Semantic Program Alignment for Equivalence Checking

Berkley Churchill, Oded Padon, Rahul Sharma, Alex Aiken

PLDI, 2019

$4^{th} July, 2022$:

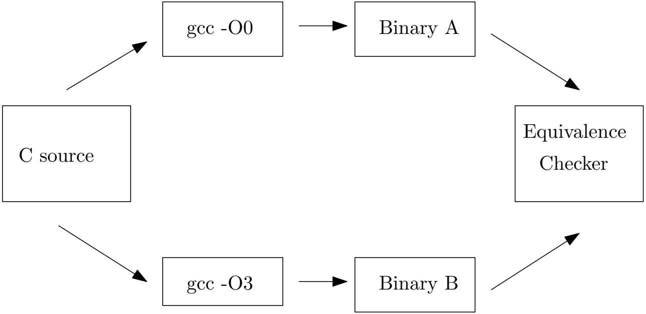
glibc strlen example

```
size_t strlen(char * s){

    char * p;
    for(p = s; *p; ++p);
    return (p - s);
}
```

```
size_t strlen(char * str){
    char *ptr;
    ulong * longword_ptr;
    ulong longword, himagic, lomagic;

    for(ptr = str; ((ulong)ptr&7)! = 0; ++ptr)
      if(*ptr == \0)
        return ptr - str;

    longword_ptr = (ulong*)ptr;
    himagic = 0x8080808080808080L;
    lomagic = 0x0101010101010101L;

   for(;;)
   {
     longword = *longword_ptr++;
     if((longword - lomagic) & ~longword & himagic)
     {  char * cp = (char*)(longword_ptr - 1);

        if(cp[0] == 0) return (cp - str);
        if(cp[1] == 0) return (cp - str + 1);
        if(cp[2] == 0) return (cp - str + 2);
        if(cp[3] == 0) return (cp - str + 3);
        if(cp[4] == 0) return (cp - str + 4);
        if(cp[5] == 0) return (cp - str + 5);
        if(cp[6] == 0) return (cp - str + 6);
        if(cp[7] == 0) return (cp - str + 7);
     }
   }
}
```

Equivalence Checking

Checking optimization correctness



Equivalence of two programs

Two programs are equivalent if running on the same input

- Both terminate on the same output state OR
- Both fail (either loop forever or encounter hardware exception)

Past techniques - Summarizing loops

$f(x)\{$

$\quad while(*)\{A;\}$

$\quad return\ a;$

$\}$

$g(x)\{$

$\quad while(*)\{B;\}$

$\quad return\ b;$

$\}$

$ProductProgram(x)\{$

$\quad while(*)\{A;\}$

$\quad while(*)\{B;\}$

$\quad assert(a == b)$

$\}$

Past techniques - Syntactic composition

```
f(x){
    while(*){A;}
    return a;
}

g(x){
    while(*){B;}
    return b;
}
```

```
ProductProgram(x){
    while(*){
        assert(Inv);
        A;
        B;
    }
    assert(a == b);
}
```

Limitations for syntactic composition

- Different number of loop execution — failure
- No 1-1 correspondence
- Syntactic choices can make problems harder for SMT solvers

The proposed Method

- A semantic-driven blackbox technique for equivalence checking is proposed
- Given two functions, a trace alignment is found over a set of concrete executions of both the programs
- A product program is constructed to check equivalence and invariants are learned
- Equivalence is established by solvers
- The authors verified correctness of vector implementation of strlen function that ships as part of GNU C library and vectorization optimization for 56 benchmarks for Test Suite for Vectorizing Compilers

Alignment

$f(x)\{$

$\quad while(*)\{A;\}$

$\quad return\ a;$

$\}$

$g(x)\{$

$\quad while(*)\{B;\}$

$\quad return\ b;$

$\}$

f
A
A
A
A
A
A
A

g
B
B
B
B
B
B
B

Alignment

$f(x)\{$

$\quad while(*)\{A;\}$

$\quad return\ a;$

$\}$

$g(x)\{$

$\quad while(*)\{B;\}$

$\quad return\ b;$

$\}$

f	g
A	B
A	B
A	B
A	B
A	B
A	B
A	B

Running example

$void\ f(int*\ array, uint\ len)\{$
$\quad for(uint\ i = 0; i < len; i++)$
$\qquad array[i] = 0xffffffff;$
$\}$

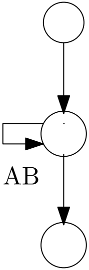
$void\ g(int*\ array, uint\ len)\{$
$\quad if(len\%2 == 1)\{$
$\qquad *array = 0xffffffff;$
$\qquad array++;$
$\qquad len--;$
$\quad \}$
$\quad while(len)\{$
$\qquad *((long*)array) = 0xffffffffffffffff;$
$\qquad array+=2;$
$\qquad len-=2;$
$\quad \}$
$\}$

- Idea : Instead of doing a syntactic composition, find a semantic way to "align" concrete execution traces. Use concrete alignment to align traces.

Running example

- Idea : Find a semantic way to "align" concrete execution traces. Use concrete alignment to align traces.
  - An alignment predicate that helps us find corresponding paths and build a product program
  - Given product program, the author leverages existing techniques to complete proof

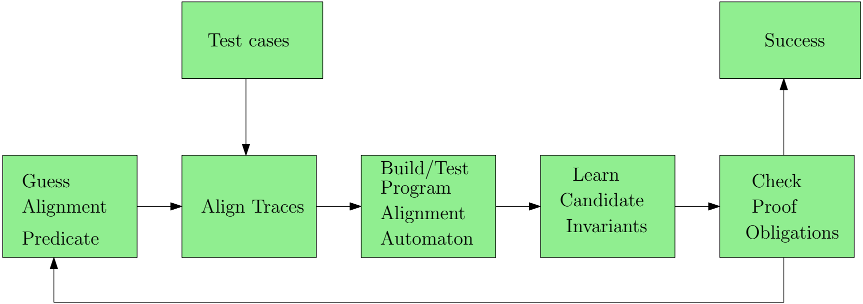
```
ProductProgram(x) {
    while(*) {
        assert(Inv);
        A;
        B;
    }
}
```



Other prior work

- (Barthe et al., PPoPP '13)
  - Cannot handle loop peeling
  - Cannot handle all forms of vectorization
- (Dahiya and Bansal, APLAS '17)
  - Searching for predicate is inefficient
  - Cannot handle some loop vectorization/unrolling benchmarks

Algorithm

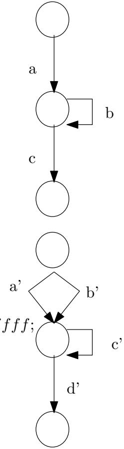


Test cases
Success
Guess Alignment Predicate
Align Traces
Build/Test Program Alignment Automaton
Learn Candidate Invariants
Check Proof Obligations

Alignment Predicate

```
void f(int * array, uint len){
    for(uint i = 0; i < len; i++)
        array[i] = 0xffffffff;
}
```

```
void g(int * array, uint len){
    if(len%2 == 1){
        *array = 0xffffffff;
        array++;
        len--;
    }
    while(len){
        *((long*)array) = 0xffffffffffffffff;
        array+ = 2;
        len- = 2;
    }
}
```



Δ	i	len	$array$
–	–	5	100000
a	0	5	100000
b	1	5	100000
b	2	5	100000
b	3	5	100000
b	4	5	100000
b	4	5	100000
c	5	5	100000

Δ'	len'	array'
–	5	100000
a'	4	100004
c'	2	100012
c'	0	100020
d'	0	100020

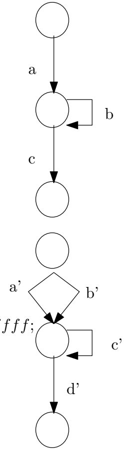
Alignment Predicate

```
void f(int * array, uint len){
    for(uint i = 0; i < len; i++)
        array[i]^ = 0xffffffff;
}
```

$@array + 4i$

```
void g(int * array, uint len){
  if(len%2 == 1){
    *array^ = 0xffffffff;
    array++;
    len--;
  }
  while(len){
    *((long*)array)^ = 0xffffffffffffffff;
    array+ = 2;
    len- = 2;
  }
```

$@array'$



f

Δ	i	len	$array$
–	–	5	100000
a	0	5	100000
b	1	5	100000
b	2	5	100000
b	3	5	100000
b	4	5	100000
b	4	5	100000
c	5	5	100000

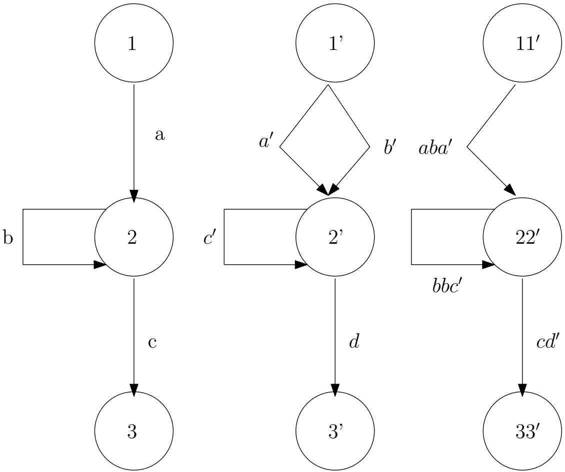
g

Δ'	len'	array'
–	5	100000
a'	4	100004
c'	2	100012
c'	0	100020
d'	0	100020

Alignment Predicate

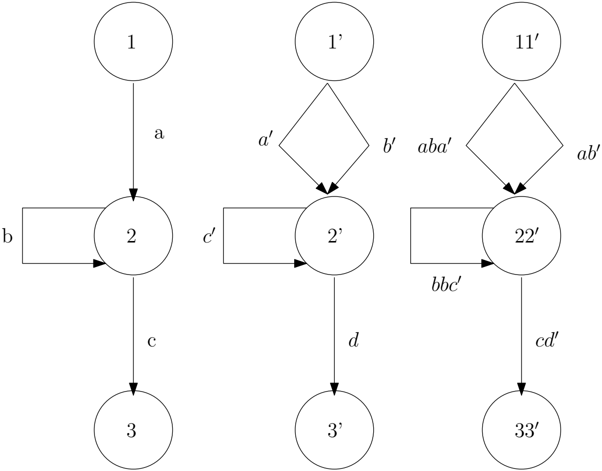
- The authors pick alignment predicates by guess and check
  - $c_1 v_1 - c_2 v_2 = k$
  - k is an integer mined from execution data
  - $c_1, c_2 \in \{1, 2, 4, 8, 16\}$
  - v_1, v_2 are registers or stack-allocated values

Building the PAA



- $(ab, a'), (bb, c'), (c, d')$

Building the PAA

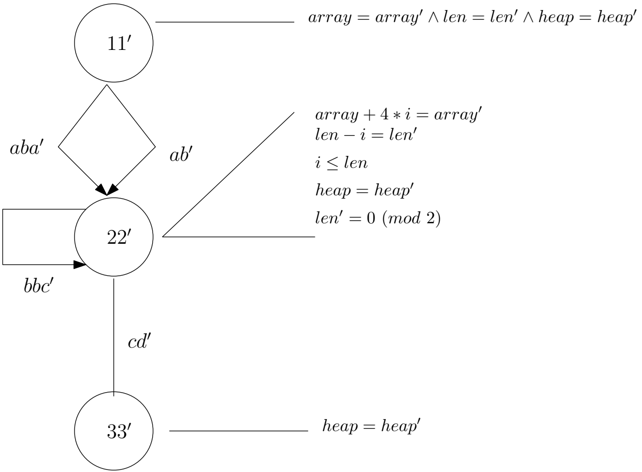


- $(a, b'), (bb, c'), (c, d')$

Learning invariants

- A data-driven approach is taken
- The test cases are used to guess a conjunction of predicates for each node
- Later the conjuncts that cannot be proven are discarded
  - This is done by a fixed-point method

Candidate invariants



Proof obligations

1. Invariants hold
2. There are no missing edge/transitions in PAA (PAA is sound over-approximation of both the programs)
3. The invariants at the final state implies equality of outputs (memory + registers)

Soundness

- **Theorem** : If the proof obligations hold for a PAA, then the two procedures are equivalent
  - By induction on the length of computations of f and g

Evaluation

- "Test Suite for Vectorizing Compilers"
- Ran on 28 C functions
  - gcc -O3 with gcc -O1
  - clang -O3 with gcc -O1
- Total of 56 benchmarks
- Used Z3 and CVC4 and 30-minute timeout per query

Evaluations

These 56 benchmarks are from :

- Vectorization (50 benchmarks)
- Loop unrolling (47 benchmarks)
- Loop peeling (9 benchmarks)
- Floating point (2 benchmarks)
- Doubly nested (2 benchmarks)
- Different loop traversal (e.g., strides, forward, backward)
- Other optimizations (e.g., transformed branch conditions)

They have verified 55/56 benchmarks

Limitations

- The method cannot reason about transformations that reorder an unbounded number of memory writes
  - Loop splitting
  - Loop fusion
  - Loop interchange
  - Loop tiling

Future work

- **Alingment predicates**
  - Can be generalized where three or more registers are involved
  - Different alignment predicates for different loops
- **Loop invariant**
  - Learning and proving different invariants over unbounded heap locations

Conclusion

- **Key idea** : use a weak invariant, the alignment predicate, to bootstrap the construction of the product program
- Use product program to learn the remaining invariants
- The method handles real optimizations performed by the modern compilers

Thank you !