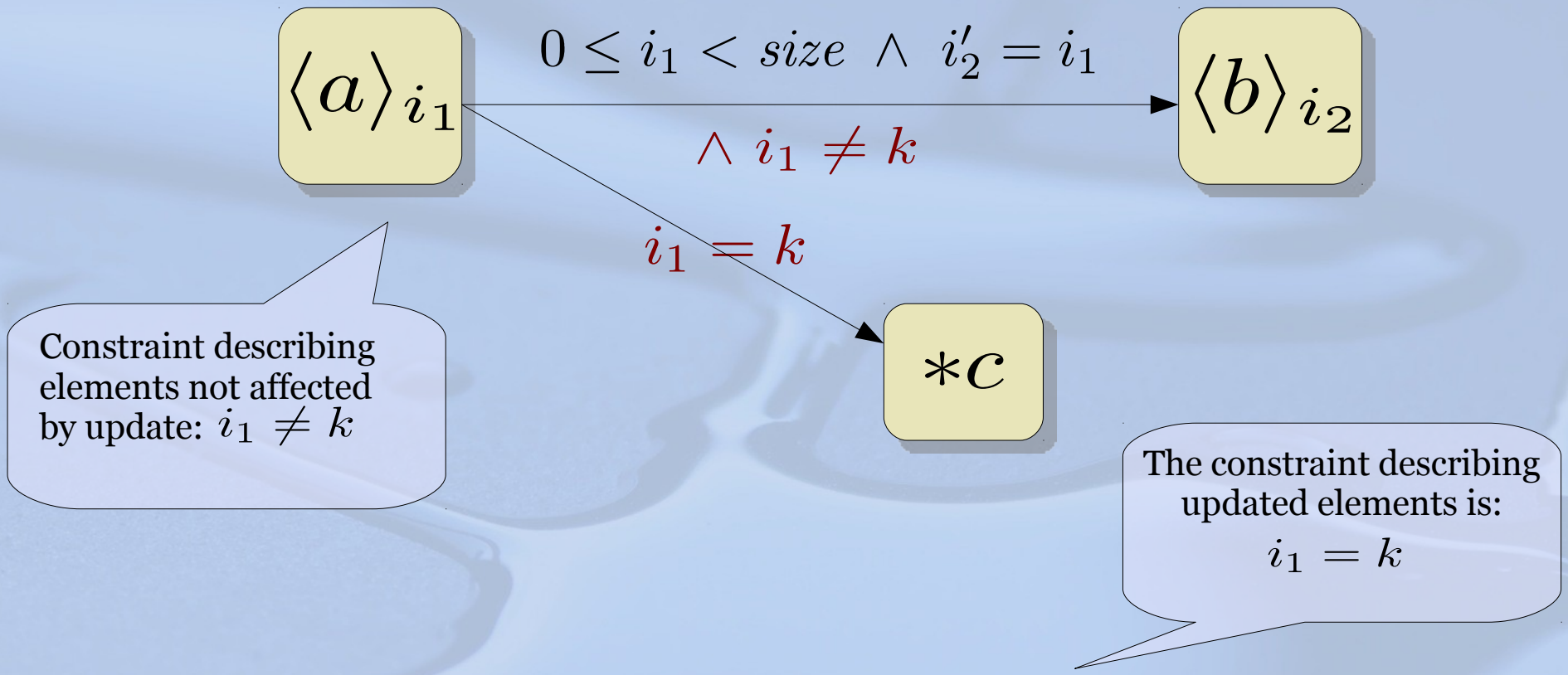
- Key Idea #2:
  - *Constraints on index variables* select which concrete elements in the source location point to which concrete elements in the target location.



Fluid Updates

- To perform a *fluid update* on the symbolic heap:
  - Compute a constraint Φ specifying which elements in an abstract location A are modifed by the update.
  - Now, the negation of Φ, $\neg\Phi$, specifies the concrete elements in A *not affected* by the update.
  - Hence, a fluid update conjoins $\neg\Phi$ with all existing edges from A, while adding new edge under Φ.

Fluid Update Example



Consider a statement: `a[k] = c;`

What if we aren't sure which elements are updated?

- In the previous example, we could precisely describe the *exact* set of elements that were written to.
- In general, we cannot always precisely specify which elements are updated.

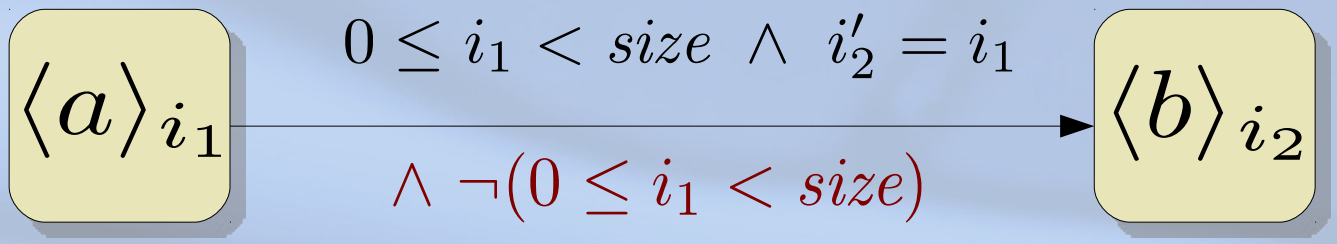
```
for(i=0; i<size; i++)
{
    if(rand())
        a[i] = NULL;
}
```

All elements in range [0, size) *may* be set to NULL, but no element *must* be set to NULL.

So, an *overapproximation* of the elements updated in the loop is:

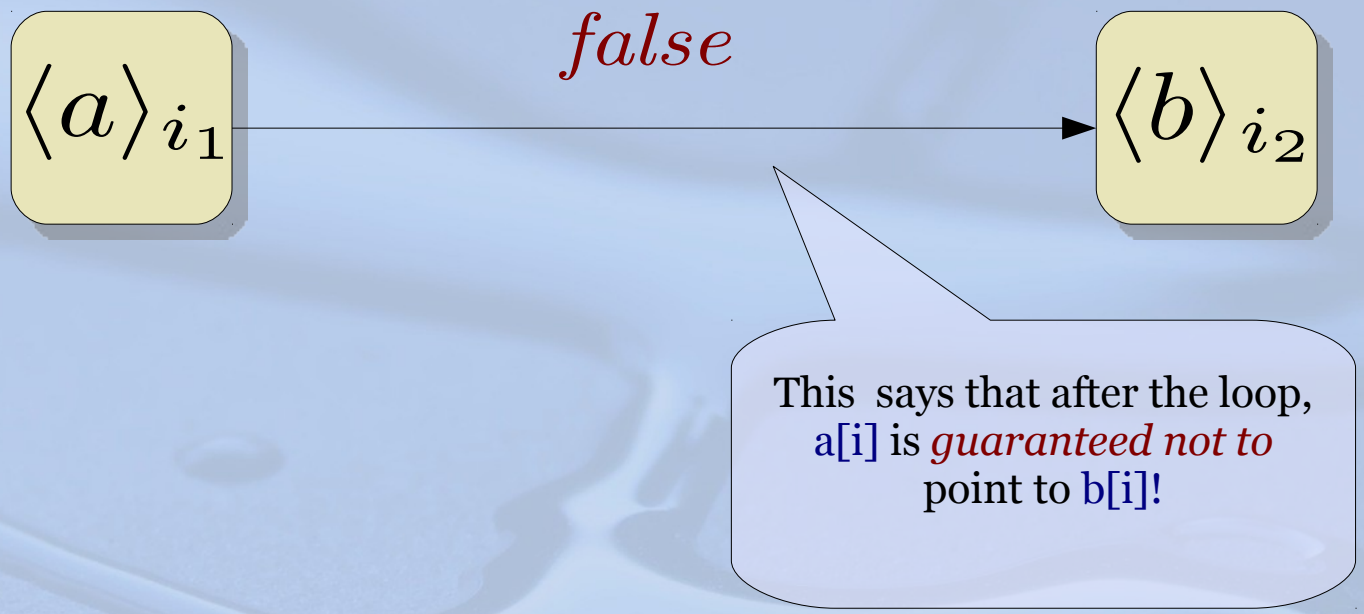
$0 \leq i_1 < size$

Fluid Update with Overapproximations



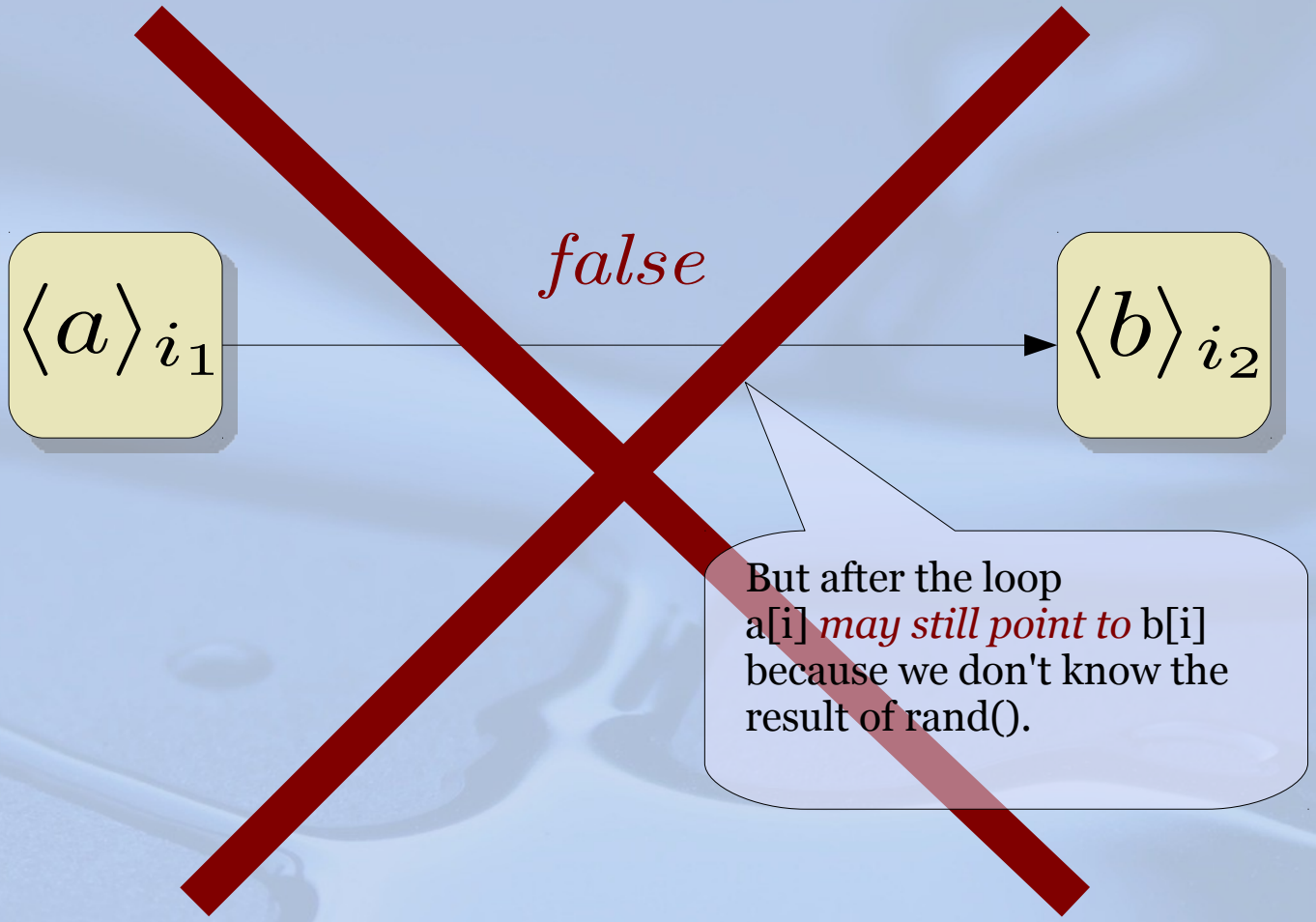
Suppose we *erroneously* use the overapproximation for the update

Fluid Update with Overapproximations



Suppose we *erroneously* use the overapproximation for the update

Fluid Update with Overapproximations



Suppose we *erroneously* use the overapproximation for the update

What went wrong?

- The *negation of an overapproximation* is an *underapproximation*.

- Hence, if the fluid update uses an overapproximation, we underapproximate the set of elements not affected by the update.

UNSOUND!

Bracketing Constraints

- Solution: Constraints in the symbolic heaps are pairs of constraints, called *bracketing constraints*:

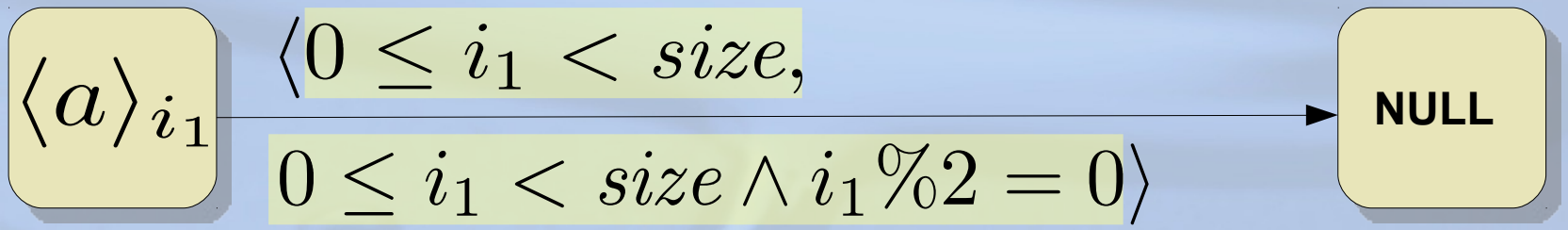
$$\langle \phi_{may},\ \phi_{must} \rangle$$

This specifies which elements in the source location ***may*** point to which elements in the target.

This specifies which elements in the source location ***must*** point to which elements in the target.

Bracketing Constraint Example

Any element in the range
[0, size) ***may*** point to null.



All ***even*** elements in the range
[0, size) ***must*** point to null.

Bracketing Constraint Example

$\langle a \rangle_{i_1}$ $\langle 0 \leq i_1 < size,$ $0 \leq i_1 < size \wedge i_1 \% 2 = 0 \rangle$ → NULL

But, ***odd*** elements in the range [0, size) ***may or may not*** point to null.

Fluid Update Using Bracketing Constraints

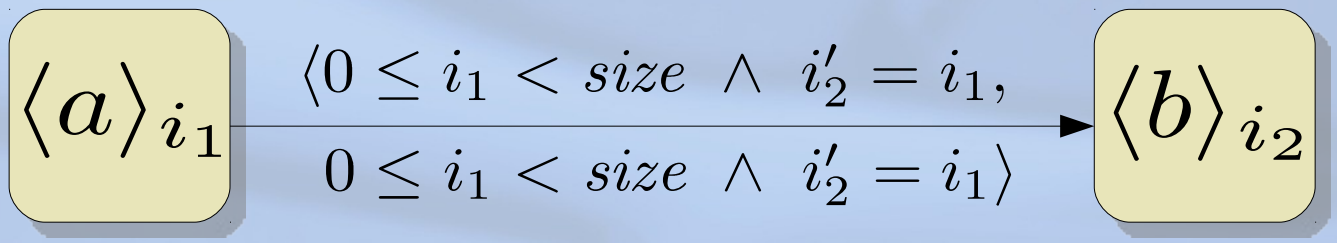
- If we use bracketing constraints, the fluid update operation described earlier is sound.
  - This is because negating a bracketing constraint yields correct over- and underapproximations.

- In particular, the negation of a bracketing constraint $\langle \phi_{may},\ \phi_{must} \rangle$ is given by:

$$\neg \langle \phi_{may},\ \phi_{must} \rangle$$
$$\Leftrightarrow$$
$$\langle \neg \phi_{must}?,\ \neg ?\phi_{may} \rangle$$

Fluid Update Using Bracketing Constraints

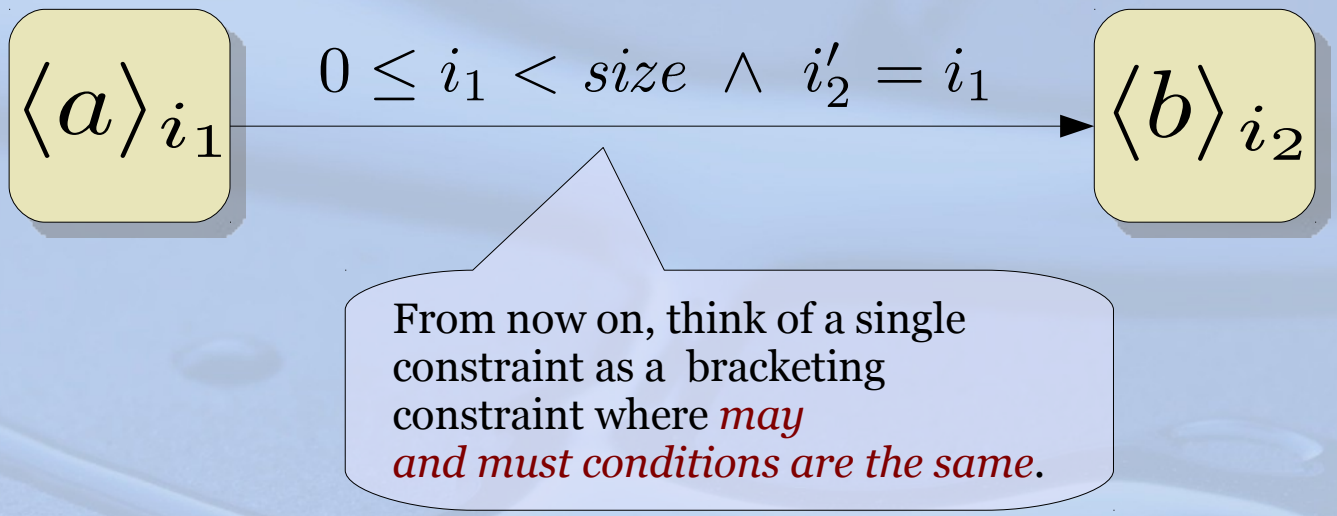
- Consider again the following initial symbolic heap:



If a bracketing constraint $\langle \phi_{may}, \ \phi_{must} \rangle$ is *precise*, i.e., $\phi_{may} \Leftrightarrow \phi_{must}$ then we write a single constraint instead of a pair.

Fluid Update Using Bracketing Constraints

- Consider again the following initial symbolic heap:



Fluid Update Using Bracketing Constraints

- Consider again the following initial symbolic heap:

The negation of this constraint is:

$\langle \neg false, \ \neg(0 \leq i_1 < size) \rangle$

A bracketing constraint describing the update condition is:

$\langle 0 \leq i_1 < size, \ false \rangle$

```
for(i=0; i<size; i++)
{
    if(rand())
        a[i] = NULL;
}
```

Fluid Update Using Bracketing Constraints

- Consider again the following initial symbolic heap:

$$\langle a \rangle_{i_1} \xrightarrow{0 \le i_1 < size \ \wedge \ i_2' = i_1} \langle b \rangle_{i_2}$$

The negation of this constraint is:

$\langle true, \ i_1 < 0 \vee i_1 \ge size \rangle$

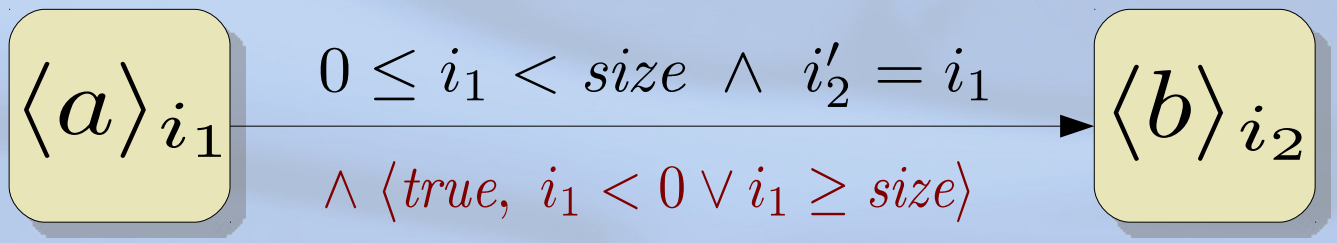
```
for(i=0; i<size; i++)
{
    if(rand())
        a[i] = NULL;
}
```

A bracketing constraint describing the update condition is:

$\langle 0 \le i_1 < size, \ false \rangle$

Example Revisited

- Consider again the following initial symbolic heap:

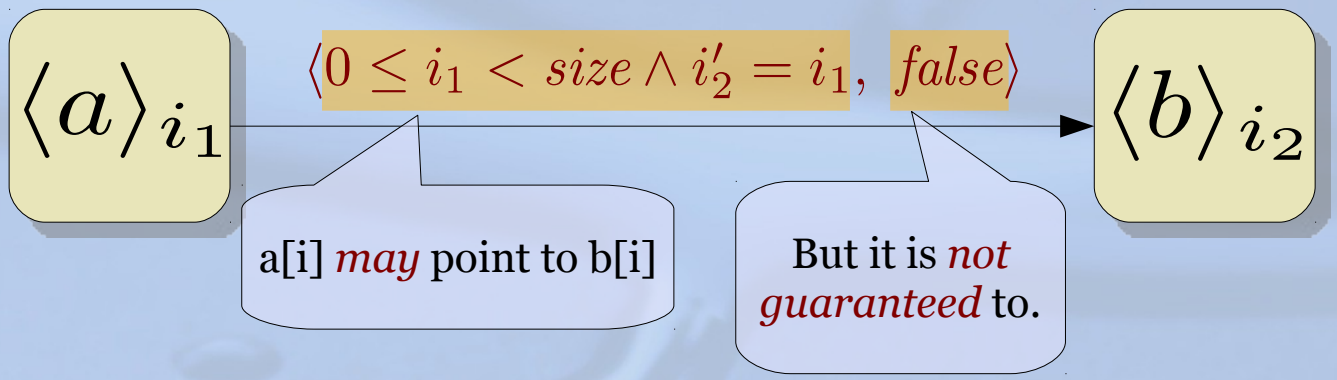



```
for(i=0; i<size; i++)
{
    if(rand())
        a[i] = NULL;
}
```

A bracketing constraint describing the update condition is:
$\langle 0 \leq i_1 < size, \ false \rangle$

Example Revisited

- Consider again the following initial symbolic heap:

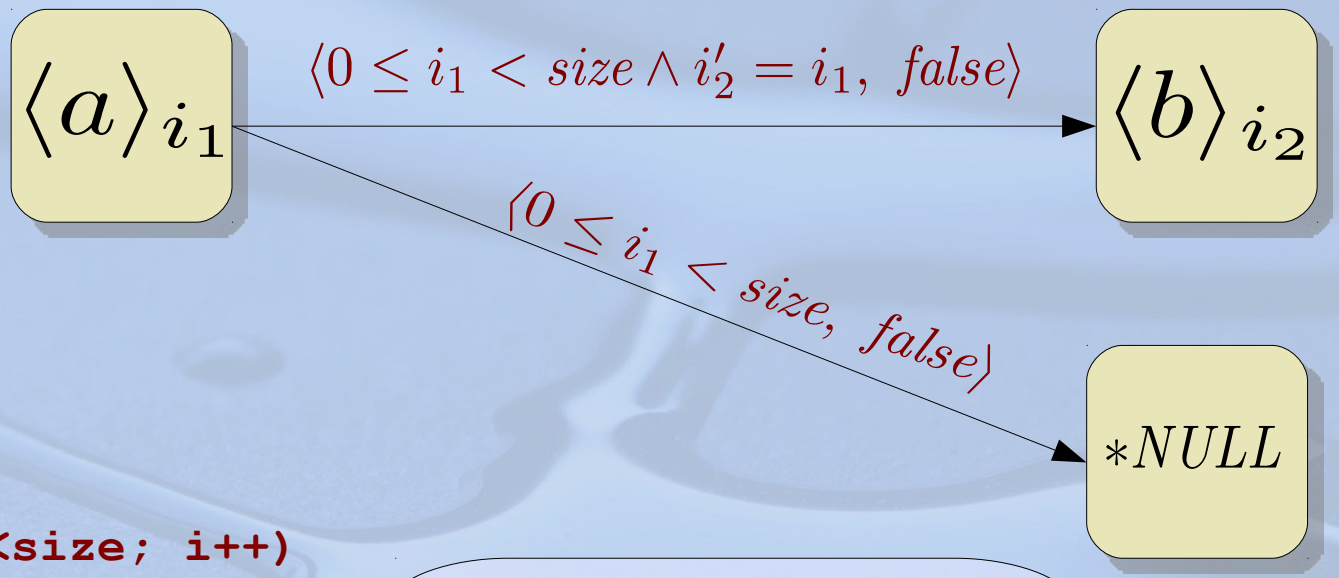



```
for(i=0; i<size; i++)
{
    if(rand())
        a[i] = NULL;
}
```

A bracketing constraint describing the update condition is:
$\langle 0 \le i_1 < size,\ false \rangle$

Example Revisited

- Consider again the following initial symbolic heap:




```
for(i=0; i<size; i++)
{
    if(rand())
        a[i] = NULL;
}
```

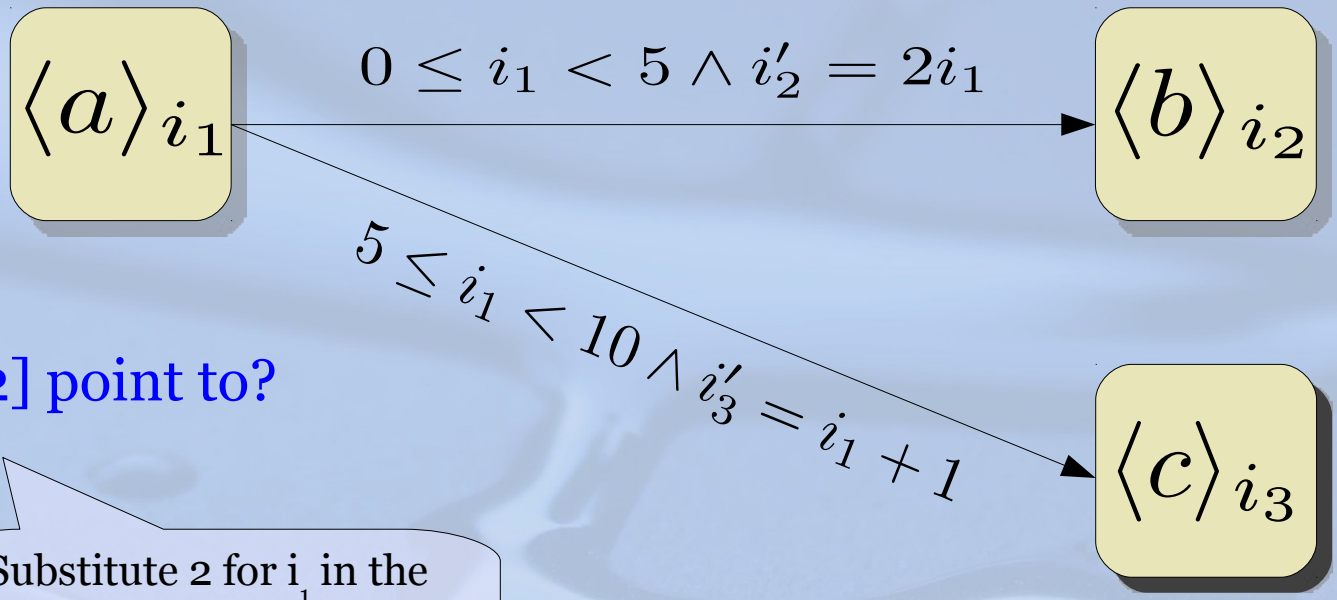
A bracketing constraint describing the update condition is:
$\langle 0 \leq i_1 < size,\ false \rangle$

Interpreting the Symbolic Heap

- So far, we described what a symbolic heap looks like and defined a fluid update operation on this heap.
- But how do we interpret/understand the facts that are encoded by the symbolic heap?

Interpreting the Symbolic Heap

- Consider the following:

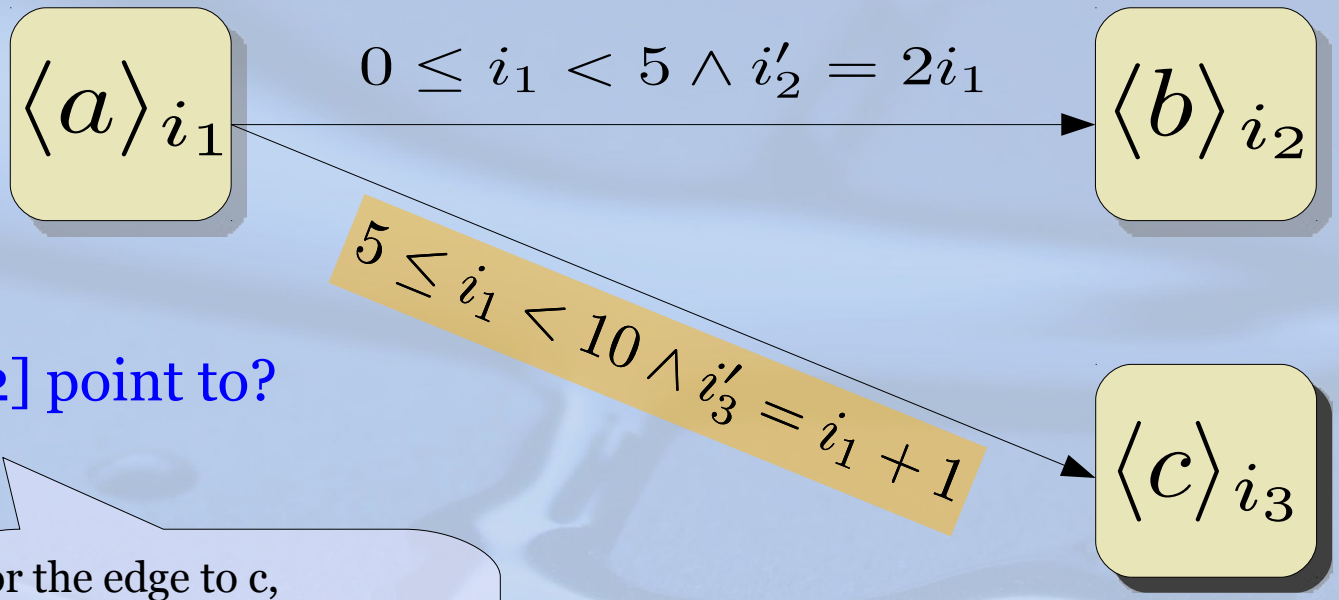


Where does a[2] point to?

Substitute 2 for i_1 in the edge constraints.

Interpreting the Symbolic Heap

- Consider the following:

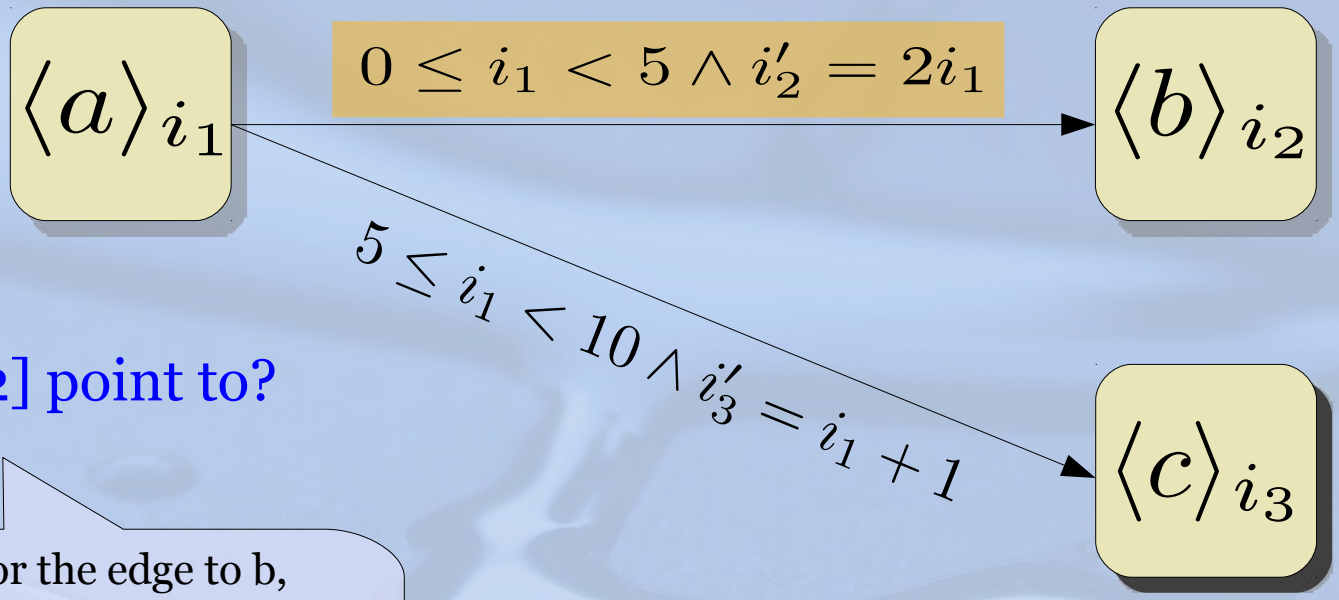


Where does a[2] point to?

For the edge to c,
the constraint 5 <= 2
is unsatisfiable, hence, a[2]
cannot point to c[3].

Interpreting the Symbolic Heap

- Consider the following:

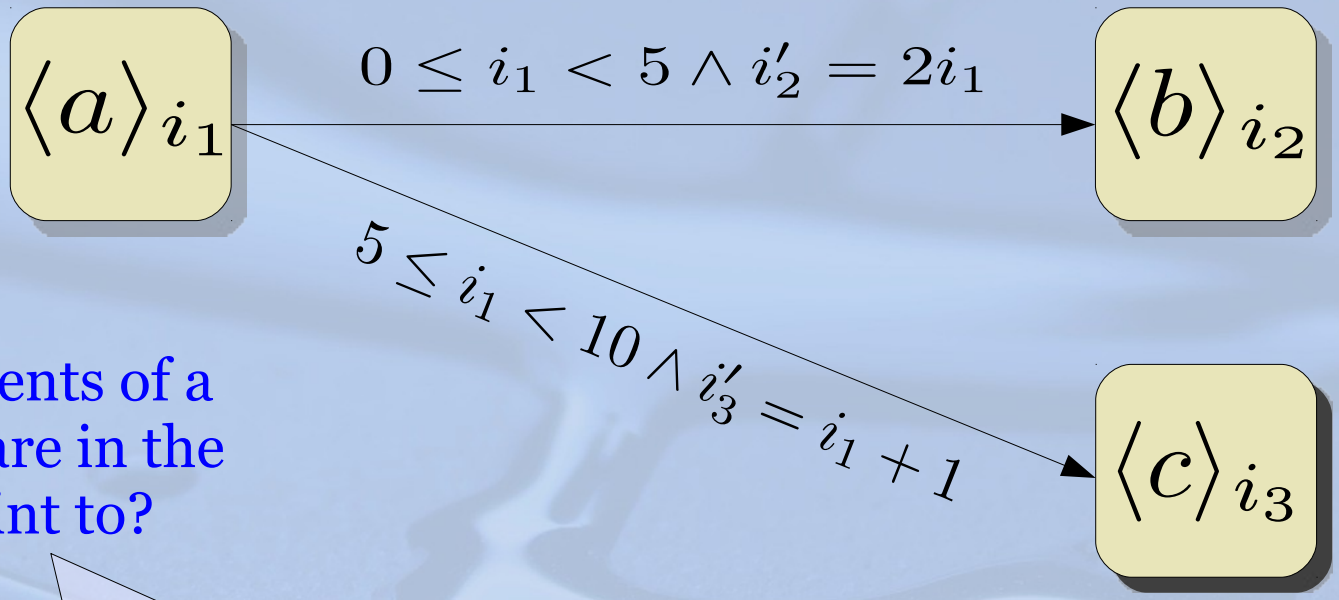


Where does a[2] point to?

For the edge to b, substituting 2 yields $i_2' = 4$; so a[2] points to b[4].

Interpreting the Symbolic Heap

- Consider the following:



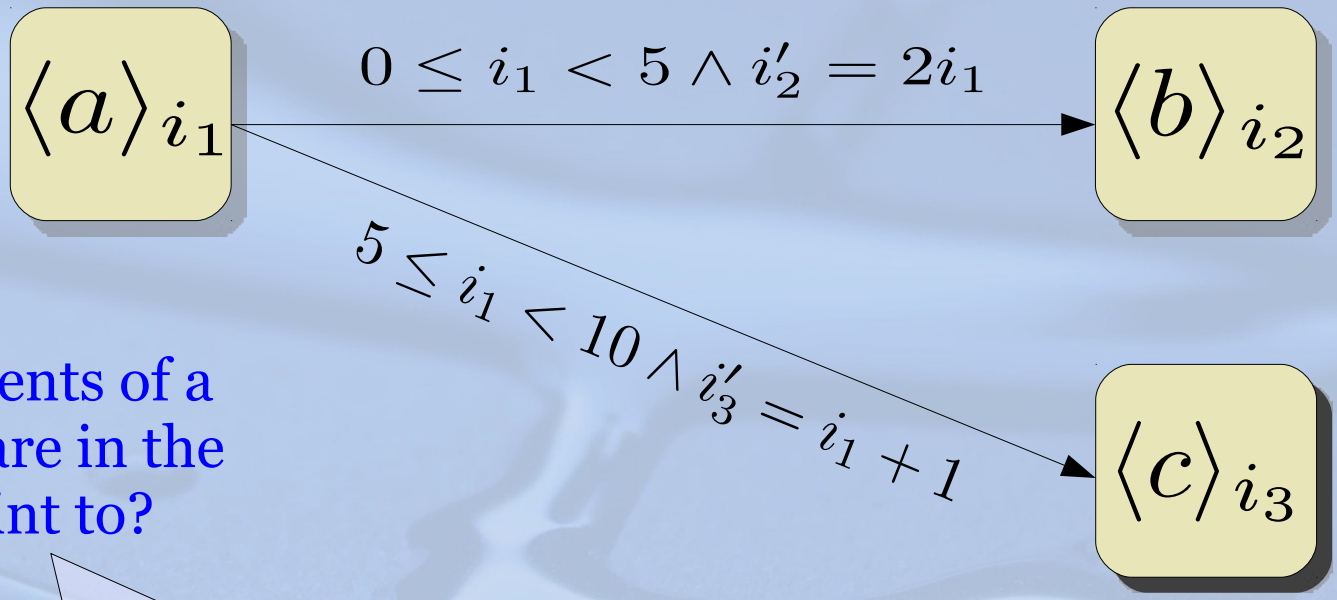
Where do elements of a whose indices are in the range [0, 3] point to?

In general, we need to know the points-to targets of those elements whose indices *satisfy some constraint*.

Here, $0 \leq i_1 \leq 3$

Interpreting the Symbolic Heap

- Consider the following:

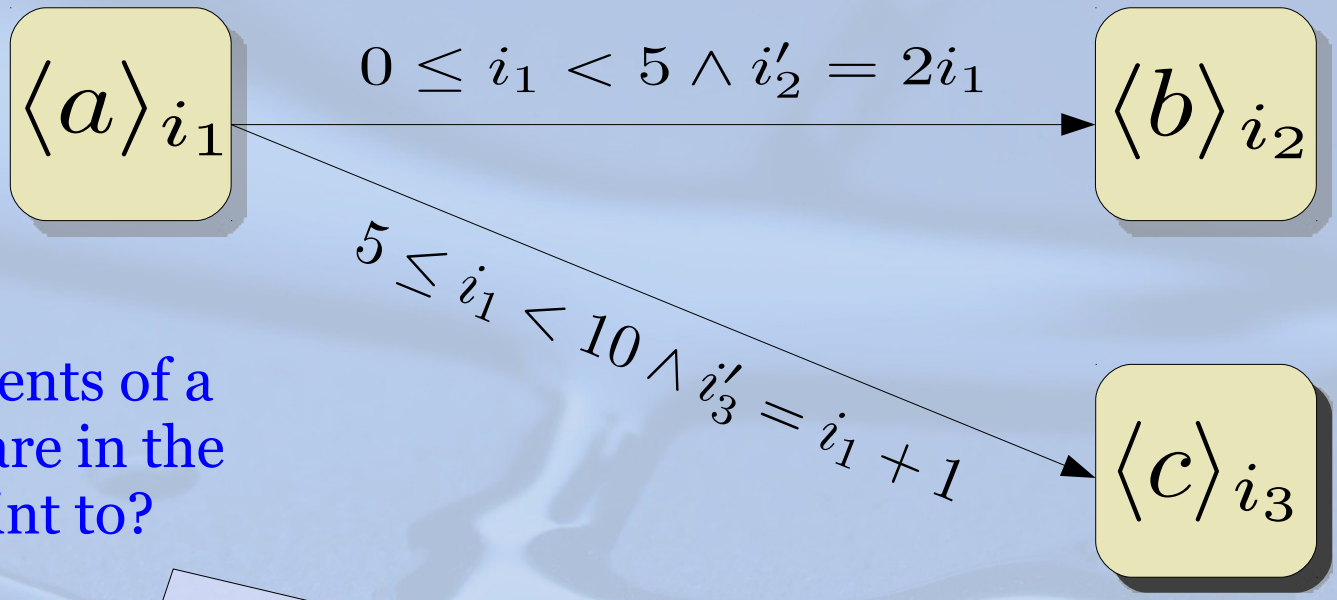


Where do elements of a whose indices are in the range [0, 3] point to?

So, we need a generalized form of substitution, i.e., *existential quantifier elimination.*

Interpreting the Symbolic Heap

- Consider the following:



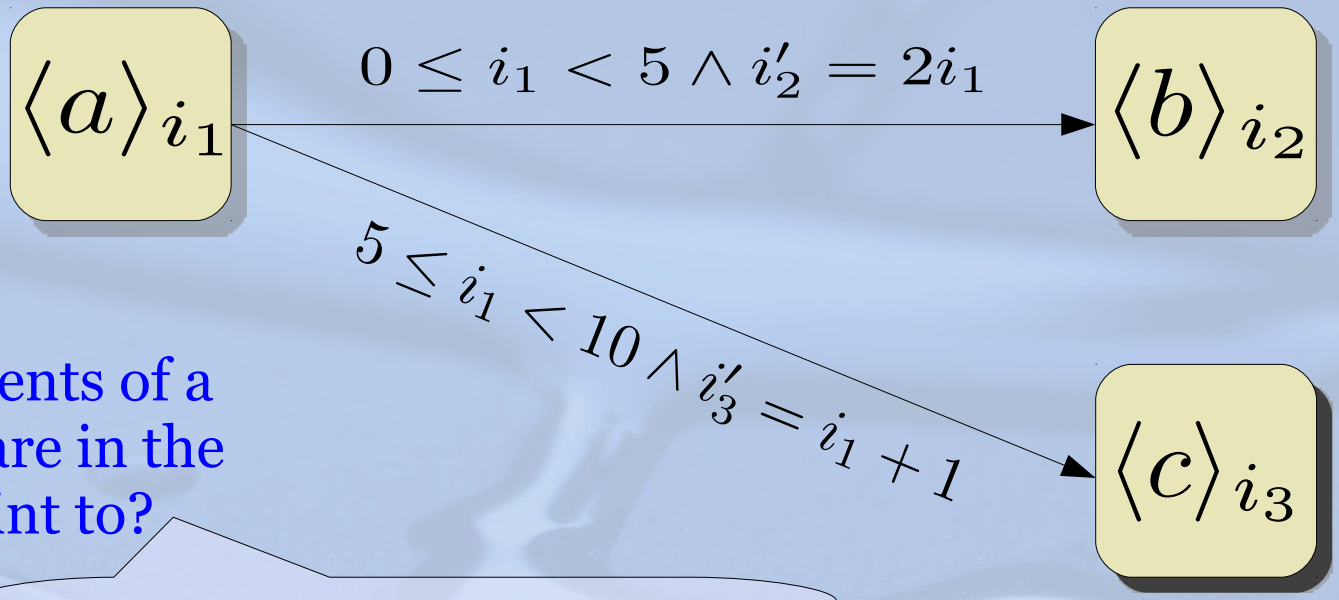
Where do elements of a whose indices are in the range [0, 3] point to?

We can determine the points-to targets of elements whose indices are in [0, 3], by *eliminating the quantifier* from the following formula (for the edge to b):

$$\exists i_1.\ 0 \leq i_1 \leq 3 \ \wedge (\ 0 \leq i_1 < 5 \ \wedge \ i_2' = 2i_1)$$

Interpreting the Symbolic Heap

- Consider the following:



Where do elements of a whose indices are in the range [0, 3] point to?

This yields: $i_2' \% 2 = 0 \ \wedge \ 0 \leq i_2' \leq 6$

Elements of a in range [0,3] point to *even* elements of b *in range [0, 6]*.

Interpreting the Symbolic Heap

- To summarize, *traversing edges* (i.e., going from one abstract location to its points-to target) requires *existential quantifier elimination.*
  - Similar to image computation and computation of strongest post-conditions.

Implementation

- The combination of symbolic heap abstraction and fluid updates forms the core of the *Compass* program verification system for C programs.
  - Compass is path- and context-sensitive
    - Summary-based
  - Used for checking *memory safety properties* (buffer overruns, null errors, uninitialized reads, casting errors, …) as well as arbitrary *user-provided assertions*.
  - Has successfully been scaled to real programs in the range of a few 10,000 lines of code.

Case Study

- We first evaluate this technique on a set of challenging array benchmarks.

Program	Time	Memory	#Sat queries	Solve time
init	0.01s	< 1 MB	172	0s
init_nonconst	0.02s	< 1 MB	184	0.01s
init_partial	0.01s	< 1MB	166	0.01s
init_partial_buggy	0.02s	< 1 MB	168	0s
init_even	0.04s	< 1 MB	146	0.04s
init_even_buggy	0.04s	< 1 MB	166	0.03s
2D_array_init	0.04s	< 1 MB	311	0.04s
copy	0.01s	< 1 MB	209	0.01s
copy_partial	0.01s	< 1 MB	220	0.01s
copy_odd	0.04s	< 1 MB	243	0.02s
copy_odd_buggy	0.05s	< 1 MB	246	0.05s
reverse	0.03s	< 1 MB	273	0.01s
reverse_buggy	0.04s	< 1 MB	281	0.02s
swap	0.12s	2 MB	590	0.11s
swap_buggy	0.11s	2 MB	557	0.06s
double_swap	0.16s	2 MB	601	0.1s
strcpy	0.07s	< 1 MB	355	0.04s
strlen	0.02s	< 1 MB	165	0.01s
strlen_buggy	0.01s	< 1 MB	89	0.01s
memcpy	0.04s	< 1 MB	225	0.04s
find	0.02s	< 1 MB	119	0.02s
find_first_nonnull	0.02s	< 1 MB	183	0.02s
append	0.02s	< 1 MB	183	0.01s
merge_interleave	0.09s	< 1 MB	296	0.07s
merge_interleave _buggy	0.11s	< 1 MB	305	0.09s
alloc_fixed_size	0.02s	< 1 MB	176	0.02s
alloc_fixed_size_buggy	0.02s	< 1 MB	172	0.02s
alloc_nonfixed_size	0.03s	< 1 MB	214	0.02

Also very memory efficient.

Very fast despite being very precise.

Compass

Enter Code | Callgraph | Cfg | Summary | Statistics | Step through Unit | Regressions

Project used: current Name: Case Study/Swap

Files | Units | Callg

Unit	Time
check_swap	
check_swap@3	
check_swap@7	
swap	
swap@3	

test.c

New File

```
void swap(int* a, int* b, int size)
{
	int i;
	for(i=0; i<size; i++) {
		int t = a[i];
		a[i] = b[i];
		b[i] = t;
	}
}

void check_swap(int size, int* a, int* b)
{
	int i;
	int* a_copy = malloc(sizeof(int)*size);
	int* b_copy = malloc(sizeof(int)*size);
	for(i=0; i<size; i++) {
		a_copy[i] = a[i];
		b_copy[i] = b[i];
	}
	swap(a, b, size);
	for(i=0; i<size; i++) {
		static_assert(a[i] == b_copy[i]);
		static_assert(b[i] == a_copy[i]);
	}

	free(a_copy);
	free(b_copy);
}
```

Status

Num CPUs 1

Walltime 0

CPU time 0

0%

5/ 0

Check Buffers

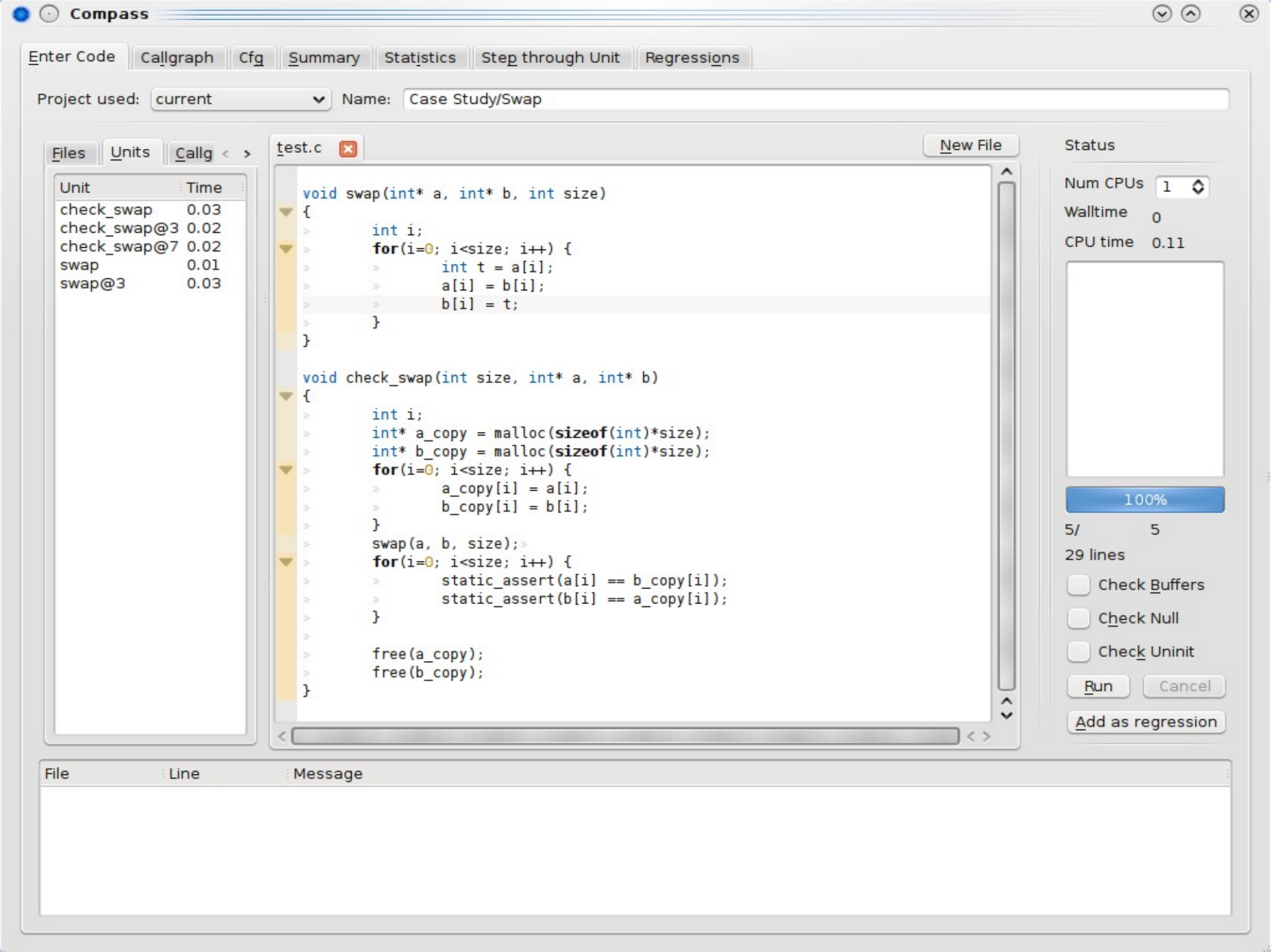
Check Null

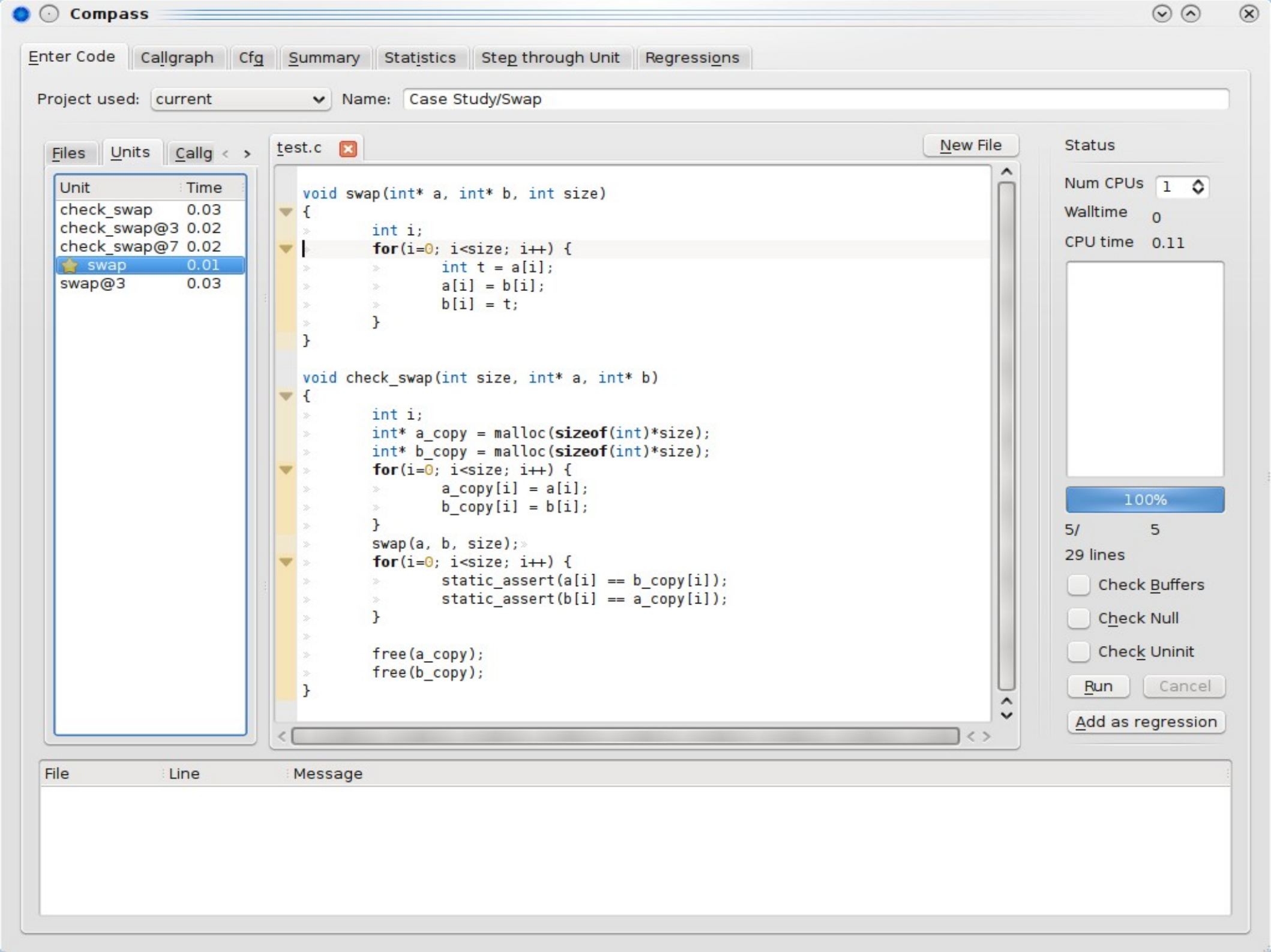
Check Uninit

Run | Cancel

Add as regression

File	Line	Message

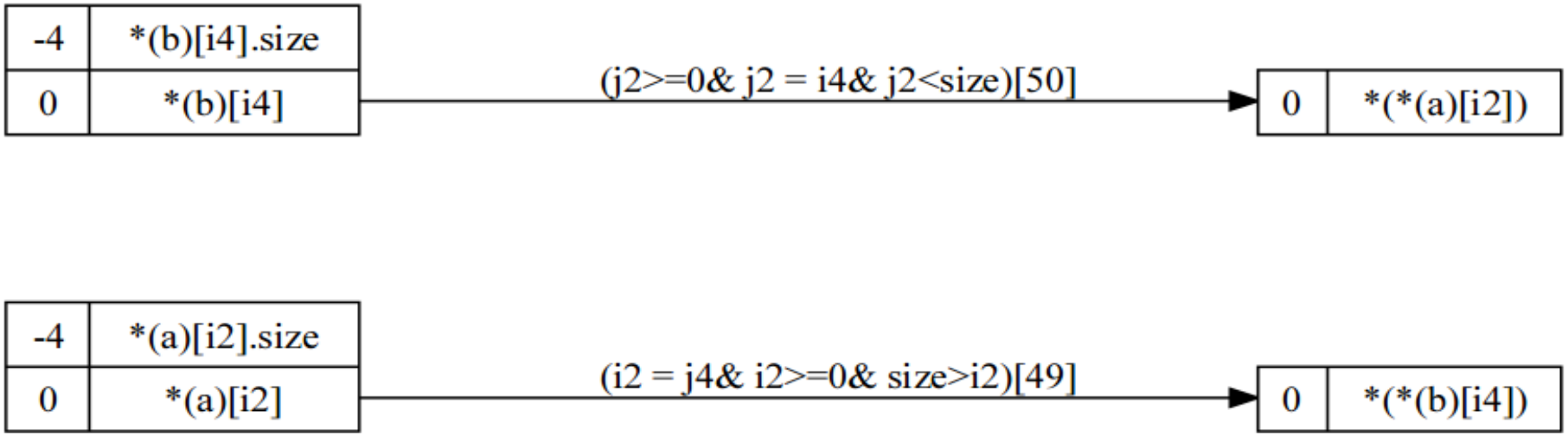

Compass
Enter Code
Callgraph
Cfg
Summary
Statistics
Step through Unit
Regressions
Project used:
current
Name:
Case Study/Swap
Files
Units
Callg
Unit
Time
check_swap 0.03
check_swap@3 0.02
check_swap@7 0.02
swap 0.01
swap@3 0.03
test.c
New File
void swap(int* a, int* b, int size)
{
int i;
for(i=0; i<size; i++) {
int t = a[i];
a[i] = b[i];
b[i] = t;
}
}
void check_swap(int size, int* a, int* b)
{
int i;
int* a_copy = malloc(sizeof(int)*size);
int* b_copy = malloc(sizeof(int)*size);
for(i=0; i<size; i++) {
a_copy[i] = a[i];
b_copy[i] = b[i];
}
swap(a, b, size);
for(i=0; i<size; i++) {
static_assert(a[i] == b_copy[i]);
static_assert(b[i] == a_copy[i]);
}
free(a_copy);
free(b_copy);
}
Status
Num CPUs
1
Walltime
0
CPU time
0.11
100%
5/
5
29 lines
Check Buffers
Check Null
Check Uninit
Run
Cancel
Add as regression
File
Line
Message


Compass
Enter Code
Callgraph
Cfg
Summary
Statistics
Step through Unit
Regressions
Project used:
current
Name:
Case Study/Swap
Files
Units
Callg
Unit
Time
check_swap 0.03
check_swap@3 0.02
check_swap@7 0.02
swap 0.01
swap@3 0.03
test.c
New File
void swap(int* a, int* b, int size)
{
int i;
for(i=0; i<size; i++) {
int t = a[i];
a[i] = b[i];
b[i] = t;
}
}
void check_swap(int size, int* a, int* b)
{
int i;
int* a_copy = malloc(sizeof(int)*size);
int* b_copy = malloc(sizeof(int)*size);
for(i=0; i<size; i++) {
a_copy[i] = a[i];
b_copy[i] = b[i];
}
swap(a, b, size);
for(i=0; i<size; i++) {
static_assert(a[i] == b_copy[i]);
static_assert(b[i] == a_copy[i]);
}
free(a_copy);
free(b_copy);
}
Status
Num CPUs
1
Walltime
0
CPU time
0.11
100%
5/
5
29 lines
Check Buffers
Check Null
Check Uninit
Run
Cancel
Add as regression
File
Line
Message

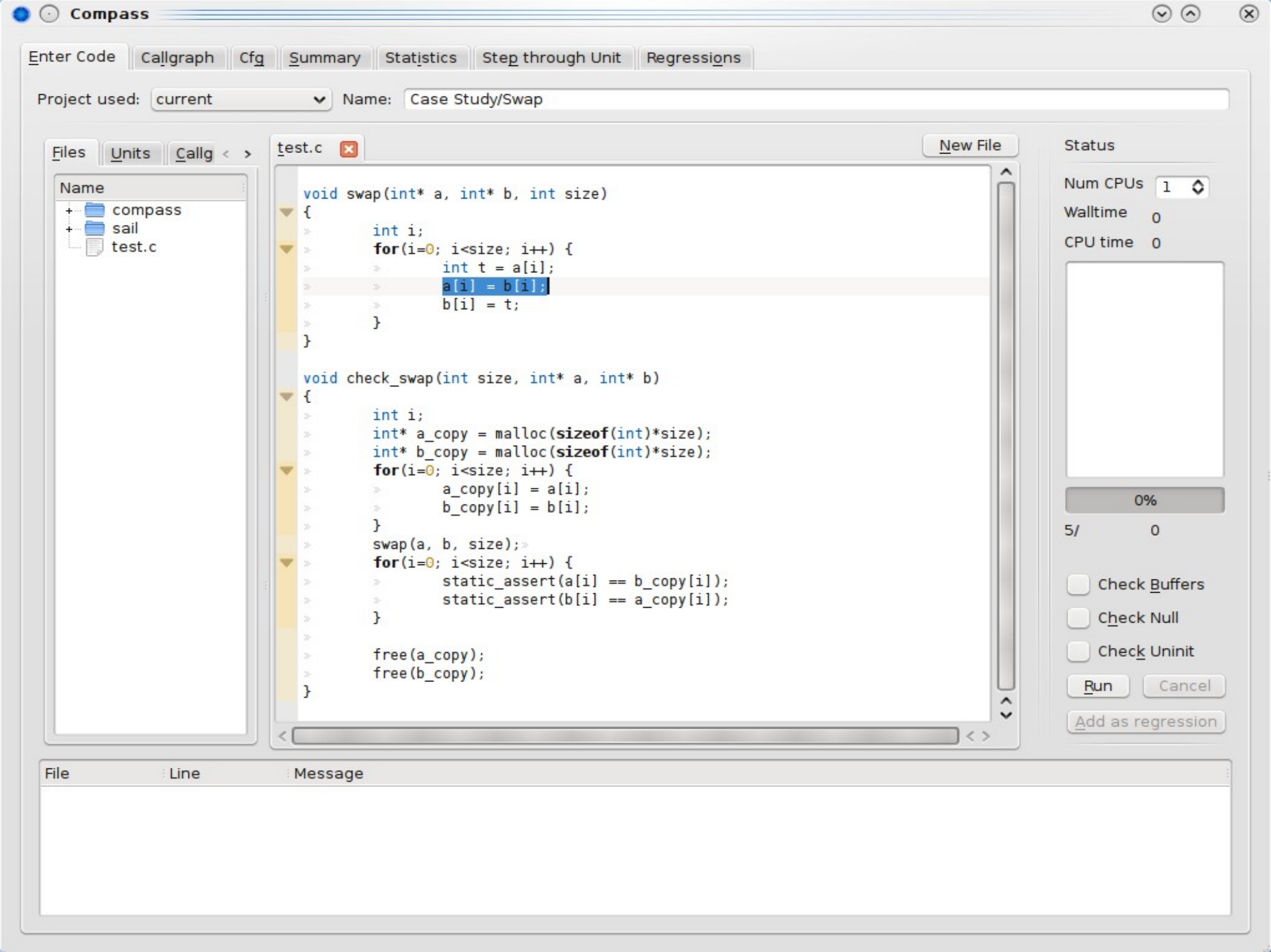
Compass

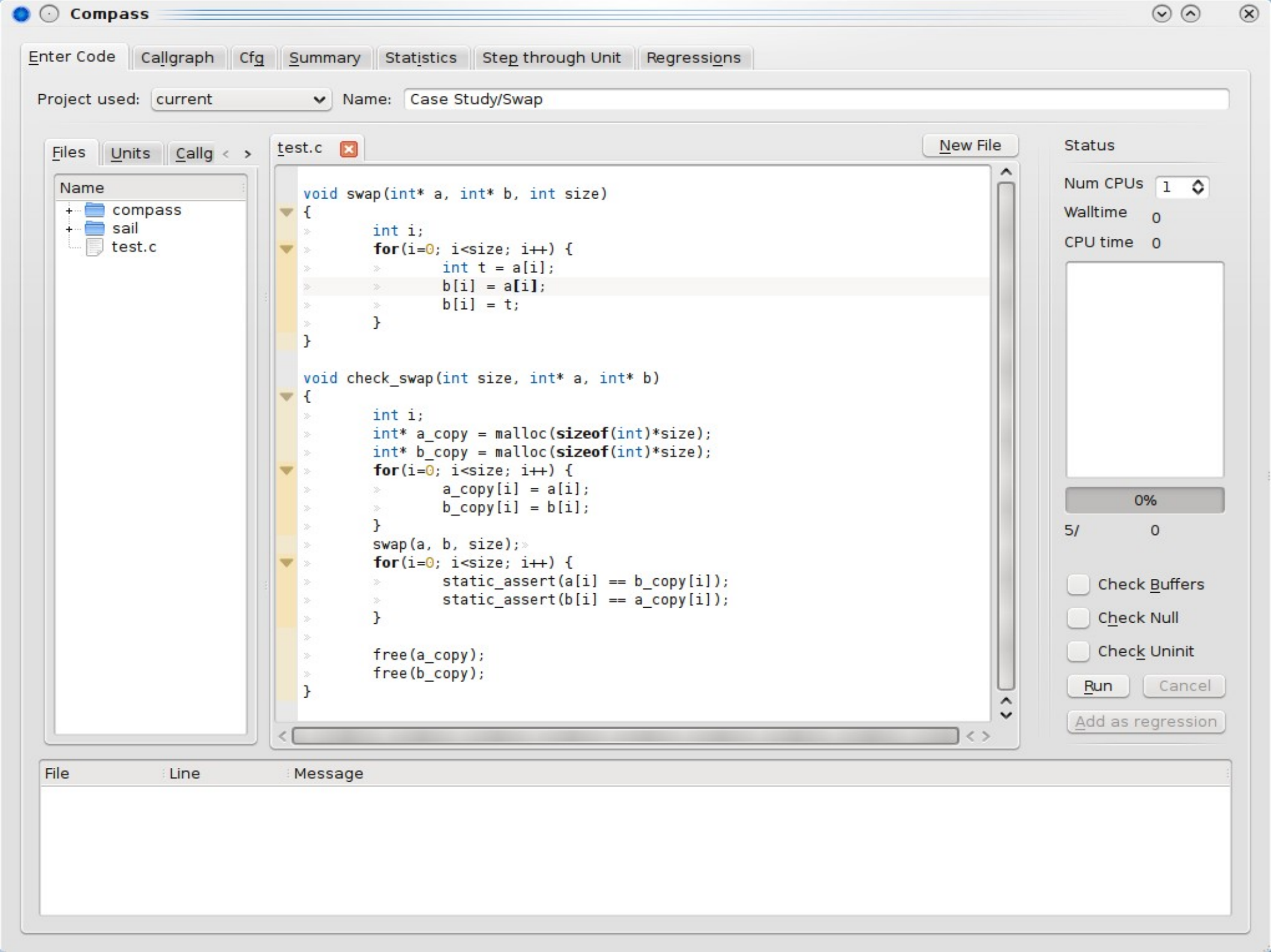
Enter Code | Callgraph | Cfg | Summary | Statistics | Step through Unit | Regressions

Summary Unit: /home/tdillig/.compass/current/test.c/swap<[nonret] (int*, int*, int)>



Return Condition: true
Loop Counters:
Error trace summary:


Compass
Enter Code
Callgraph
Cfg
Summary
Statistics
Step through Unit
Regressions
Project used:
current
Name:
Case Study/Swap
Files
Units
Callg
Name
compass
sail
test.c
test.c
New File
void swap(int* a, int* b, int size)
{
int i;
for(i=0; i<size; i++) {
int t = a[i];
a[i] = b[i];
b[i] = t;
}
}
void check_swap(int size, int* a, int* b)
{
int i;
int* a_copy = malloc(sizeof(int)*size);
int* b_copy = malloc(sizeof(int)*size);
for(i=0; i<size; i++) {
a_copy[i] = a[i];
b_copy[i] = b[i];
}
swap(a, b, size);
for(i=0; i<size; i++) {
static_assert(a[i] == b_copy[i]);
static_assert(b[i] == a_copy[i]);
}
free(a_copy);
free(b_copy);
}
Status
Num CPUs
1
Walltime
0
CPU time
0
0%
5/
0
Check Buffers
Check Null
Check Uninit
Run
Cancel
Add as regression
File
Line
Message


Compass
Enter Code
Callgraph
Cfg
Summary
Statistics
Step through Unit
Regressions
Project used:
current
Name:
Case Study/Swap
Files
Units
Callg
Name
compass
sail
test.c
test.c
New File
void swap(int* a, int* b, int size)
{
int i;
for(i=0; i<size; i++) {
int t = a[i];
b[i] = a[i];
b[i] = t;
}
}
void check_swap(int size, int* a, int* b)
{
int i;
int* a_copy = malloc(sizeof(int)*size);
int* b_copy = malloc(sizeof(int)*size);
for(i=0; i<size; i++) {
a_copy[i] = a[i];
b_copy[i] = b[i];
}
swap(a, b, size);
for(i=0; i<size; i++) {
static_assert(a[i] == b_copy[i]);
static_assert(b[i] == a_copy[i]);
}
free(a_copy);
free(b_copy);
}
Status
Num CPUs
1
Walltime
0
CPU time
0
0%
5/
0
Check Buffers
Check Null
Check Uninit
Run
Cancel
Add as regression
File
Line
Message

Compass

Enter Code | Callgraph | Cfg | Summary | Statistics | Step through Unit | Regressions

Project used: current Name: Case Study/Swap

Files | Units | Callg

Name
- compass
- sail
- test.c

test.c

New File

```
void swap(int* a, int* b, int size)
{
        int i;
        for(i=0; i<size; i++) {
                int t = a[i];
                b[i] = a[i];
                b[i] = t;
        }
}

void check_swap(int size, int* a, int* b)
{
        int i;
        int* a_copy = malloc(sizeof(int)*size);
        int* b_copy = malloc(sizeof(int)*size);
        for(i=0; i<size; i++) {
                a_copy[i] = a[i];
                b_copy[i] = b[i];
        }
        swap(a, b, size);
        for(i=0; i<size; i++) {
                static_assert(a[i] == b_copy[i]);
                static_assert(b[i] == a_copy[i]);
        }

        free(a_copy);
        free(b_copy);
}
```

Status

Num CPUs 1

Walltime 0

CPU time 0.1

100%

5/ 5

29 lines

Check Buffers

Check Null

Check Uninit

Run | Cancel

Add as regression

File	Line	Message
test.c	23	Static assert failed

Unix Coreutils

- A collection of widely used command-line utilities.
- Heavily use arrays, pointers, and string buffers.
- Precise heap analysis necessary for successful verification.

Program	Lines	Total Time	Memory	#Sat queries	Solve Time
hostname	304	0.13s	5 MB	1533	0.12s
chroot	371	0.13s	3 MB	1821	0.10s
rmdir	483	1.05s	12 MB	3461	1.02s
su	1047	1.86s	32 MB	6088	1.69s
mv	1151	0.70s	21 MB	7427	0.68s
Total	3356	3.87s	73 MB	20330	3.61

Compass verified absence of buffer overruns and null dereferences with *no false positives* and *no annotations*.

Thank You

- Gopan, D., Reps, T., Sagiv, M.: *A framework for numeric analysis of array operations*. In: POPL, NY, USA, ACM (2005) 338–350
- Deutsch, A.: *Interprocedural may-alias analysis for pointers: Beyond k-limiting*. In: PLDI, ACM NY, USA (1994) 230–241
- Reps, T.W., Sagiv, S., Wilhelm, R.: *Static program analysis via 3-valued logic*. In:CAV. Volume 3114 of Lecture Notes in Comp. Sc., Springer (2004) 15–30
- Chase, D.R., Wegman, M., Zadeck, F.K.: *Analysis of pointers and structures*. In:PLDI, NY, USA, ACM (1990) 296–310
- Halbwachs, N., Peron, M.: *Discovering properties about arrays in simple programs*. In: PLDI, NY, USA, ACM (2008) 339–34
- Jhala, R., Mcmillan, K.L.: *Array abstractions from proofs*. In: CAV. (2007)
- Schmidt, D.A.: *A calculus of logical relations for over- and underapproximating static analyses.* Sci. Comput. Program. 64(1) (2007) 29–53