An SMT Encoding of LLVM's Memory Model for Bounded Translation Validation

Seoul National Univ.

Juneyoung Lee
Dongjoo Kim
Chung-Kil Hur

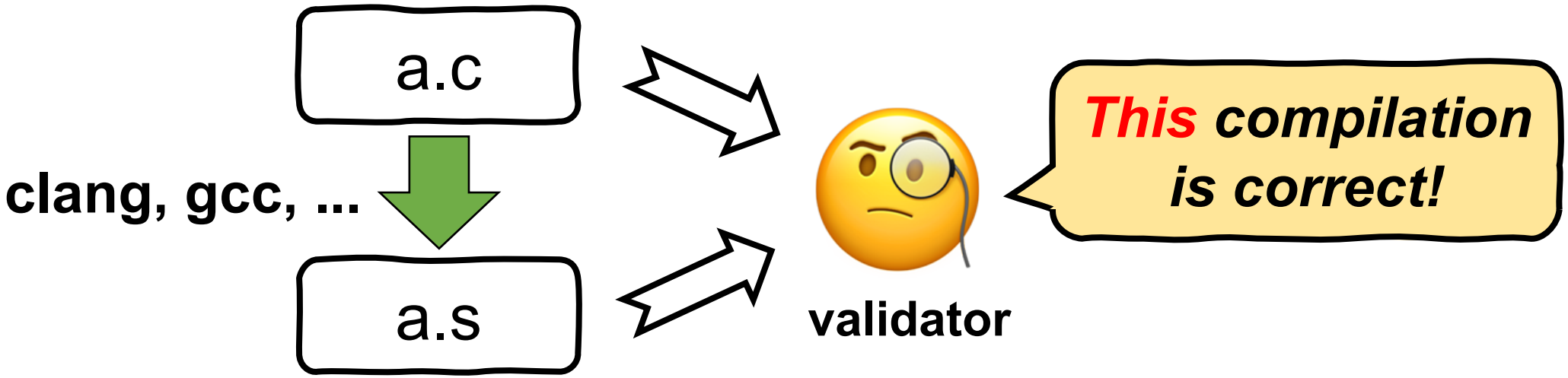
Microsoft Research

Microsoft Research

Nuno P. Lopes

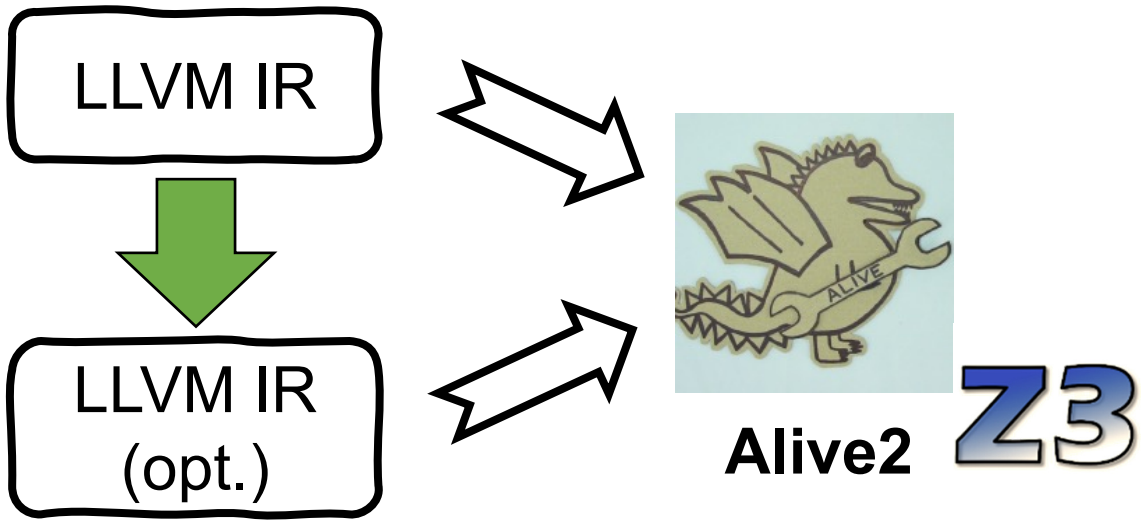
Verifying Compiler using *Translation Validation*

- Compiler correctness is the foundation of software's correctness
- Translation validation (TV): verify compilation of a *specific* program



Alive2: SMT-based Bounded TV for LLVM

1. SMT-based: Automatically check correctness using an SMT solver
2. Bounded TV (BTV): Bounded verification to keep run time reasonable



- Used by >100 LLVM patches
- Found & reported 50 bugs

https://alive2.llvm.org

An SMT encoding of LLVM's Memory Model for BTV!

- This paper: Alive2's SMT encoding for LLVM's memory model
  - PLDI'21[1]: LLVM's special values (undef, poison), function calls, loops, etc
- Contributions:
  1. An efficient SMT encoding of memory for BTV
  2. Discovery of ambiguous parts in LLVM IR's semantics
  3. Finding & fixing bugs in LLVM

[1] *Alive2: Bounded Translation Validation for LLVM*, PLDI'21

A Simple Optimization Example

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

<f_{src}>

```
1′  int f(int *p) {
2′    // q removed
3′
4′    int *r = g(p+1);
5′    *r = 37;
6′    return 42;
7′  }
```

<f_{tgt}>

A Simple Optimization Example

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

<f_{src}>

```
1′  int f(int *p) {
2′    // q removed
3′
4′    int *r = g(p+1);
5′    *r = 37;
6′    return 42;
7′  }
```

<f_{tgt}>

Why is this compiler optimization correct?
What if r = q?

Why Is This Optimization Correct?

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

<f_src>

```
1′  int f(int *p) {
2′    // q removed
3′
4′    int *r = g(p+1);
5′    *r = 37;
6′    return 42;
7′  }
```

<f_tgt>

This optimization is correct because...

Why Is This Optimization Correct?

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

0x100

0x100

<f_{src}>

```
1′  int f(int *p) {
2′    // q removed
3′
4′    int *r = g(p+1);
5′    *r = 37;
6′    return 42;
7′  }
```

<f_{tgt}>

This optimization is correct because...

Why Is This Optimization Correct?

```
1  int f(int *p) {
2    int *q = malloc
3    *q = 42;
4    int *r = g(p+1);
5    *r = 37;
6    return *q;
```

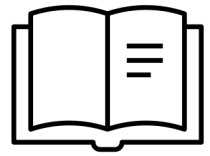
0x100

0x100

Cannot update *q as well!

Cannot update *q!

According to LLVM Language Reference, `0x100` can be accessed via `q` only!



<f_tgt>

This optimization is correct because...

Why Is This Optimization Correct?

```
1   int f(int *p) {
2     int *q = malloc
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
```

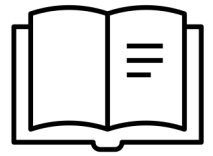
0x100

0x100

Cannot update *q as well!

Cannot update *q!

According to LLVM Language Reference, `0x100` can be accessed via `q` only!



This optimization is correct because...

1. f_{src} and f_{tgt} both returns 42.
2. f_{src} and f_{tgt} both updates *r to 37

(It has more details, but omitted for brevity)

Explaining Opt. Using Formal Memory Model

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

```
1′  int f(int *p) {
2′    // q removed
3′
4′    int *r = g(p+1);
5′    *r = 37;
6′    return 42;
7′  }
```

LLVM's formal memory model (OOSPLA'18)

Explaining Opt. Using Formal Memory Model

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

```
1′  int f(int *p) {
2′    // q removed
3′
4′    int *r = g(p+1);
5′    *r = 37;
6′    return 42;
7′  }
```

LLVM's formal memory model (OOSPLA'18)

1. A memory is a set of memory blocks

Explaining Opt. Using Formal Memory Model

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

```
1′  int f(int *p) {
2′    // q removed
3′
4′    int *r = g(p+1);
5′    *r = 37;
6′    return 42;
7′  }
```

LLVM's formal memory model (OOSPLA'18)

1. A memory is a set of memory blocks
2. An allocation creates a fresh memory block

Explaining Opt. Using Formal Memory Model

```
1  int f(int *p) {
2    int *q = malloc(4);
3    *q = 42;
4    int *r = g(p+1);
5    *r = 37;
6    return *q;
7  }
```

Creates a new memory block b_1.

```
3′
4′   int *r = g(p+1);
5′   *r = 37;
6′   return 42;
7′ }
```

LLVM's formal memory model (OOSPLA'18)

1. A memory is a set of memory blocks
2. An allocation creates a fresh memory block

Explaining Opt. Using Formal Memory Model

```
1  int f(int *p) {
2    int *q = malloc(4);
3    *q = 42;
4    int *r = g(p+1);
5    *r = 37;
6    return *q;
7  }
```

Creates a new memory block b_1.

```
3′
4′   int *r = g(p+1);
5′   *r = 37;
6′   return 42;
7′ }
```

LLVM's formal memory model (OOSPLA'18)

1. A memory is a set of memory blocks
2. An allocation creates a fresh memory block
3. A pointer tracks which memory block it can access (*provenance*)

Explaining Opt. Using Formal Memory Model

```
1  (int *p) {
2      int *q = malloc(4);
3      *q = 42;
4      int *r = g(p+1);
5      *r = 37;
6      return *q;
7  }
```

$(b_1, 0x100)$

(b_r, off_r)

Creates a new memory block b_1.

```
3′
4′     int *r = g(p+1);
5′     *r = 37;
6′     return 42;
7′ }
```

LLVM's formal memory model (OOSPLA'18)

1. A memory is a set of memory blocks
2. An allocation creates a fresh memory block
3. A pointer tracks which memory block it can access (*provenance*)

Explaining Opt. Using Formal Memory Model

```
1  f(int *p) {
2      int *q = malloc(4);
3      *q = 42;
4      int *r = g(p+1);
5      *r = 37;
6      return *q;
7  }
```

(b_1, 0x100)

(b_r, off_r)

Creates a new memory block b_1.

```
3′
4′     int *r = g(p+1);
5′     *r = 37;
6′     return 42;
7′ }
```

LLVM's formal memory model (OOSPLA'18)

1. A memory is a set of memory blocks
2. An allocation creates a fresh memory block
3. A pointer tracks which memory block it can access (*provenance*)
4. Dereferencing a pointer is successful if the block is alive & the offset is within the block's bounds

Explaining Opt. Using Formal Memory Model

```
1  f(int *p) {
2      int *q = malloc(4);
3      *q = 42;
4      int *r = g(p+1);
5      *r = 37;
6      return *q;
7  }
```

(b_1, 0x100)

(b_r, off_r)

Creates a new memory block b_1.

Since b_1 != b_r, *r fails if off_r is 0x100!

LLVM's formal memory model (OOSPLA'18)

1. A memory is a set of memory blocks
2. An allocation creates a fresh memory block
3. A pointer tracks which memory block it can access (*provenance*)
4. Dereferencing a pointer is successful if the block is alive & the offset is within the block's bounds

Encoding The Memory Model In SMT

Initial memory: m_0

(b_p, off_p)

$(b_1, 0x100)$

(b_r, off_r)

```
1  int f(int *p) {
2    int *q = malloc(4);
3    *q = 42;
4    int *r = g(p+1);
5    *r = 37;
6    return *q;
7  }
```

<f_{src}>

(b_p, off_p)

```
1′ int f(int *p) {
2′   // q removed
3′
4′   int *r = g(p+1);
5′   *r = 37;
6′   return 42;
7′ }
```

<f_{tgt}>

(the intermediate & final states are omitted)

How to encode m_0, b_p, b_1, b_r, off_r, ... in SMT? 🤔

Efficient SMT Encoding of LLVM's Memory Model

We introduce our two important techniques

- Technique I: Bounding # of memory blocks
- Technique II: Using partial-order reduction to shrink # of aliasing blocks

Technique I: Bounding # of Memory Blocks

- Can we bound # of blocks that is necessary to verify an optimization?
  - Determines # of byte arrays and the bitwidth of block ids
  - In BTV, loops are assumed to have bounded iterations

Technique I: Bounding # of Memory Blocks

- Can we bound # of blocks that is necessary to verify an optimization?
  - Determines # of byte arrays and the bitwidth of block ids
  - In BTV, loops are assumed to have bounded iterations
- We calculate the bound via two steps:
  A. # of blocks that is enough to encode the behavior of each function ($\mathbf{f_{src}}$ and $\mathbf{f_{tgt}}$)
  B. # of blocks that is enough to encode the correctness of $\mathbf{f_{src}} \rightarrow \mathbf{f_{tgt}}$.

A. Bounding # of Memory Blocks of f_{src}

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

<f_{src}>

N_l^{src}: # of blocks allocated inside f

N_{nl}^{src}: # of blocks allocated outside f

$$(N_l^{src}, N_{nl}^{src}) = (0, 0)$$

A. Bounding # of Memory Blocks of f_{src}

Using p,
only **one block** (p.bid) can be touched!

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

<f_{src}>

N_l^{src}: # of blocks allocated inside f

N_{nl}^{src}: # of blocks allocated outside f

$$(N_l^{src}, N_{nl}^{src}) = (0, 0)$$

A. Bounding # of Memory Blocks of f_{src}

Using p,
only **one block** (p.bid) can be touched!

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

<f_{src}>

N_l^{src}: # of blocks allocated inside f

N_{nl}^{src}: # of blocks allocated outside f

$$(N_l^{src}, N_{nl}^{src}) = (0, 1)$$

A. Bounding # of Memory Blocks of f_{src}

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

<f_{src}>

N_l^{src}: # of blocks allocated inside f

N_{nl}^{src}: # of blocks allocated outside f

$$(N_l^{src}, N_{nl}^{src}) = (0, 1)$$

A. Bounding # of Memory Blocks of f_{src}

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

<f_{src}>

N_l^{src}: # of blocks allocated inside f

N_{nl}^{src}: # of blocks allocated outside f

$$(N_l^{src}, N_{nl}^{src}) = (1, 1)$$

A. Bounding # of Memory Blocks of f_{src}

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

<f_{src}>

N_l^{src}: # of blocks allocated inside f

N_{nl}^{src}: # of blocks allocated outside f

$$(N_l^{src}, N_{nl}^{src}) = (1, 1)$$

A. Bounding # of Memory Blocks of f_{src}

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
```

N_l^{src}: # of blocks allocated inside f

N_{nl}^{src}: # of blocks allocated outside f

$$(N_l^{src}, N_{nl}^{src}) = (1, 1)$$

1. g(p+1) can return an unseen pointer
   ➔ **+1 extra block**
2. g(p+1) can access unseen blocks
   ➔ Finding *one* mismatched block is enough to create a counter-example
   ➔ **+1 extra block is enough**

A. Bounding # of Memory Blocks of f_{src}

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
```

N_l^{src}: # of blocks allocated inside f

N_{nl}^{src}: # of blocks allocated outside f

$$(N_l^{src}, N_{nl}^{src}) = (1, 3)$$

1. g(p+1) can return an unseen pointer
   ➔ **+1 extra block**
2. g(p+1) can access unseen blocks
   ➔ Finding *one* mismatched block is enough to create a counter-example
   ➔ **+1 extra block is enough**

A. Bounding # of Memory Blocks of f_{src}

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
```

N_l^{src}: # of blocks allocated inside f

N_{nl}^{src}: # of blocks allocated outside f

$$(N_l^{src}, N_{nl}^{src}) = (1, 3)$$

1. g(p+1) can return an unseen pointer
   ➔ **+1 extra block**
2. g(p+1) can access unseen blocks
   ➔ Finding *one* mismatched block is enough to create a counter-example
   ➔ **+1 extra block is enough**

Q: If there is g2(), **+1** again?

A: **g2() doesn't count!**

A. Bounding # of Memory Blocks of f_{src}

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

<f_{src}>

N_l^{src}: # of blocks allocated inside f

N_{nl}^{src}: # of blocks allocated outside f

$$(N_l^{src}, N_{nl}^{src}) = (1, 3)$$

B. Bounding # of Memory Blocks to Verify $f_{src} \rightarrow f_{tgt}$

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

$(N_l^{src}, N_{nl}^{src}) = (1, 3)$

```
1′  int f(int *p) {
2′    // q removed
3′
4′    int *r = g(p+1);
5′    *r = 37;
6′    return 42;
7′  }
```

$(N_l^{tgt}, N_{nl}^{tgt}) = (0, 3)$

Q: How to decide the total # of memory blocks?

B. Bounding # of Memory Blocks to Verify $f_{src} \rightarrow f_{tgt}$

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

```
1′  int f(int *p) {
2′    // q removed
3′
4′    int *r = g(p+1);
5′    *r = 37;
6′    return 42;
7′  }
```

1. Local blocks:

$$N_l = N_l^{src} + N_{nl}^{tgt} = 1$$

... because local blocks in f_{src} and f_{tgt} are independent

B. Bounding # of Memory Blocks to Verify $f_{src} \rightarrow f_{tgt}$

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

```
1′  int f(int *p) {
2′    // q removed
3′
4′    int *r = g(p+1);
5′    *r = 37;
6′    return 42;
7′  }
```

2. Nonlocal blocks:

$N_{nl} = ???$

B. Bounding # of Memory Blocks to Verify $f_{src} \rightarrow f_{tgt}$

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

```
1′  int f(int *p) {
2′    // q removed
3′
4′    int *r = g(p+1);
5′    *r = 37;
6′    return 42;
7′  }
```

2. Nonlocal blocks:

$$N_{nl} = N_{nl}^{src} = 3$$

... because we only need ***one*** counter-example if $f_{src} \rightarrow f_{tgt}$ is wrong!

B. Bounding # of Memory Blocks to Verify $f_{src} \rightarrow f_{tgt}$

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

```
1′  int f(int *p) {
2′    // q removed
3′
4′    int *r = g(p-1);
5′    *r = 37;
6′    return 42;
7′  }
```

2. Nonlocal blocks:

$$N_{nl} = N_{nl}^{src} = 3$$

... because we only need ***one*** counter-example if $f_{src} \rightarrow f_{tgt}$ is wrong!

B. Bounding # of Memory Blocks to Verify $f_{src} \rightarrow f_{tgt}$

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 1;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

Well-defined

OK

Does ***not*** point to ***any*** memory block!

```
2′    ... ed
3′
4′    int *r = g(p-1);
5′    *r = 37;
6′    return 42;
7′  }
```

Undefined

Fail

2. Nonlocal blocks:

$$N_{nl} = N_{nl}^{src} = 3$$

... because we only need ***one*** counter-example if $f_{src} \rightarrow f_{tgt}$ is wrong!

Encoding Memory And Pointers Using N_l and N_{nl}

(bid, off)

Initial memory: m_0

(bid, off)

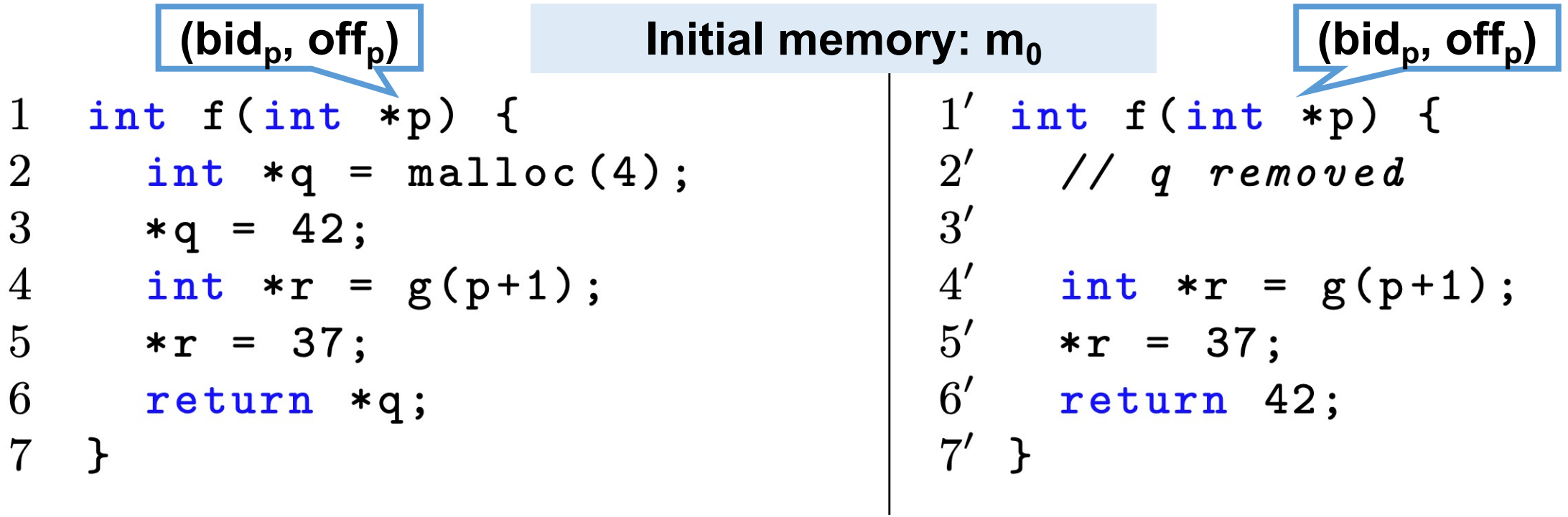
```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

```
1′  int f(int *p) {
2′    // q removed
3′
4′    int *r = g(p+1);
5′    *r = 37;
6′    return 42;
7′  }
```

- m_0: $N_l + N_{nl} = N_l^{src} + N_l^{tgt} + N_{nl}^{src}$ byte arrays
- bid: $1 + \log_2 \max(N_l^{src}, N_l^{tgt}, N_{nl}^{src})$ bit-vector

a bit for local(1)/non-local bid(0)

Technique II: Partial-Order Reduction



Technique II: Partial-Order Reduction

(000, off_p)

Initial memory: m_0

(000, off_p)

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

```
1′  int f(int *p) {
2′    // q removed
3′
4′    int *r = g(p+1);
5′    *r = 37;
6′    return 42;
7′  }
```

Technique II: Partial-Order Reduction

(000, off_p)

Initial memory: m_0

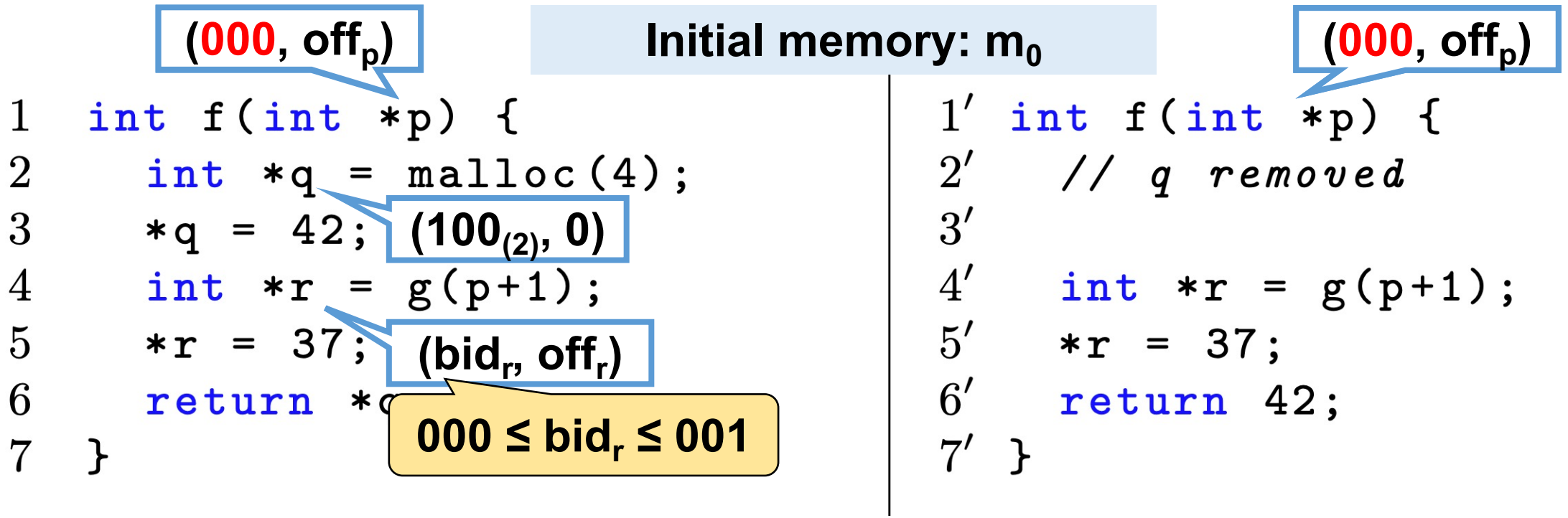
(000, off_p)

```
1   int f(int *p) {
2     int *q = malloc(4);
3     *q = 42;
4     int *r = g(p+1);
5     *r = 37;
6     return *q;
7   }
```

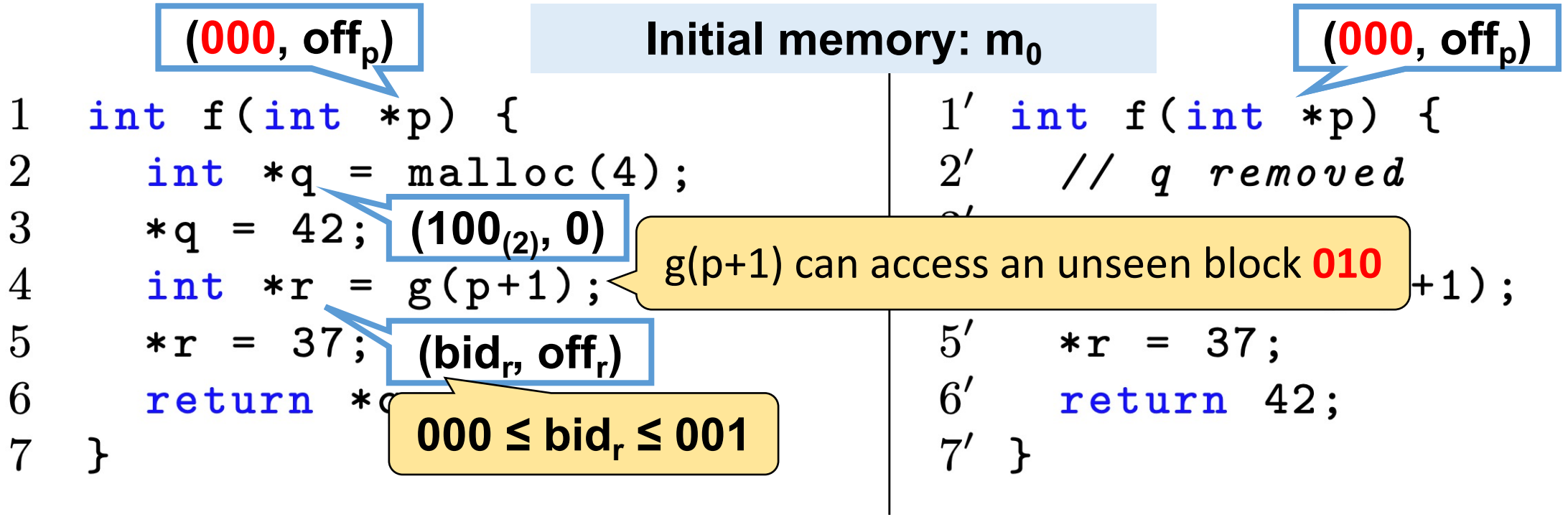
($100_{(2)}$, 0)

```
1′  int f(int *p) {
2′    // q removed
3′
4′    int *r = g(p+1);
5′    *r = 37;
6′    return 42;
7′  }
```

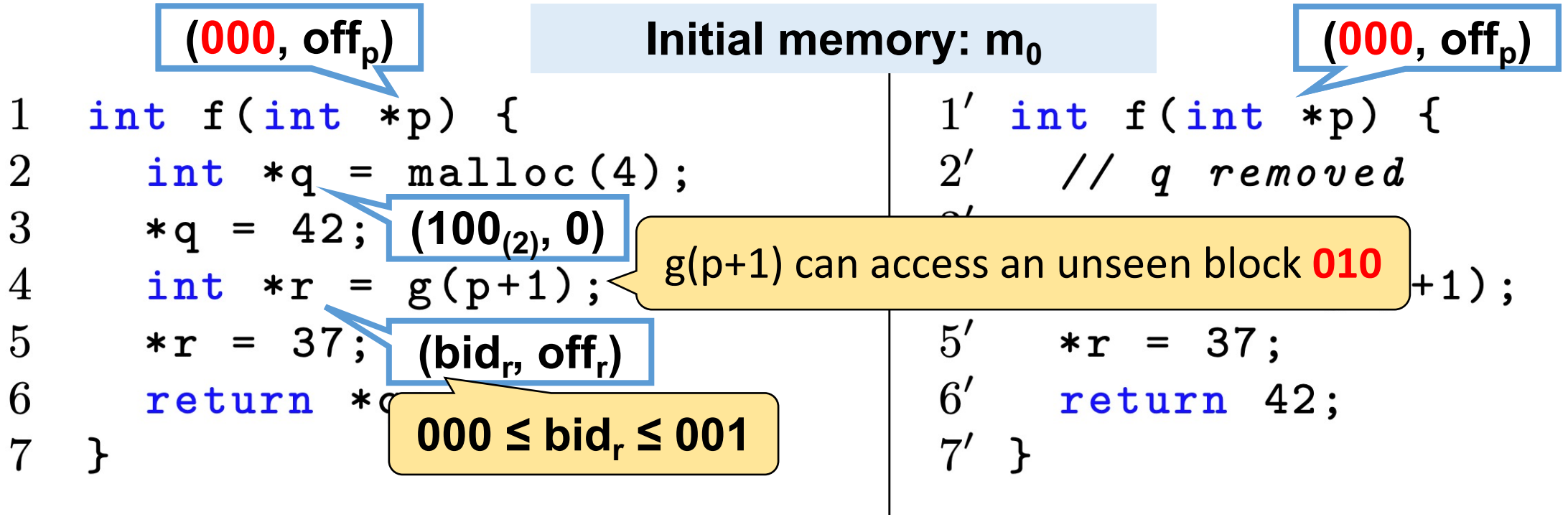
Technique II: Partial-Order Reduction



Technique II: Partial-Order Reduction



Technique II: Partial-Order Reduction



Benefit

Reduces the size of search space a solver needs to explore!

More Techniques Are Described In The Paper

Optimizations

- Specialize pointer/non-pointer bytes
- Omit disjointness of block addresses if they are never observed
- Shrink pointer offset variable’s bitwidth

More Techniques Are Described In The Paper

Optimizations

- Specialize pointer/non-pointer bytes
- Omit disjointness of block addresses if they are never observed
- Shrink pointer offset variable's bitwidth

Approximations

- Assume that local blocks are located at the upper half of memory / non-local blocks at the lower half
- Bound the # iterations of **strlen**/**memcmp**/**bcmp** to constants

Implementation & Evaluation

- Implemented our memory model in Alive2
  - Includes escaped local block support, function attributes, etc
- Run LLVM unit tests (~36,600 IR fns): 2.5 hrs on Intel Xeon 12 cores
  - Validated intraprocedural optimizations
  - Found 21 bugs in memory optimizations
  - Found that the semantics of LLVM's nonnull attribute was problematic

Implementation & Evaluation (cont.)

- Run 5 single file benchmarks with -O3: 5.1k (bzip2) ~ 141kLOC (sqlite3)
  - 71 incorrect pairs: due to mismatch between LLVM developers' informal semantics and formal semantics (OOPSLA'18)
  - ***The gap is shrinking***! sqlite3: 66 (last year) → 38

Implementation & Evaluation (cont.)

- Run 5 single file benchmarks with -O3: 5.1k (bzip2) ~ 141kLOC (sqlite3)
  - 71 incorrect pairs: due to mismatch between LLVM developers' informal semantics and formal semantics (OOPSLA'18)
  - ***The gap is shrinking***! sqlite3: 66 (last year) → 38
- Show efficiency of memory block encoding
  - 96% of the dereferenced ptrs are either local/nonlocal, but not both
  - 80% of the pointers alias with at most 3 blocks (avg. blk: 7 ~ 13)
  - Array-per-block vs. local/nonlocal: 10% increase in # verified pairs of oggenc

Conclusion

1. We devised an efficient SMT encoding of LLVM memory model for BTV
2. We implemented our encoding in Alive2 and found 21 bugs in LLVM

The paper has more topics that are not treated in this talk!

Thank you! :)