Ten Algorithms with the greatest influence on science and engg. in the 20th century

- the Metropolis algorithm for Monte Carlo
- the simplex method for linear programming
- Krylov subspace iteration methods
- the decompositional approach to matrix computations
- the Fortran optimizing compiler
- the QR algorithm for computing eigenvalues
- the quicksort algorithm for sorting
- the fast Fourier transform
- integer relation detection
- the fast multipole method

Guest editors of IEEE Computing in Science & Engineering 2000

Ten Algorithms with the greatest influence on science and engg. in the 20th century

- the Metropolis algorithm for Monte Carlo
- the simplex method for linear programming
- Krylov subspace iteration methods
- the decompositional approach to matrix computations
- the Fortran optimizing compiler
- the QR algorithm for computing eigenvalues
- the quicksort algorithm for sorting
- the fast Fourier transform
- integer relation detection
- the fast multipole method

Guest editors of IEEE Computing in Science & Engineering 2000

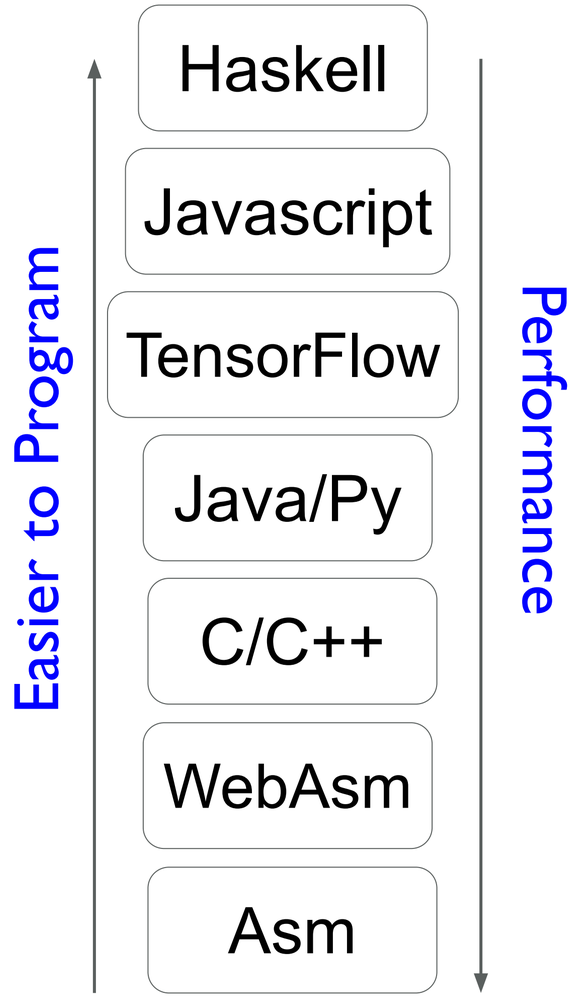
The Fortran Optimizing Compiler

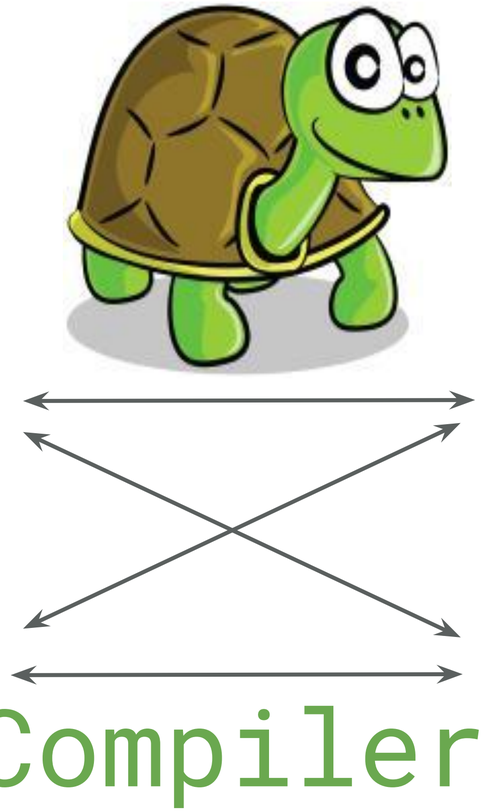
1957: John Backus leads a team at IBM in developing the **Fortran optimizing compiler**.
The creation of Fortran may rank as the single most important event in the history of computer programming: Finally, scientists

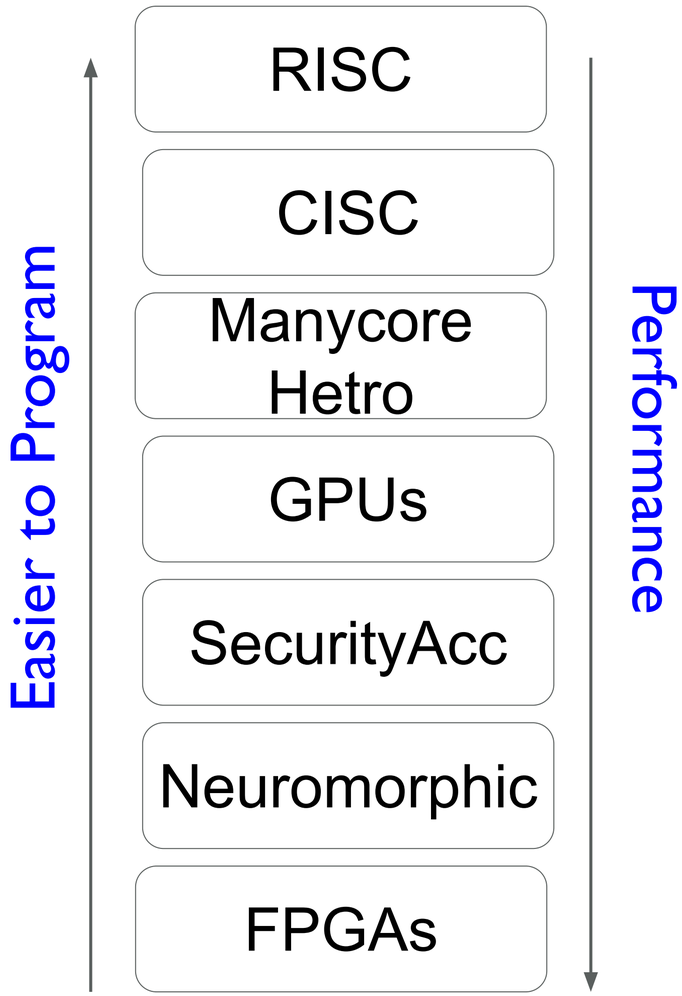
(and others) could tell the computer what they wanted it to do, without having to descend into the netherworld of machine code. Although modest by modern compiler standards—Fortran I consisted of a mere 23,500 assembly-language instructions—the early compiler was nonetheless capable of surprisingly sophisticated computations. As Backus himself recalls in a recent history of Fortran I, II, and III, published in 1998 in the *IEEE Annals of the History of Computing*, the compiler "produced code of such efficiency that its output would startle the programmers who studied it."

Guest editors of IEEE Computing in Science & Engineering 2000

End of Moore's Law will see…



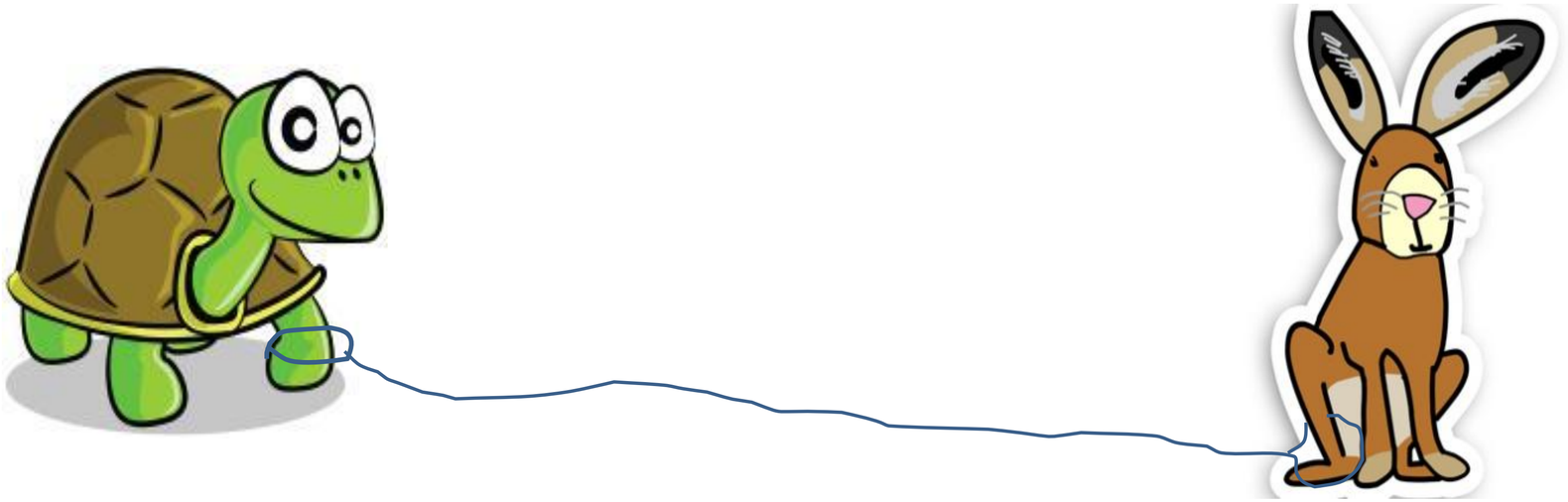




Increasing number of Languages

Increasing number of Architectures

Architects and Compilers

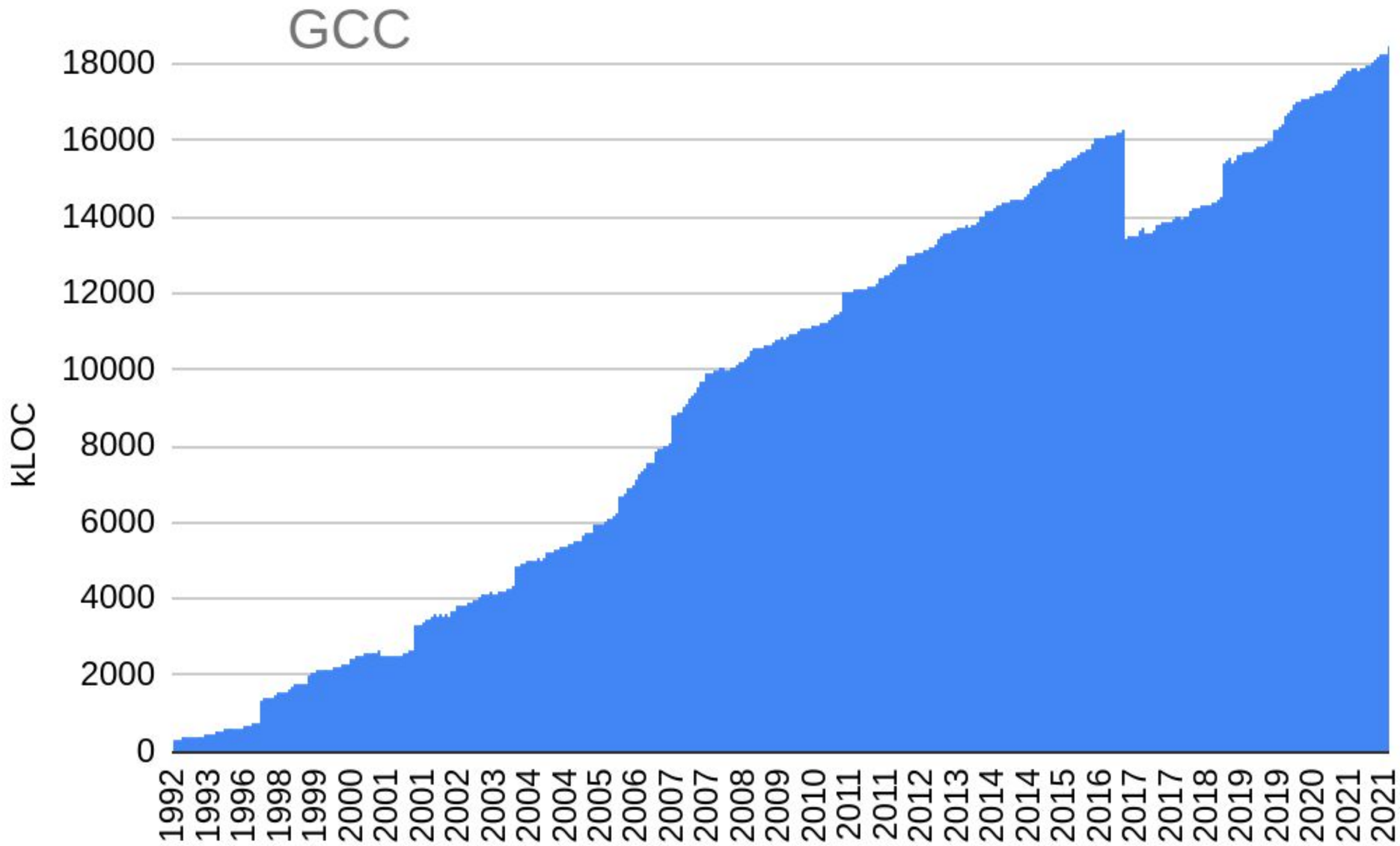


Compilers

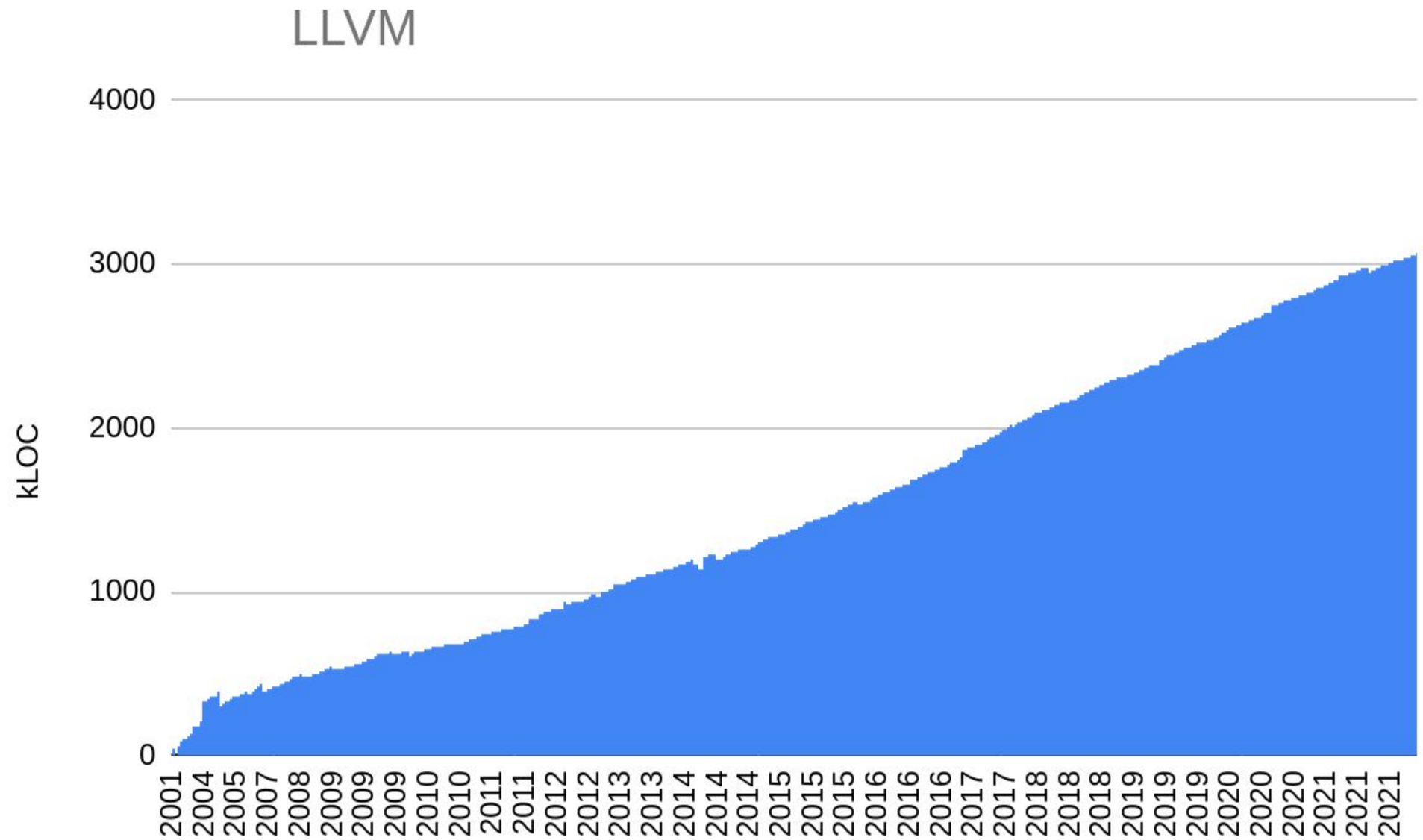
Architects

Image sources:
Tortoise: freeimages.com
Hare: worcswildlifetrust.co.uk

Compiler Complexity Growth

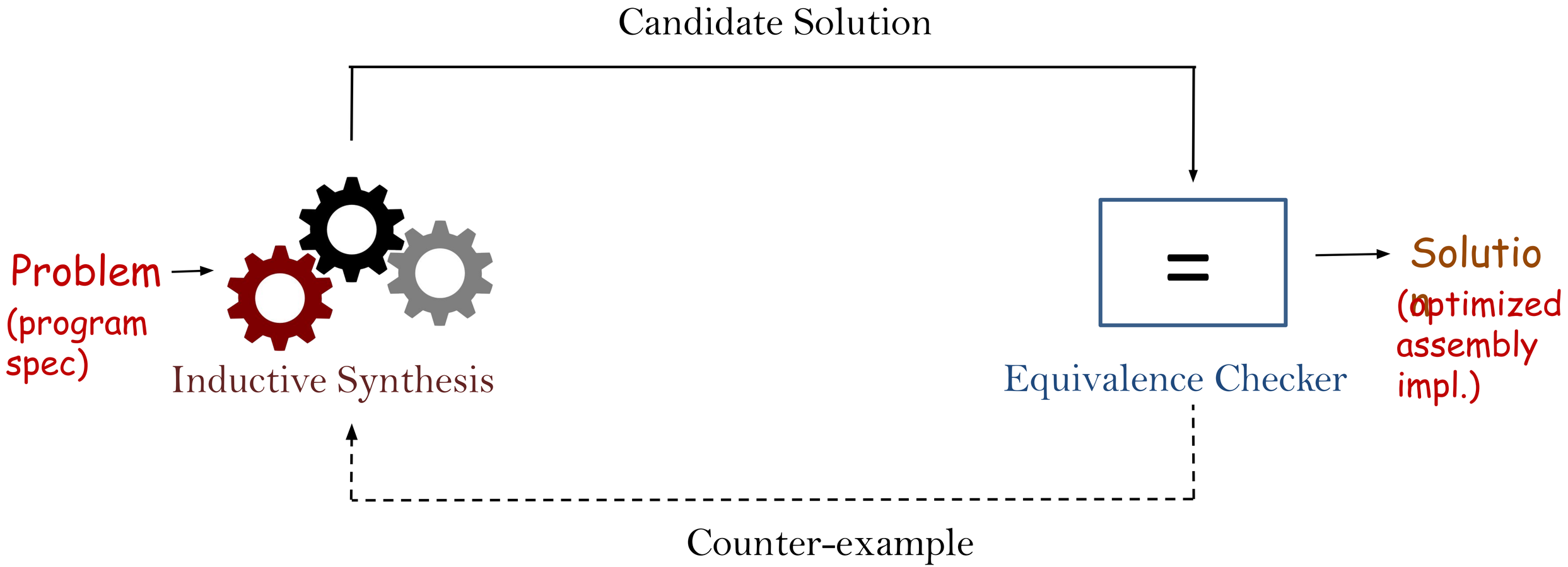


Compiler Complexity Growth

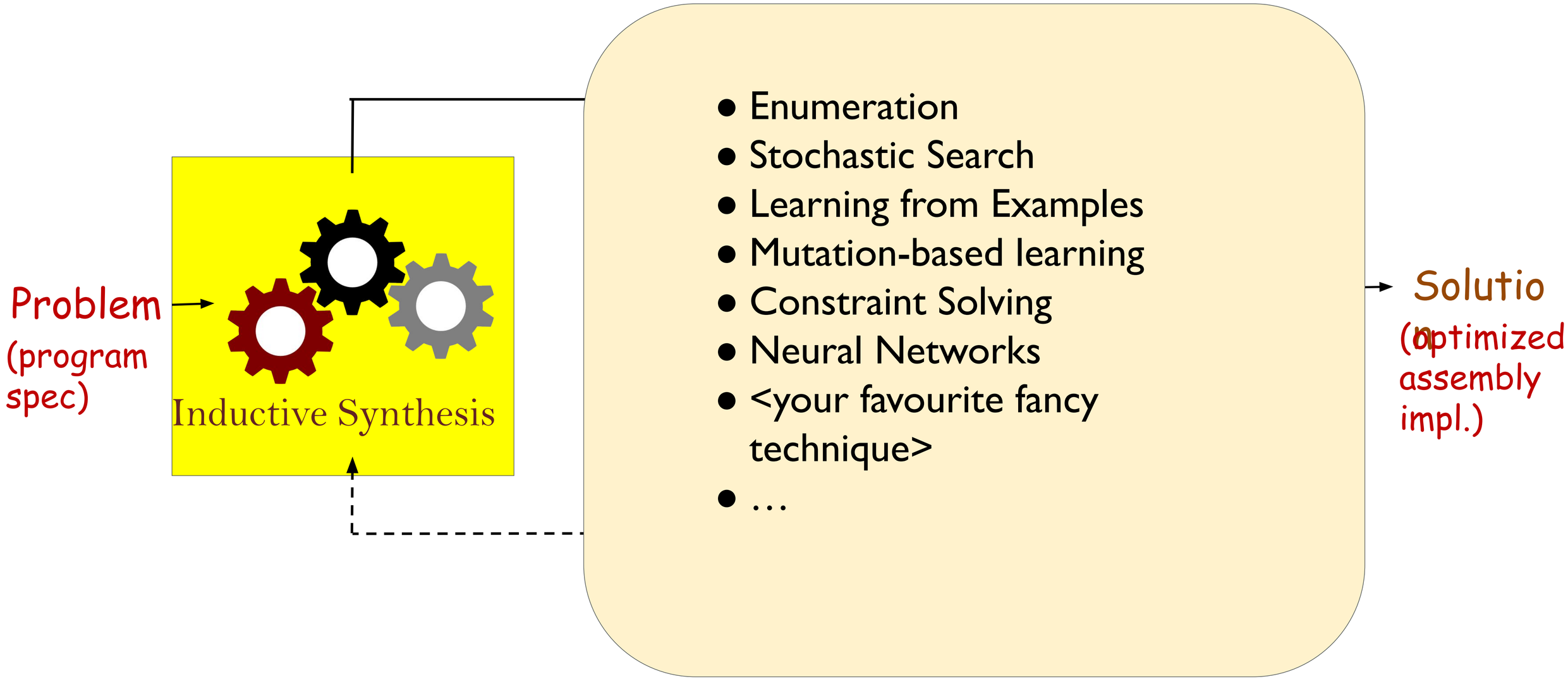


Superoptimization

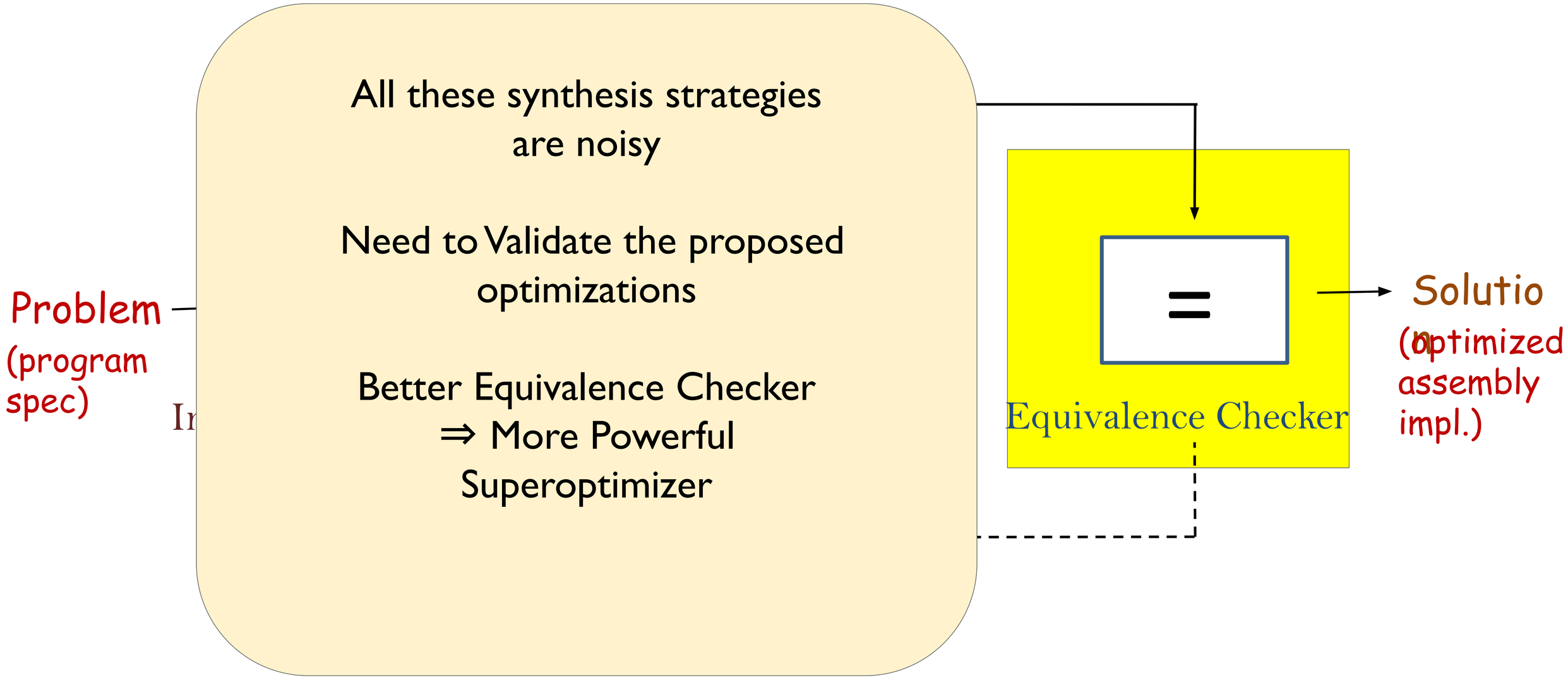
Superoptimization



Superoptimization



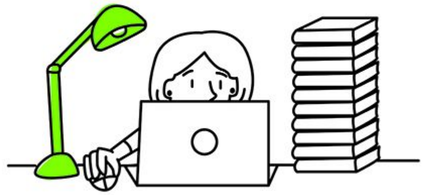
Superoptimization



Traditional Development Model for a Compiler

An engineer gets an optimization idea

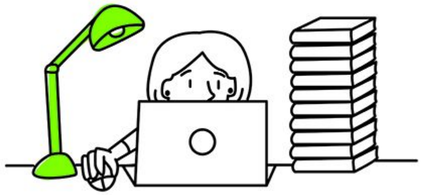
Traditional Development Model for a Compiler



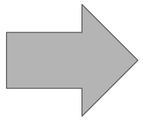
Codes it up

An engineer gets an optimization idea

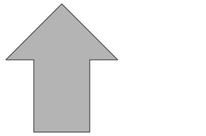
Traditional Development Model for a Compiler



Codes it up

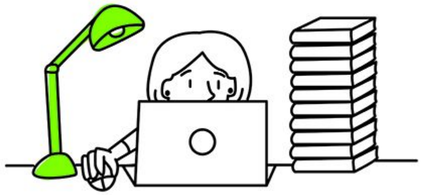


Reviewed by experts



An engineer gets an optimization idea

Traditional Development Model for a Compiler



Codes it up

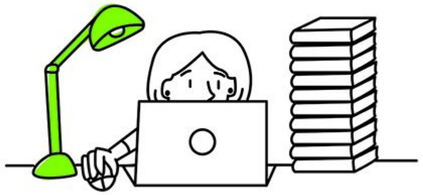
Reviewed by experts

An engineer gets an optimization idea

Reject

- Too complex compared to the benefit it entails
- Too specific to a certain PL
- Too specific to a certain architecture
- Requires compiler overhaul…

Traditional Development Model for a Compiler



Codes it up

Reviewed by experts

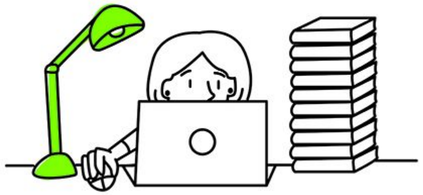
Too much human effort

An engineer gets an optimization idea

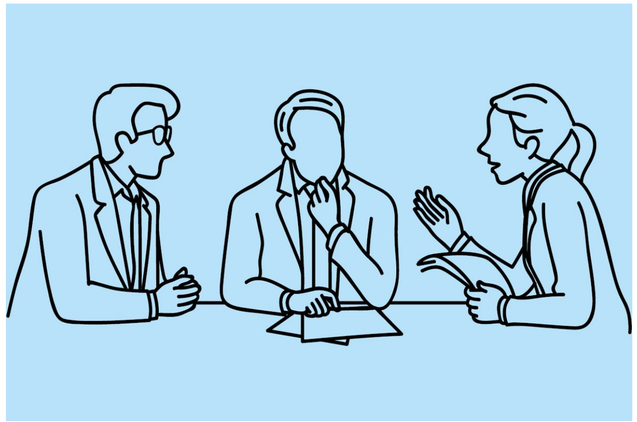
Reject

- Too complex compared to the benefit it entails
- Too specific to a certain PL
- Too specific to a certain architecture
- Requires compiler overhaul…

Traditional Development Model for a Compiler



Codes it up

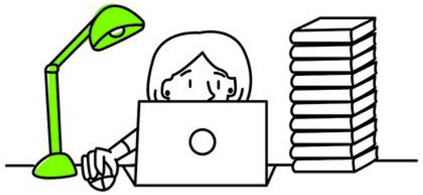


Reviewed by experts

Accept

An engineer gets an optimization idea

Traditional Development Model for a Compiler



Codes it up

Reviewed by experts

Accept

but wait…

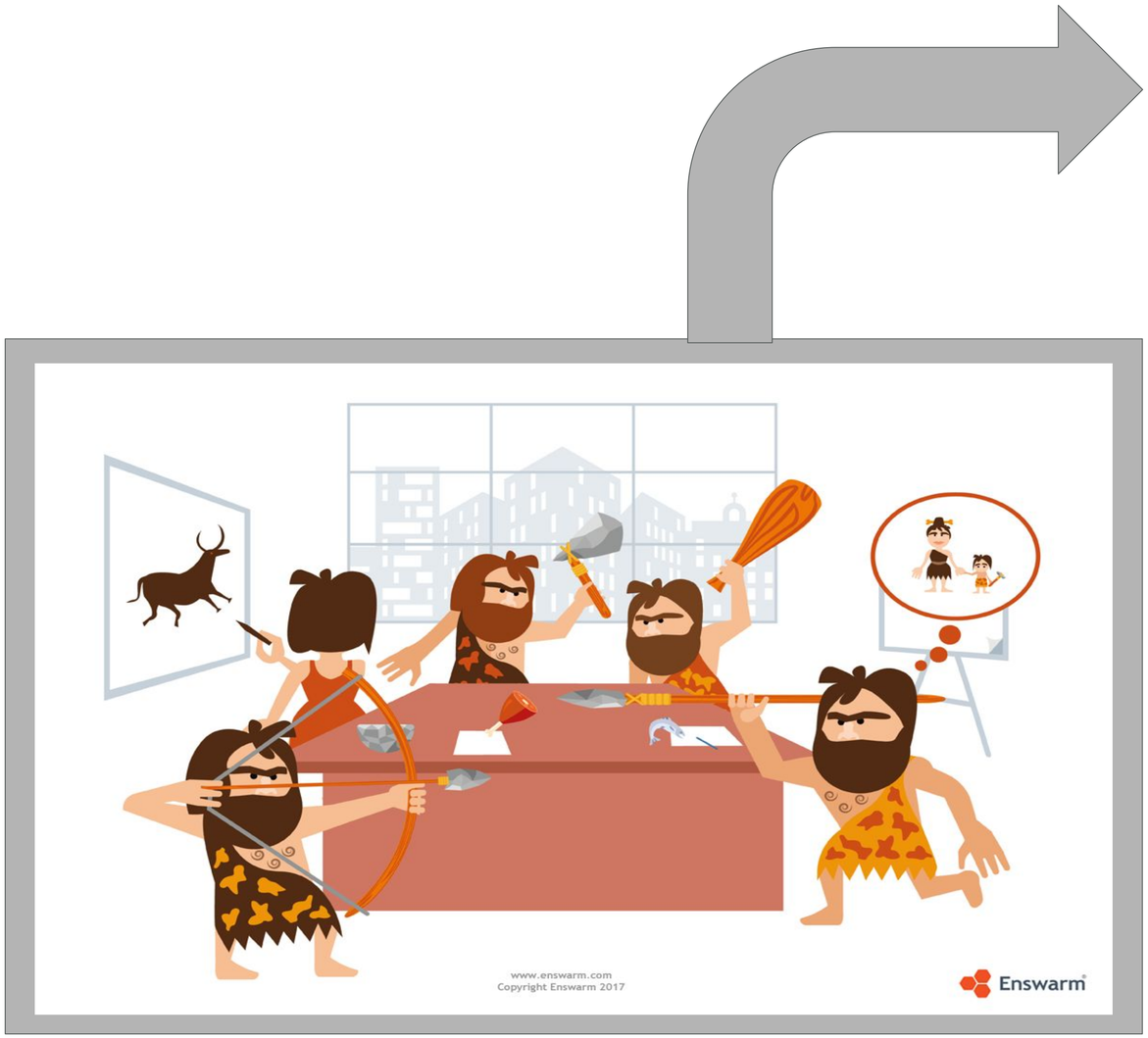
An engineer gets an optimization idea

Integer Overflow, Memory Overruns, Undefined Behaviour, …

compiler bugs are still common

Proposed Development Model

Equivalence Checker



=

Accept and Cache the problem-solution pair as a pattern-replacement

Reject

Developers of Inductive Synthesis Algorithms

Pattern-Replacement Examples

```
int signum(int x) {
    if (x > 0) return 1;
    if (x < 0) return –1;
    else return 0;
}
```

On Motorola 68020:

```
    add.l    d0, d0
    subx.l   d1, d1
    negx.l   d0
    addx.l   d1, d1
```

[H. Massalin. A Look at the Smallest Program, ASPLOS 1987]

[S. Bansal, A. Aiken. Automatic Generation of Peephole Superoptimizers, ASPLOS 2006]

Pattern-Replacement Examples

pattern	replacement
load (addr), reg store reg, (addr) *Support for memory accesses*	load (addr), reg
mul 2, reg	shl reg

mov r1, r2
mov r3, r1
mov r2,r3
live: r1,r3
	xchg r1, r3

sub %eax, %ecx
test %ecx, %ecx
je .END
mov %edx, %ebx
.END:
Support for branches
	sub %eax, %ecx
	cmovne %edx, %ebx

[S. Bansal, A. Aiken. Automatic Generation of Peephole Superoptimizers, ASPLOS 2006]

Pattern-Replacement Examples

pattern	replacement

sum += a[i];
sum += a[i+1];
…
sum += a[i+7];
	psubb %mm0, %mm0
	psadbw &a[i], %mm0
	movd %mm0, sum
	Use of vector instructions

sub %eax, %ecx
mov %ecx, %eax
dec %eax
live: %eax
	not %eax
	add %ecx, %eax

setg %al movzbl %al, %eax dec %eax and %eax, %esi live: %esi	mov $0, %eax cmovg %eax, %esi *Use of conditional-moves*

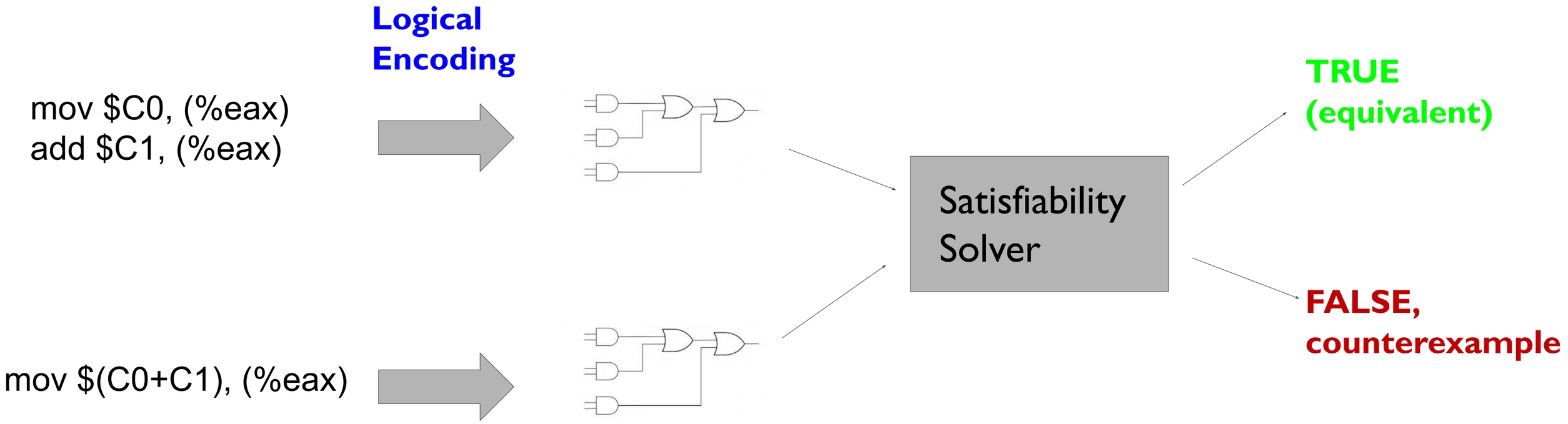
Pattern-Replacement Examples

Support for symbolic constants in pattern and replacement

pattern	replacement
mov $C0, %eax dec %eax	mov $(C0-1), %eax
mov $C0, (%eax) add $C1, (%eax)	mov $(C0+C1), (%eax)

[S. Bansal, A. Aiken. Automatic Generation of Peephole Superoptimizers, ASPLOS 2006]

Equivalence Checker



[S. Bansal, A. Aiken. Automatic Generation of Peephole Superoptimizers, ASPLOS 2006]

Important Limitations

- Loops are not supported

pattern	replacement

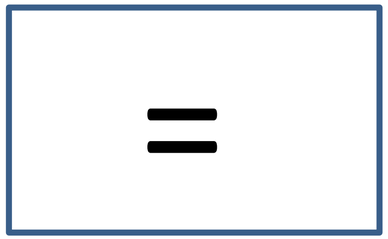
for (...) {
  r = *p;
  use(r); //*p remains unchanged
  *p = r;
}
	r = *p;
	for (...) {
	  use(r); //*p remains unchanged
	}

Important Limitations

- Aliasing information is not captured, e.g., heap access vs. stack access

pattern	replacement
load (stack-addr), reg access (heap-addr) store reg, (stack-addr)	load (stack-addr), reg access (heap-addr)

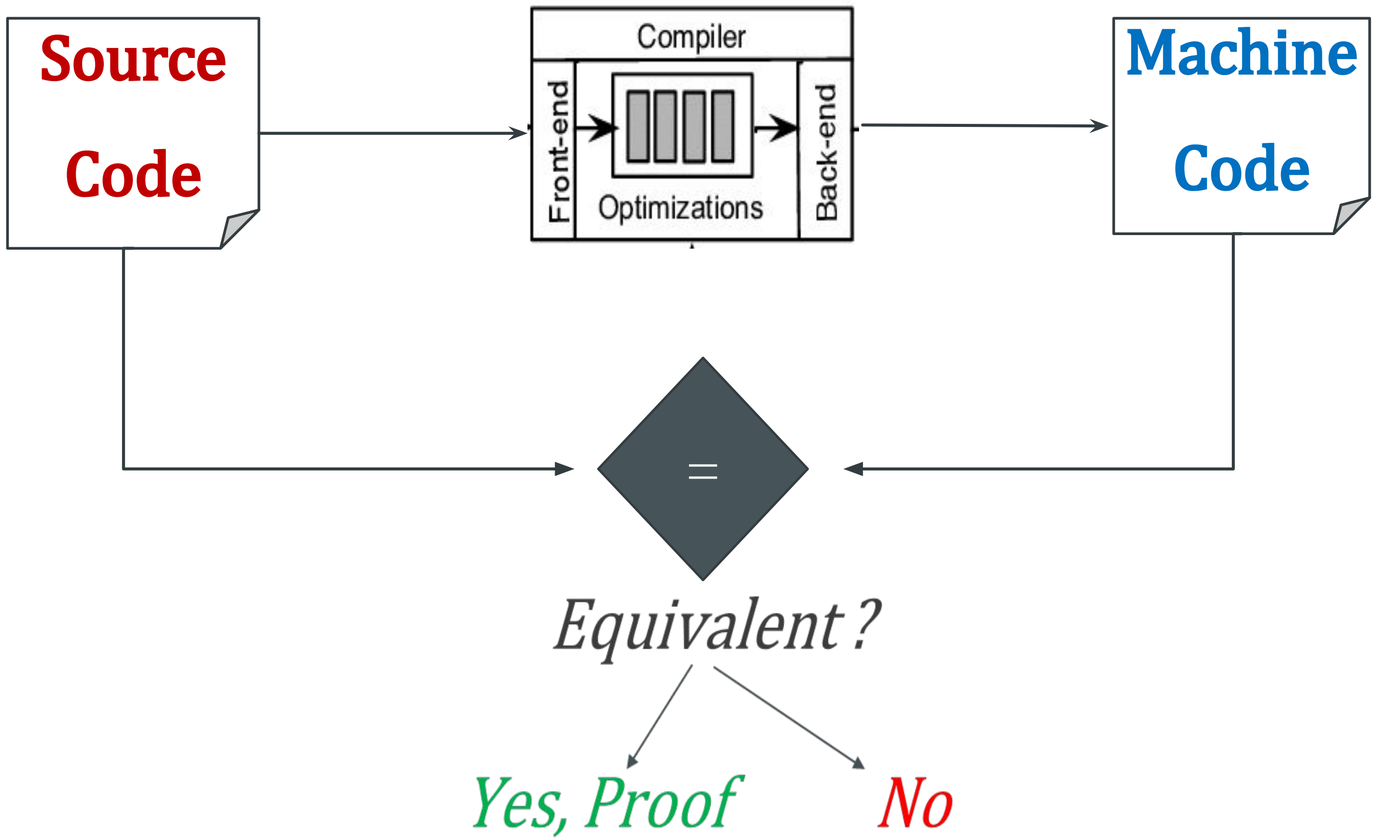
Equivalence Checker



Equivalence Checker

End-to-End, support for loops, aliasing information, function calls, …
No false-positives (sound)
Minimize false-negatives

Translation Validation



Equivalence Checking – Program with loops

```
void divP ( ) {
  for( int i=0; i < 1024; i++) {
    a[i] = b[i]/2;
  }
}
```

C Program

```
void shiftP ( ) {
  for(int r1=0; r1 < 1024; r1++) {
    a[r1] = b[r1] >> 1;
  }
}
```

(abstracted) Assembly

Equivalence Checking – Program with loops

Given, $Input_C == Input_A$

$Memory_C == Memory_A$

```
void divP ( ) {
  for( int i=0; i < 1024; i++) {
    a[i] = b[i]/2;
  }
}
```

C Program

```
void shiftP ( ) {
   for(int r1=0; r1 < 1024; r1++) {
      a[r1] = b[r1] >> 1;
   }
}
```

(abstracted) Assembly

Equivalence Checking – Program with loops

Given, $Input_C == Input_A$

$Memory_C == Memory_A$

```
void divP ( ) {
  for( int i=0; i < 1024; i++) {
    a[i] = b[i]/2;
  }
}
```

$Return_C == Return_A$

$Memory_C == Memory_A$?

C Program

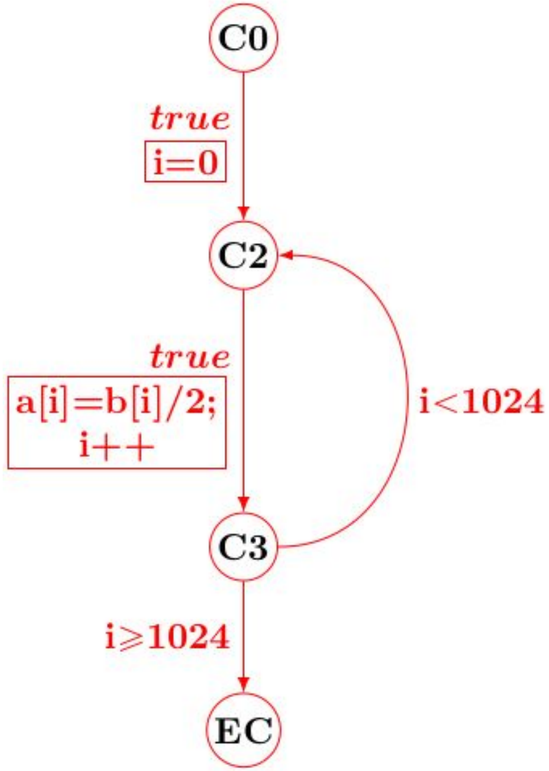
```
void shiftP ( ) {
  for(int r1=0; r1 < 1024; r1++) {
    a[r1] = b[r1] >> 1;
  }
}
```

(abstracted) Assembly

Control Flow Graph (CFG)

```
C0:  void divP ( ) {
C1:     for( int i=0; i < 1024; ) {
C2:        a[i] = b[i]/2;  i++;
C3:     }
EC:  }
```

C Program



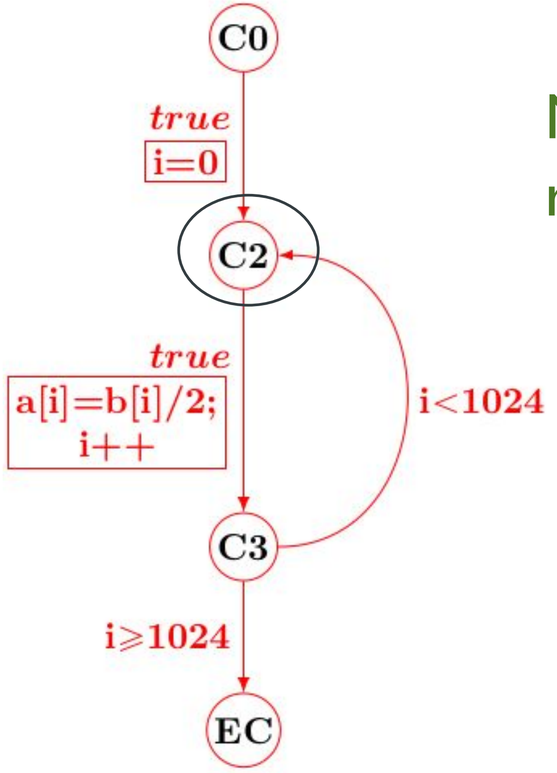
Nodes are PCs

Edges involve transfer functions

Control Flow Graph (CFG)

```
C0:  void divP ( ) {
C1:     for( int i=0; i < 1024; ) {
C2:        a[i] = b[i]/2;  i++;
C3:     }
EC:  }
```

C Program



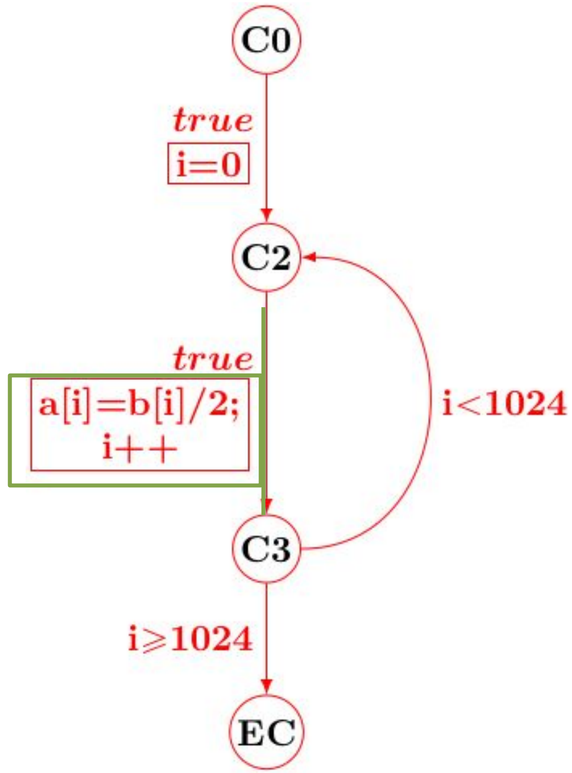
Node C2 represents a PC

Control Flow Graph (CFG)

```
C0:  void divP ( ) {
C1:     for( int i=0; i < 1024; ) {
C2:        a[i] = b[i]/2;  i++;
C3:     }
EC:  }
```

C Program

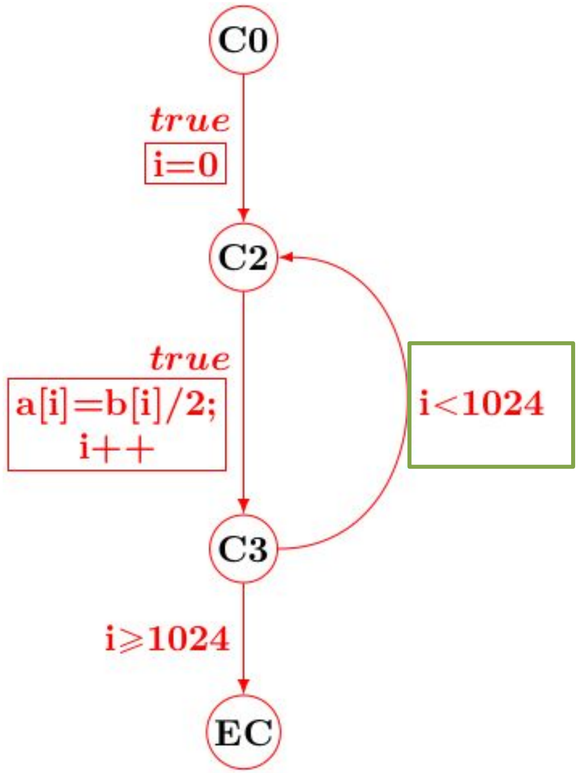
Transfer function of C2→C3



Control Flow Graph (CFG)

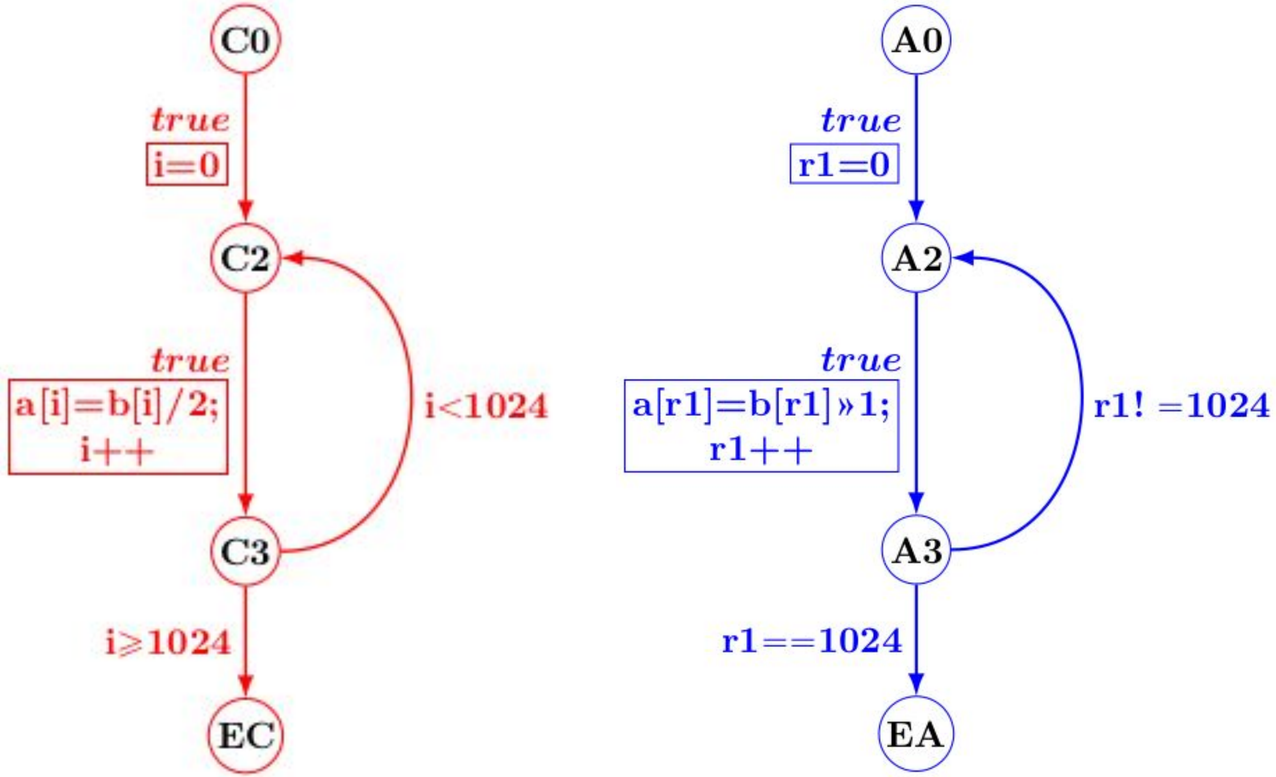
```
C0:  void divP ( ) {
C1:     for( int i=0; i < 1024; ) {
C2:        a[i] = b[i]/2;  i++;
C3:     }
EC:  }
```

C Program



Edge Condition of C3→C2

Product Program Construction

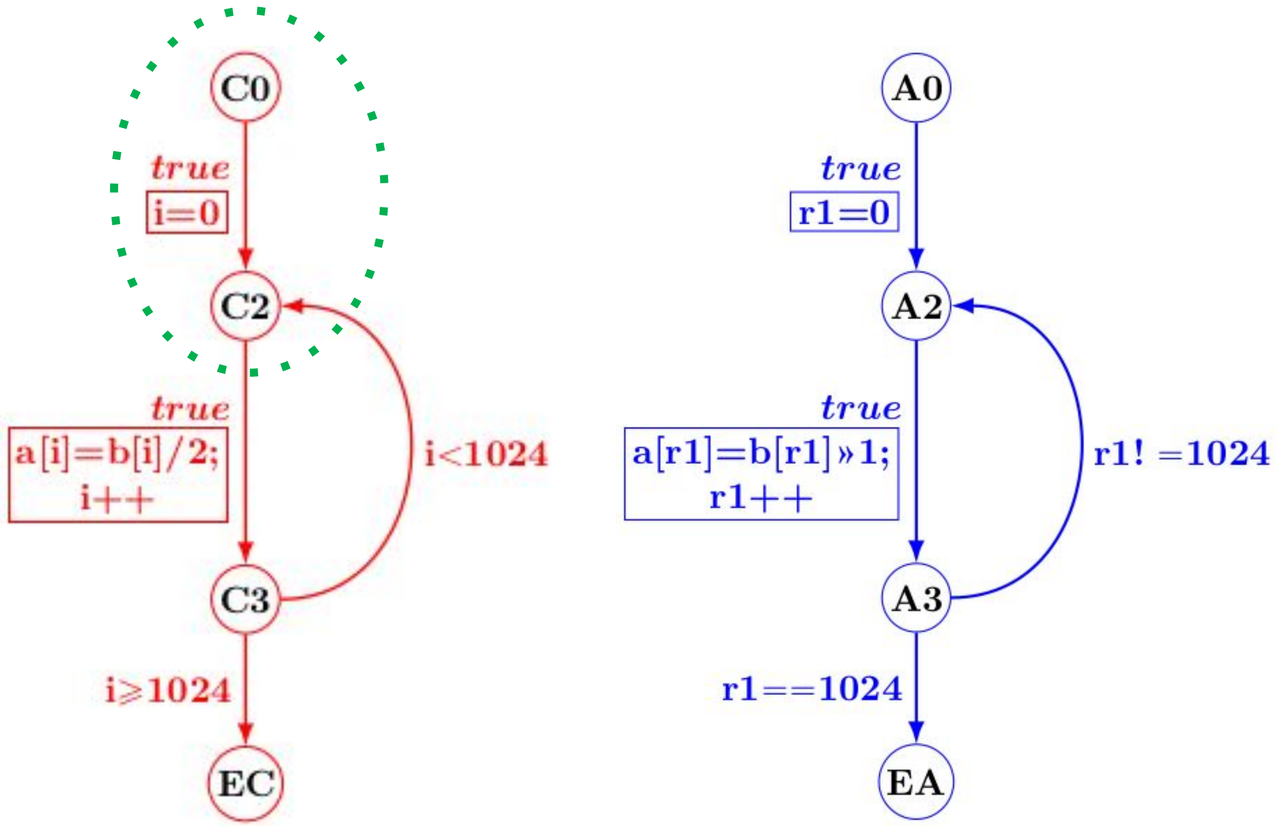


Goal: Construct a **Product Program** that executes both programs in lockstep

such that the two programs' states always remain related.

** G. C. Necula: Translation validation for an optimizing compiler. PLDI'2000

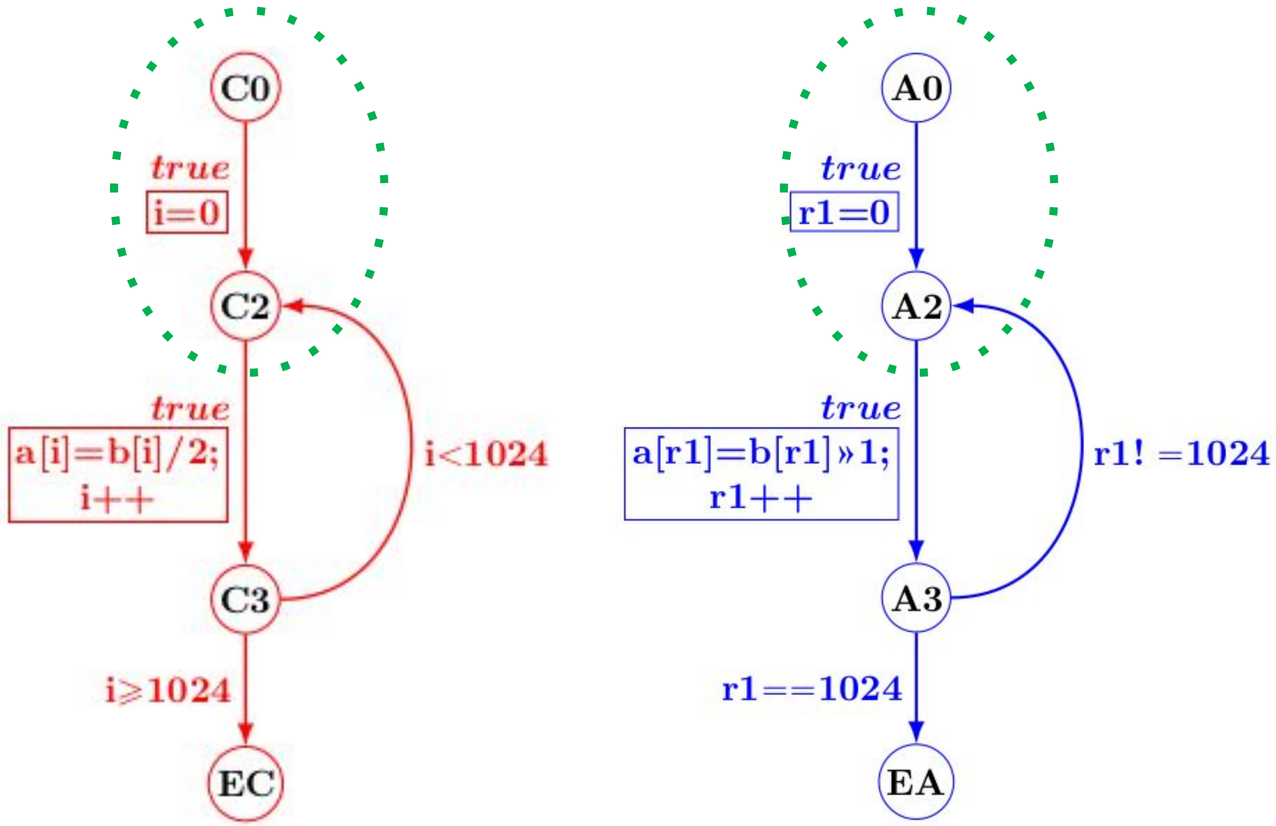
Product Program Construction



Goal: Construct a **Product Program** that executes both programs in lockstep

such that the two programs' states always remain related.

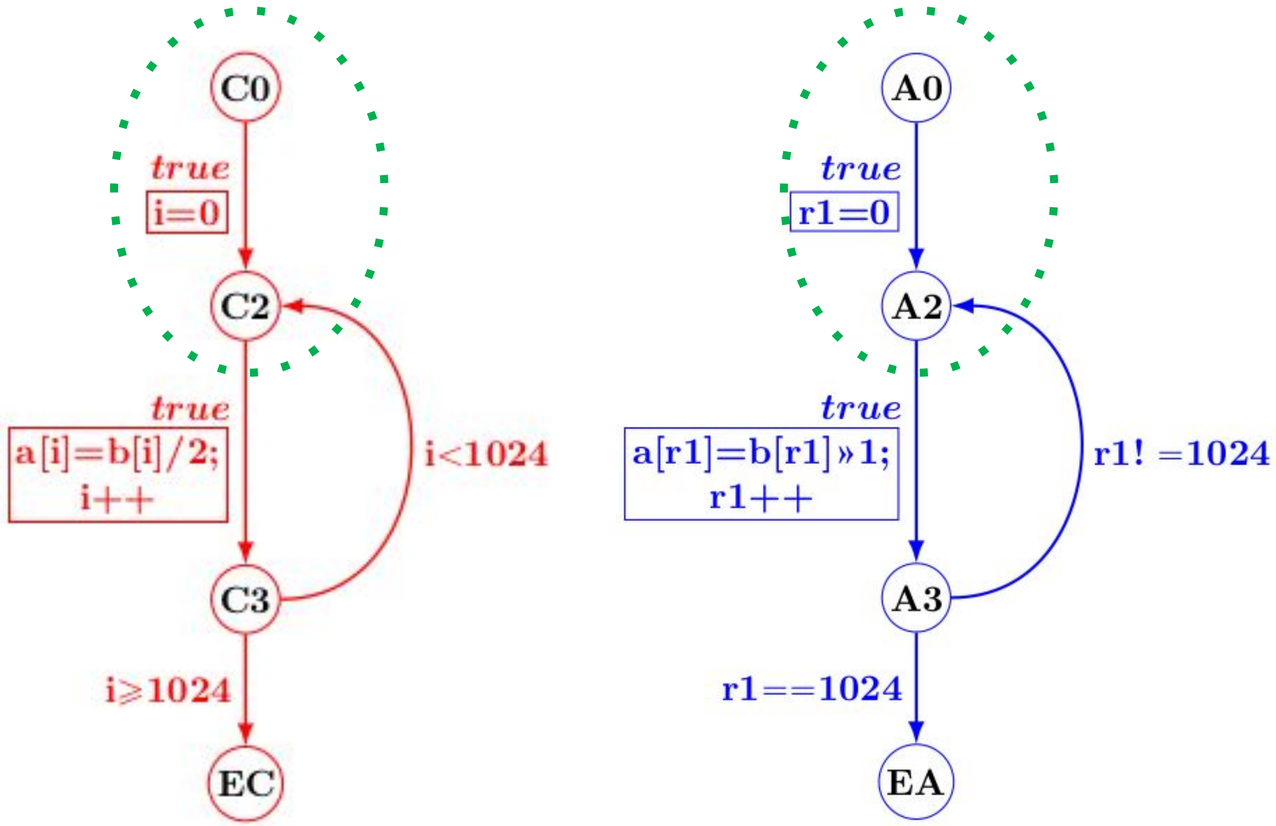
Product Program Construction

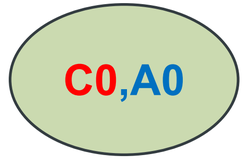


Goal: Construct a **Product Program** that executes both programs in lockstep

such that the two programs' states always remain related.

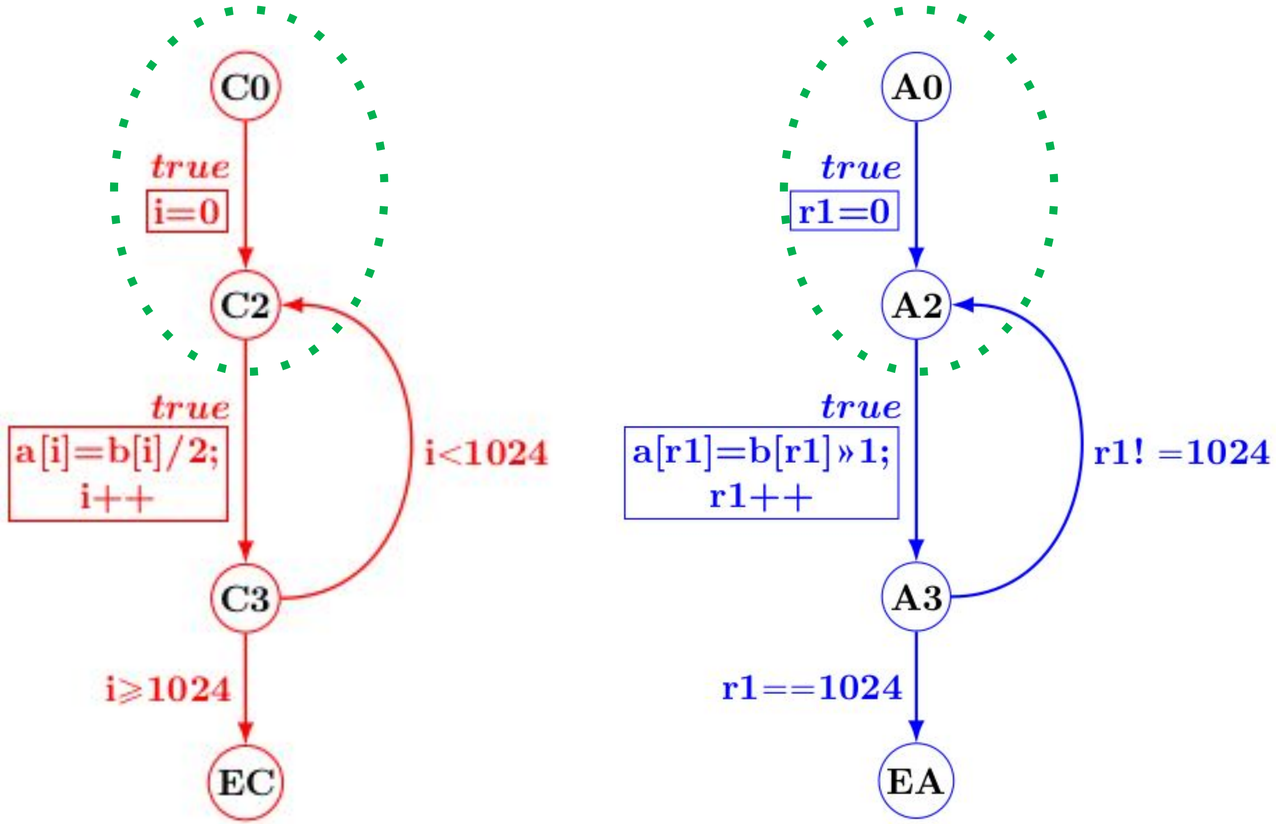
Product Program Construction

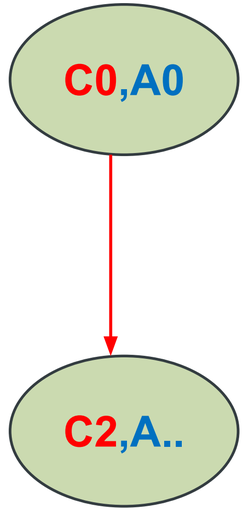




Product CFG

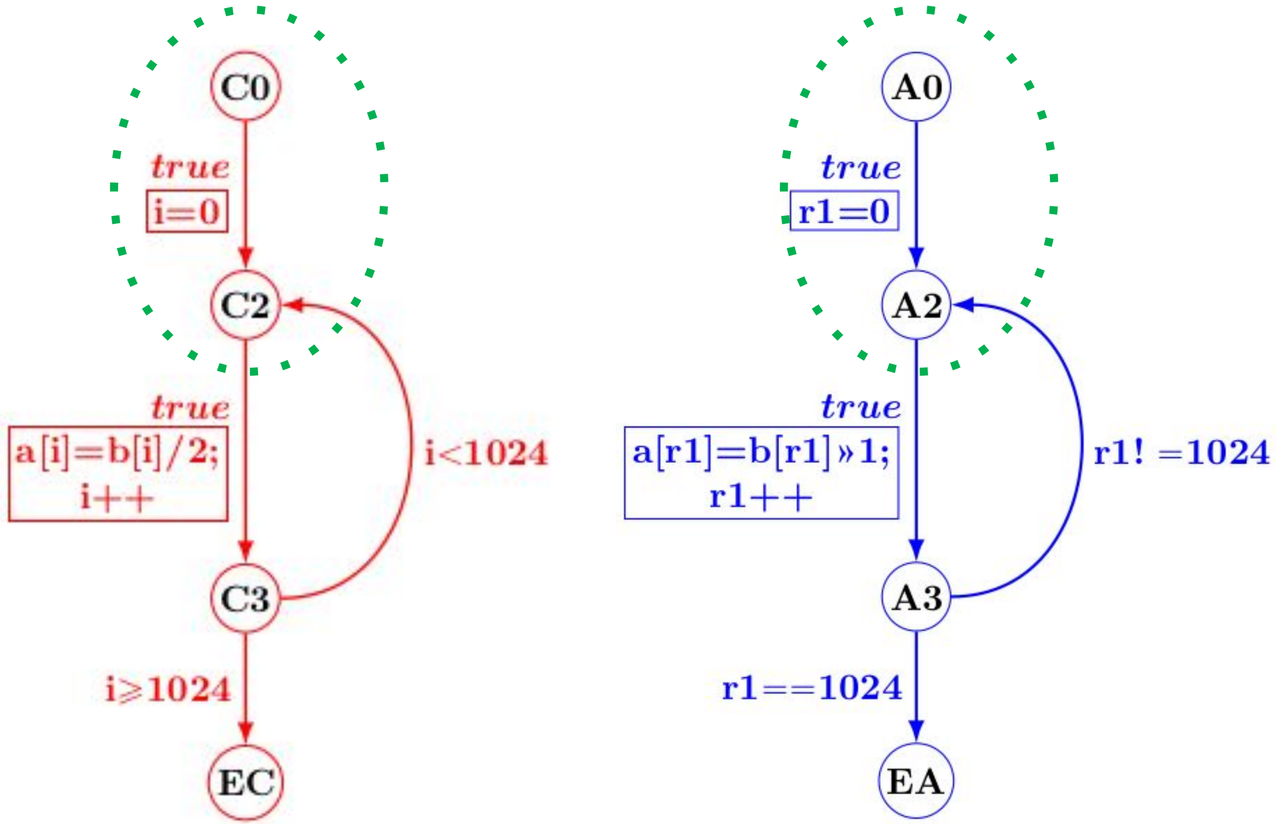
Product Program Construction

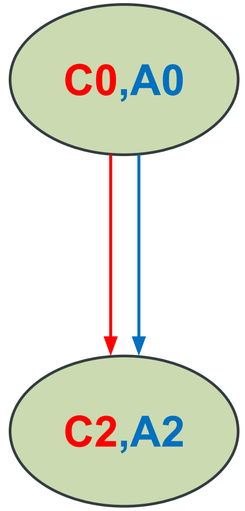




Product CFG

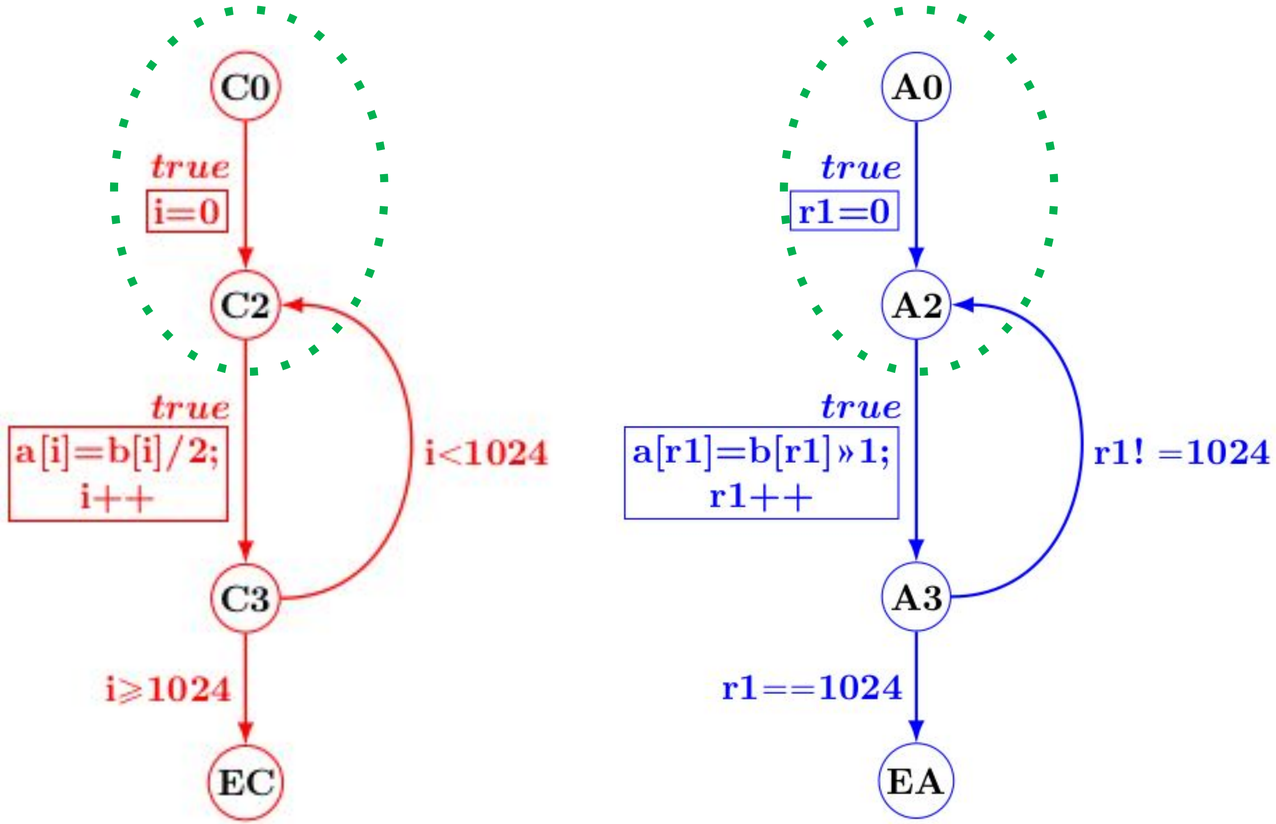
Product Program Construction



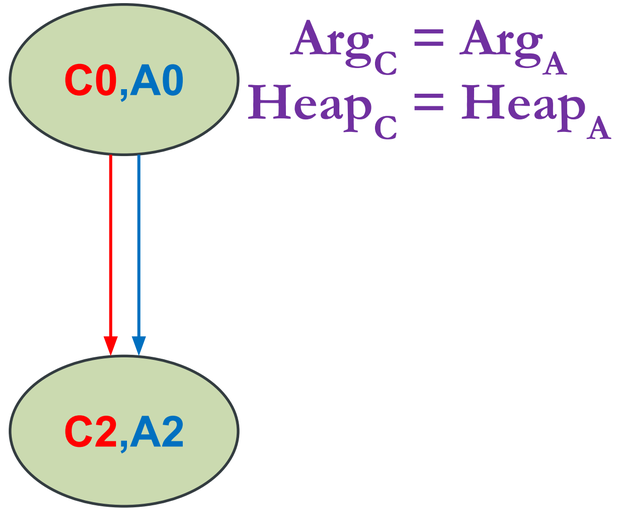


Product CFG

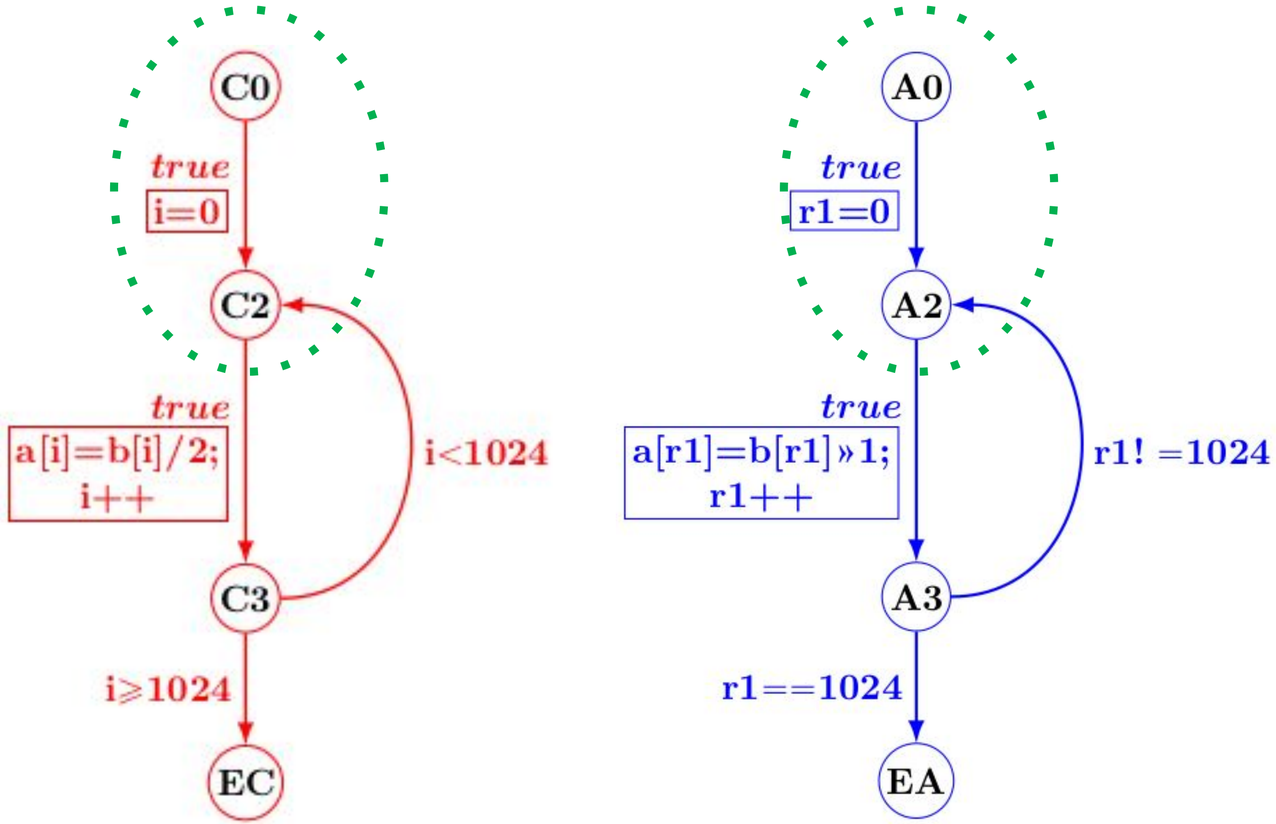
Product Program Construction



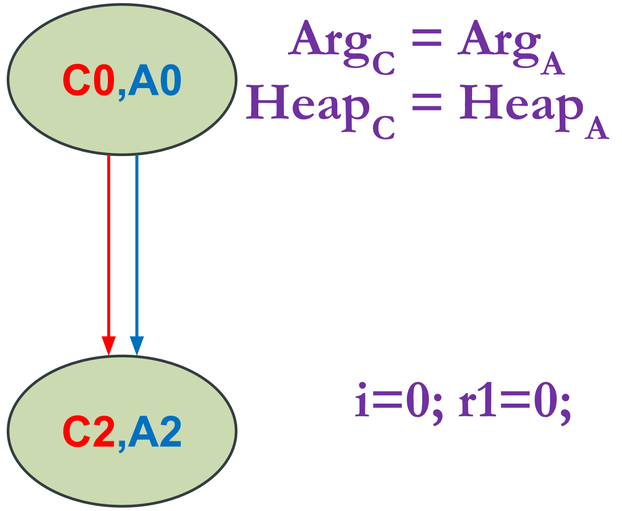
Product CFG



Product Program Construction



Product CFG

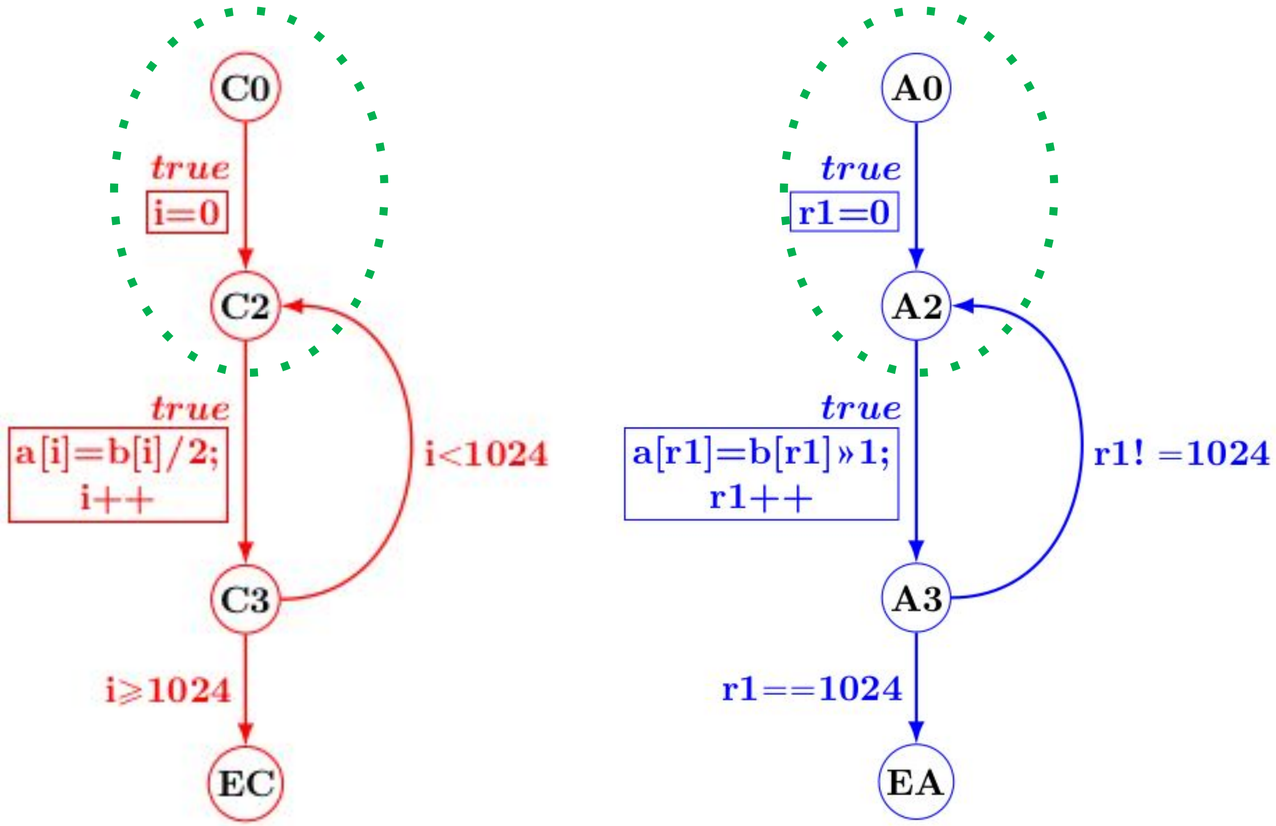


$Arg_C = Arg_A$

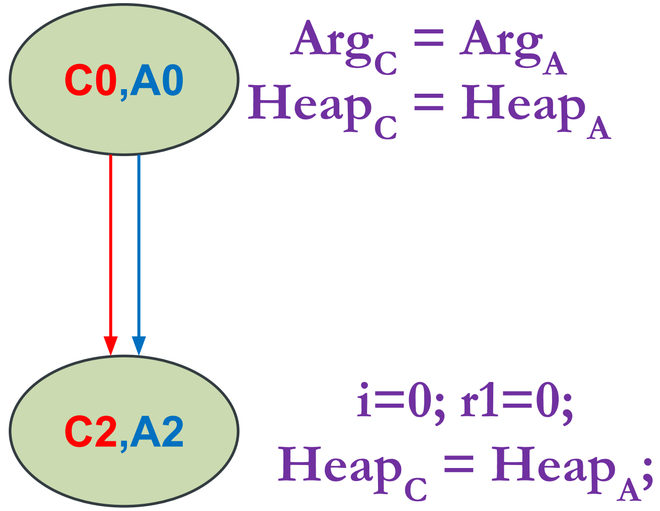
$Heap_C = Heap_A$

i=0; r1=0;

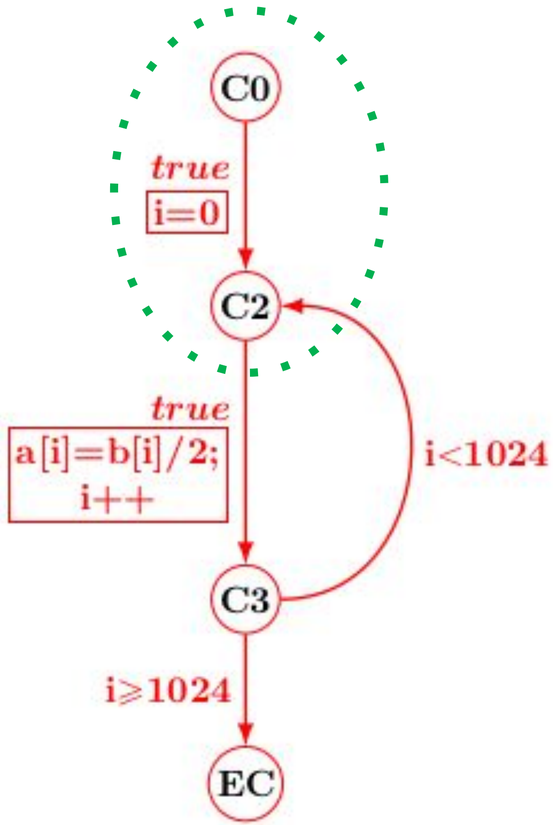
Product Program Construction

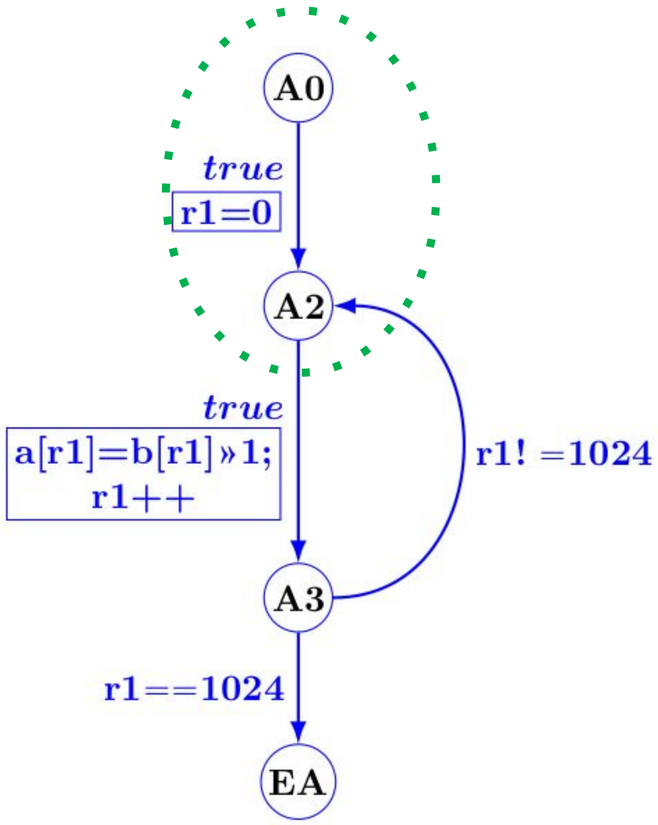


Product CFG

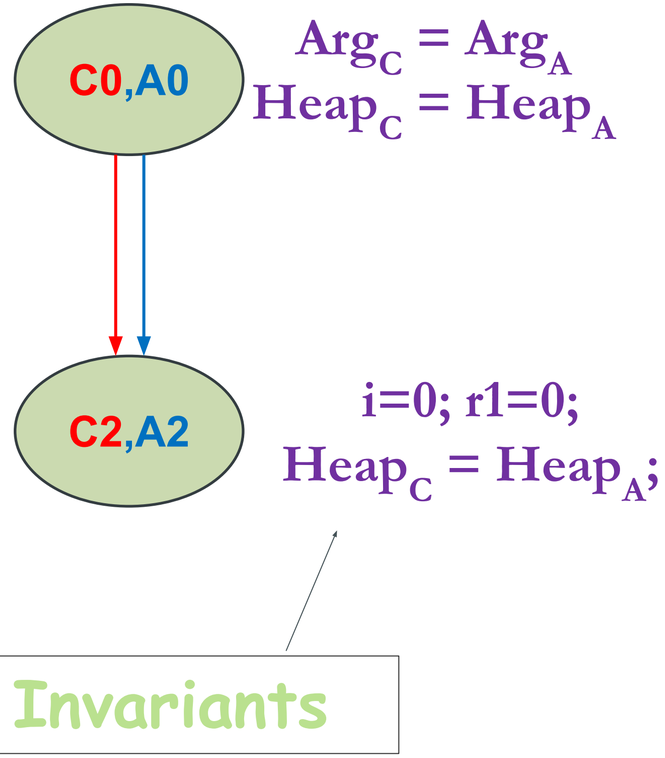


Product Program Construction



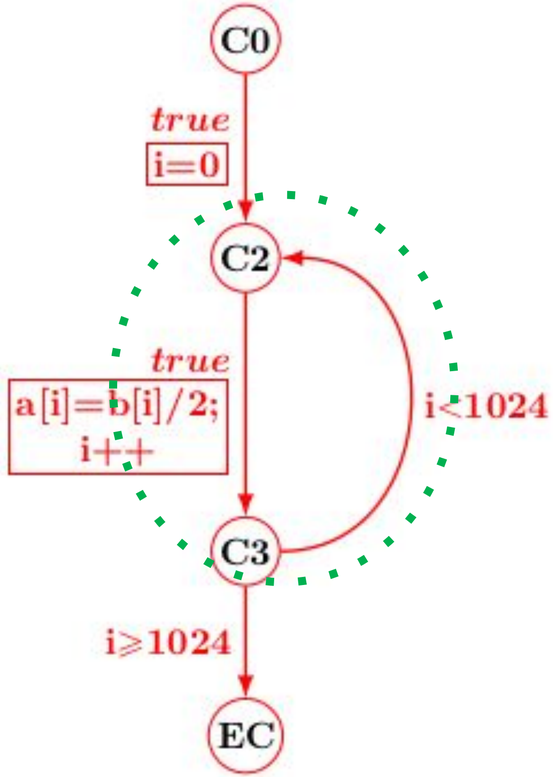


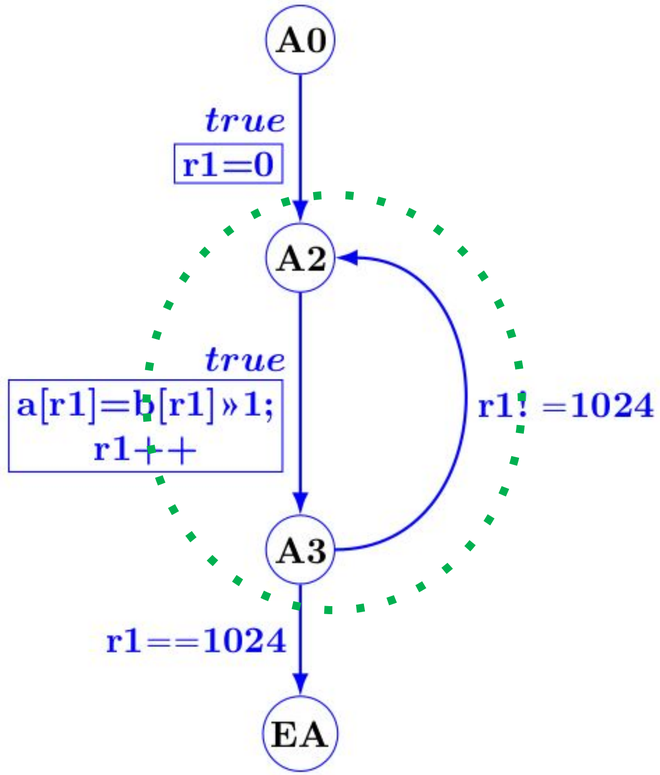
Product CFG

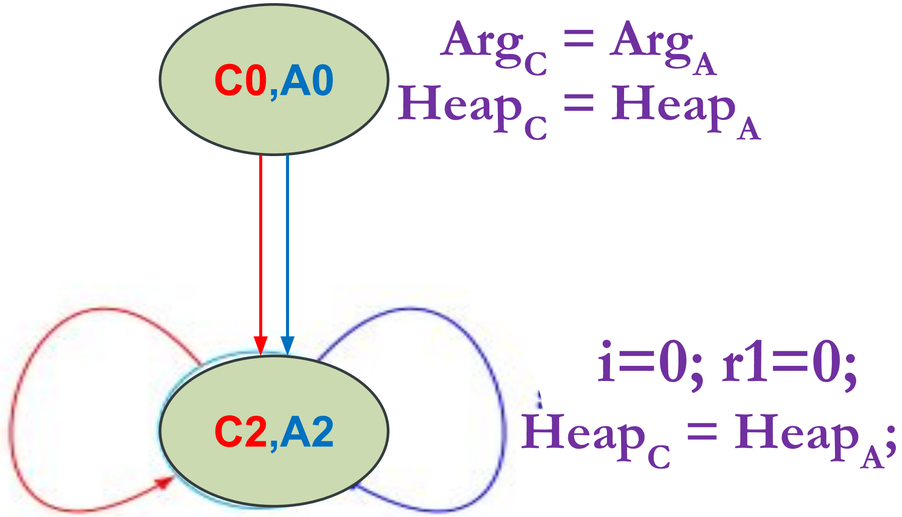


Product Program Construction

Product CFG

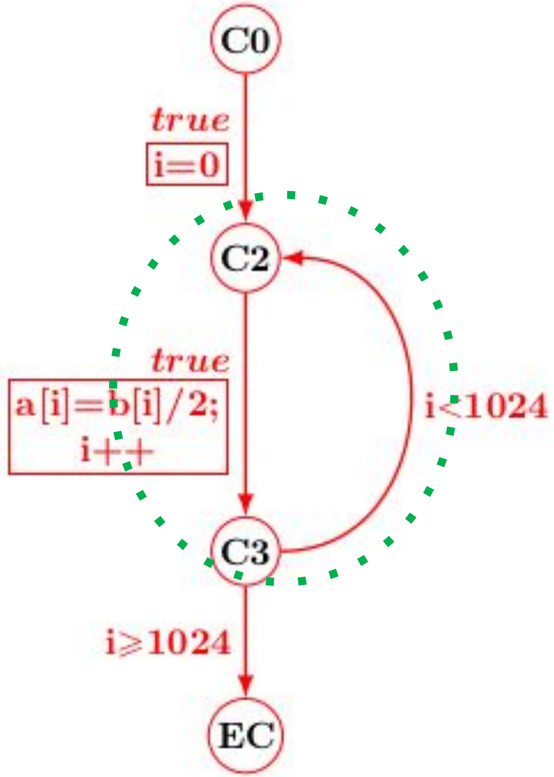


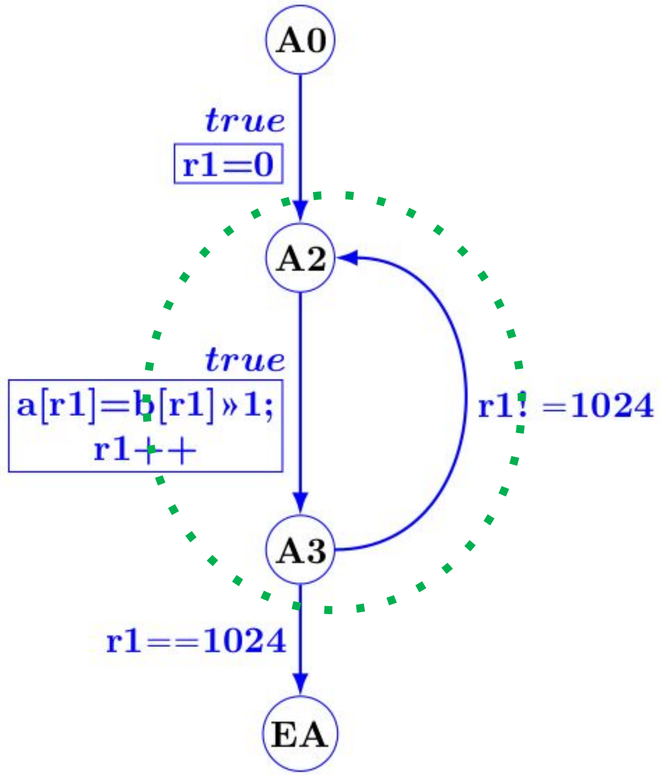


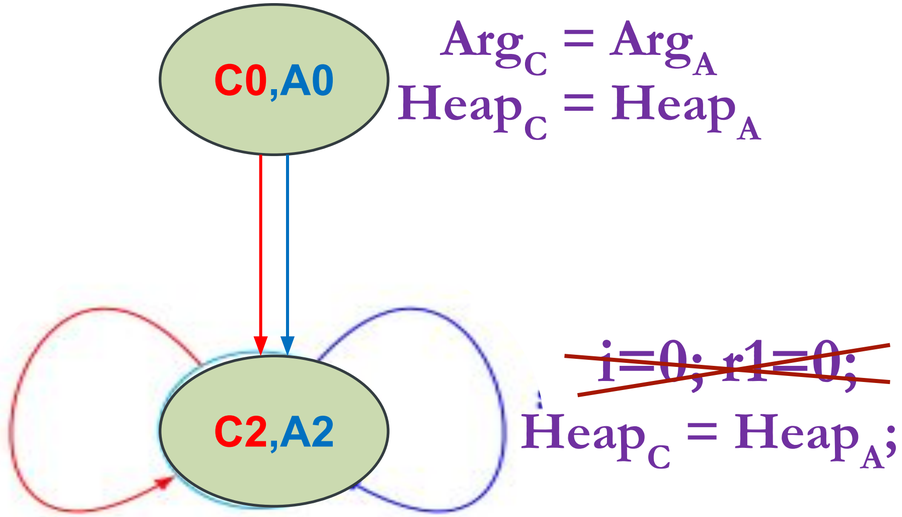


Product Program Construction

Product CFG

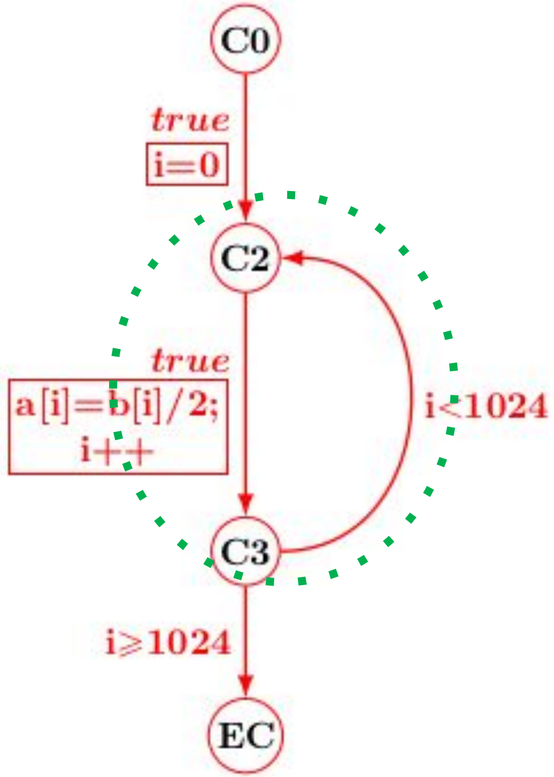


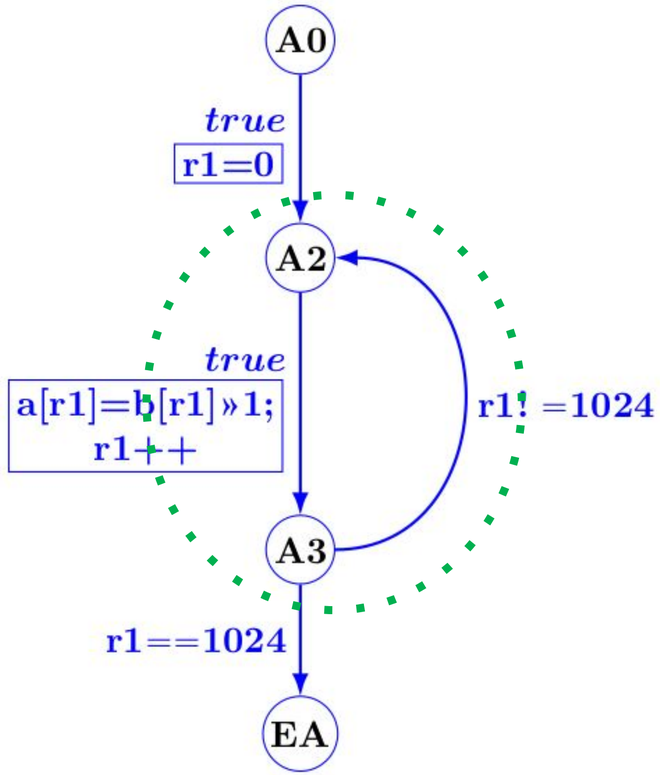


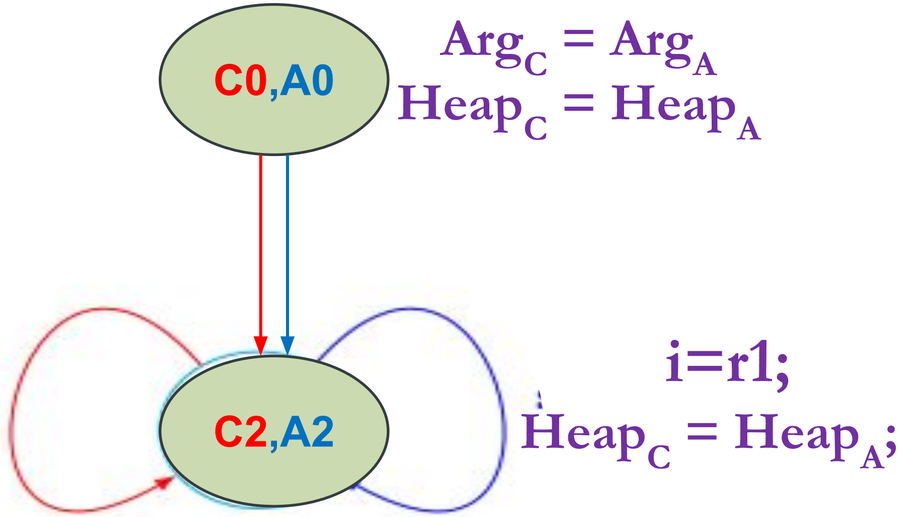


Product Program Construction

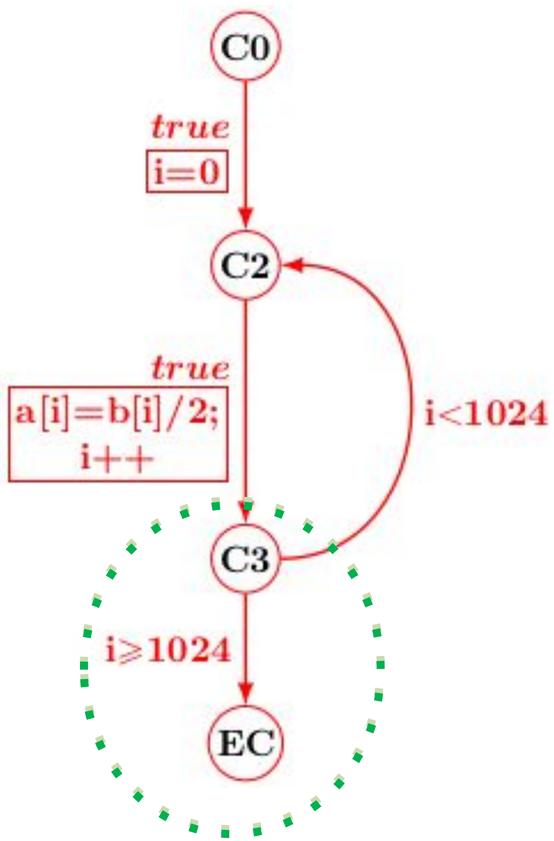
Product CFG

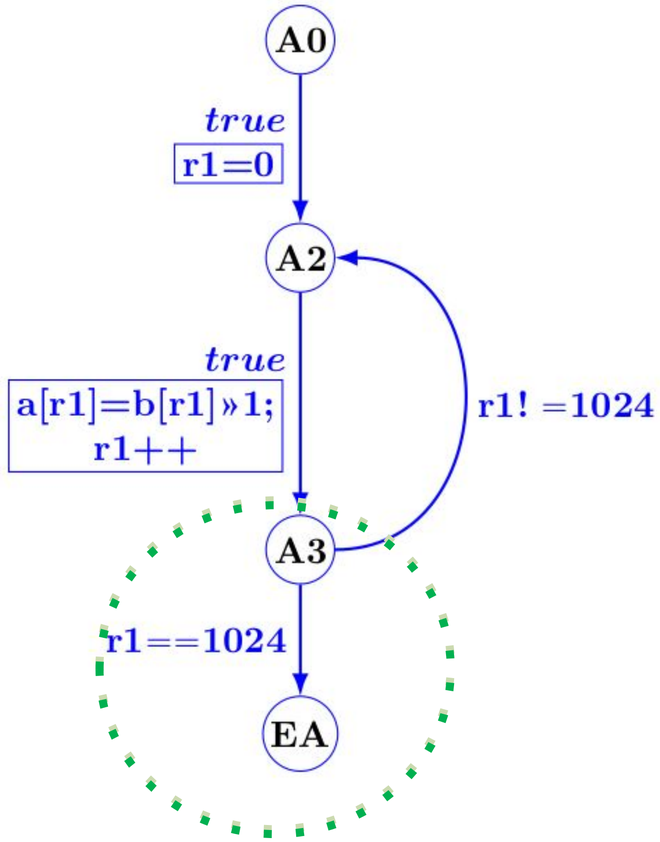




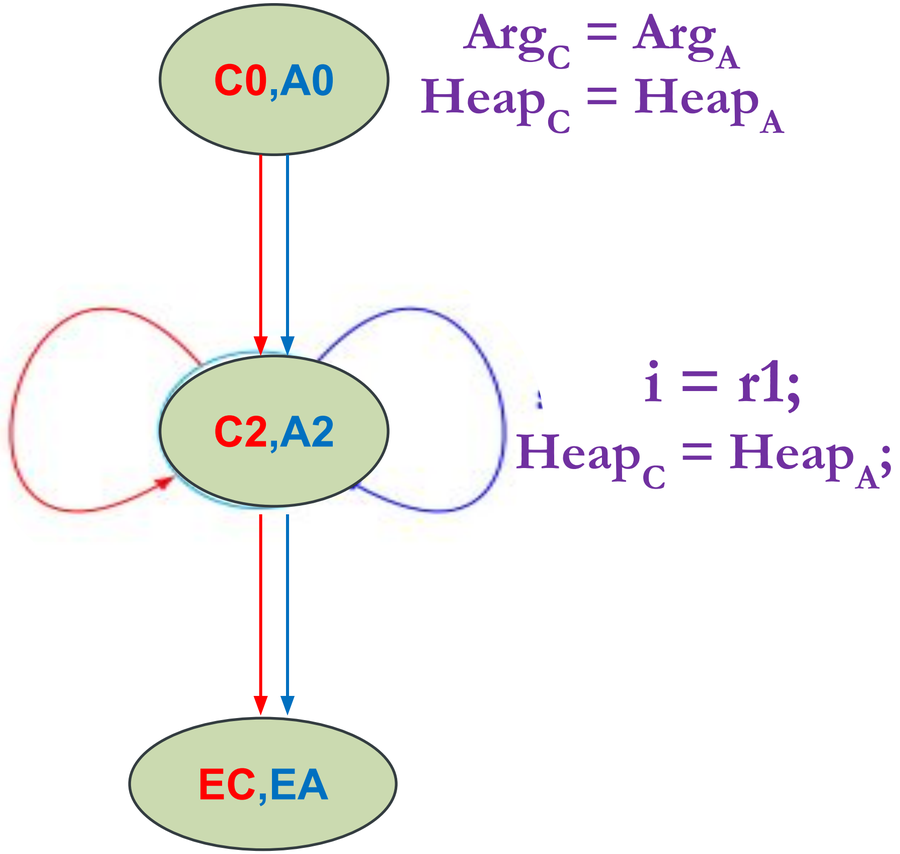


Product Program Construction

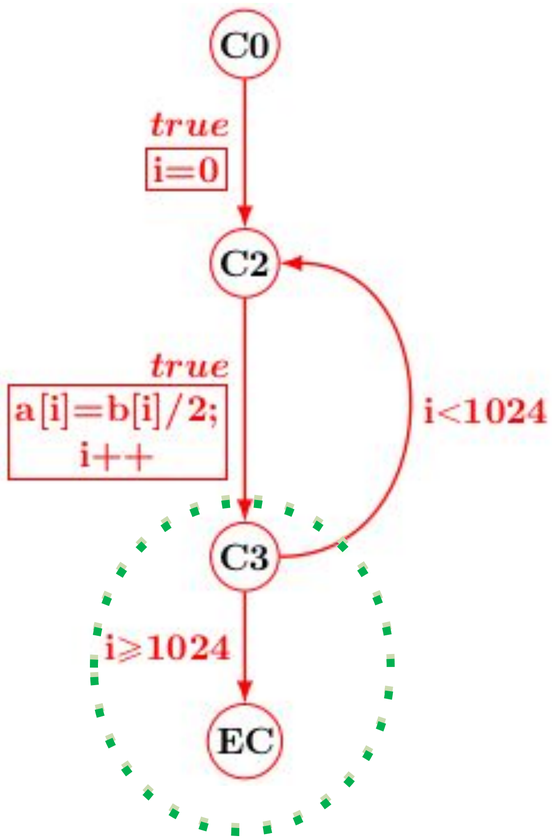


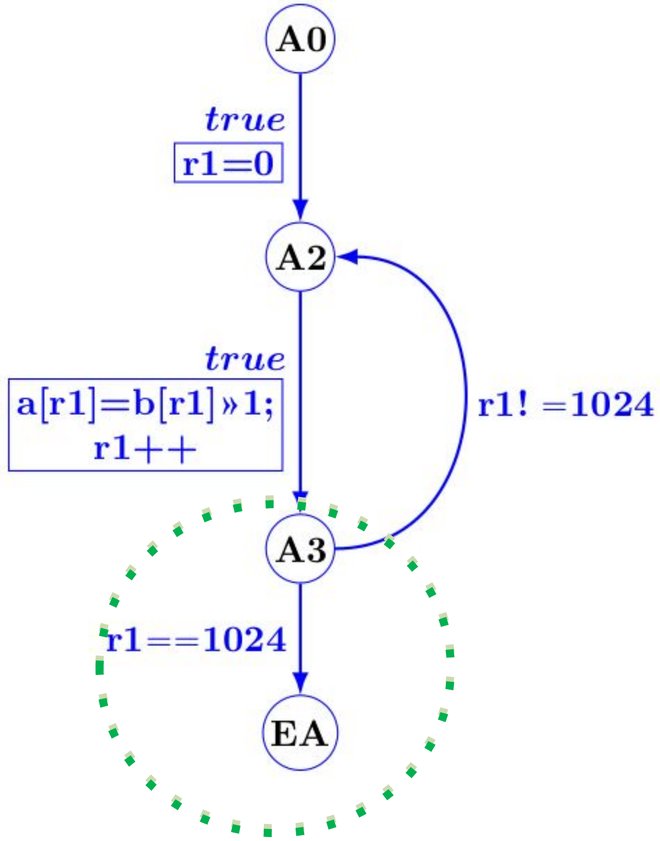


Product CFG

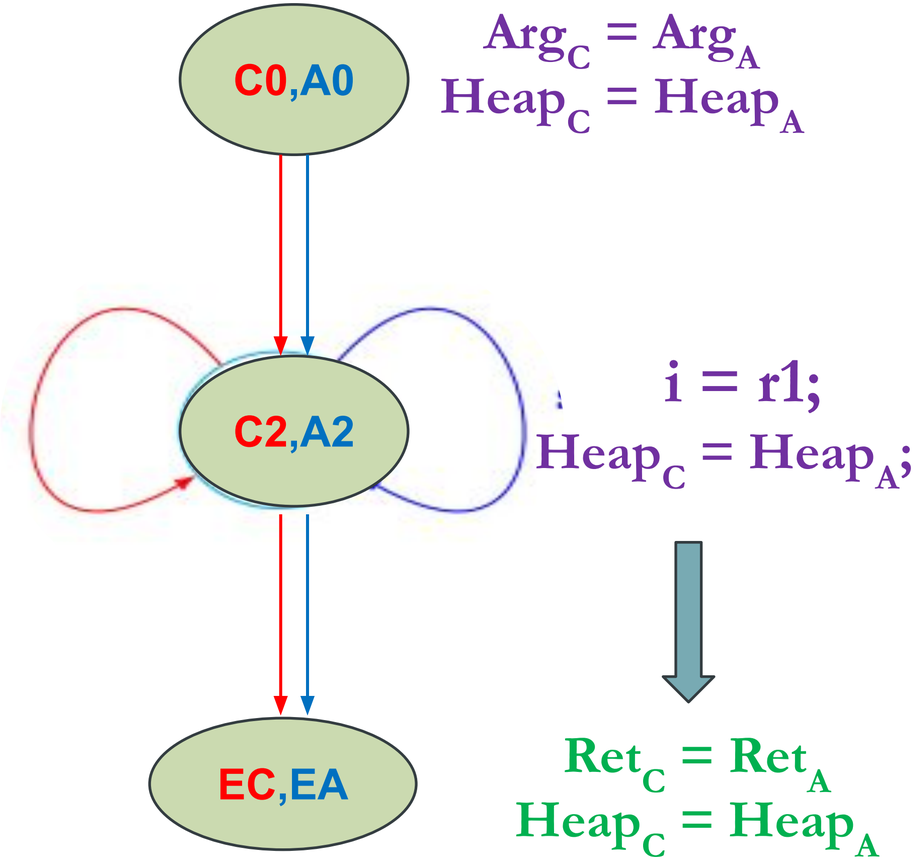


Prove at exit points





Product CFG

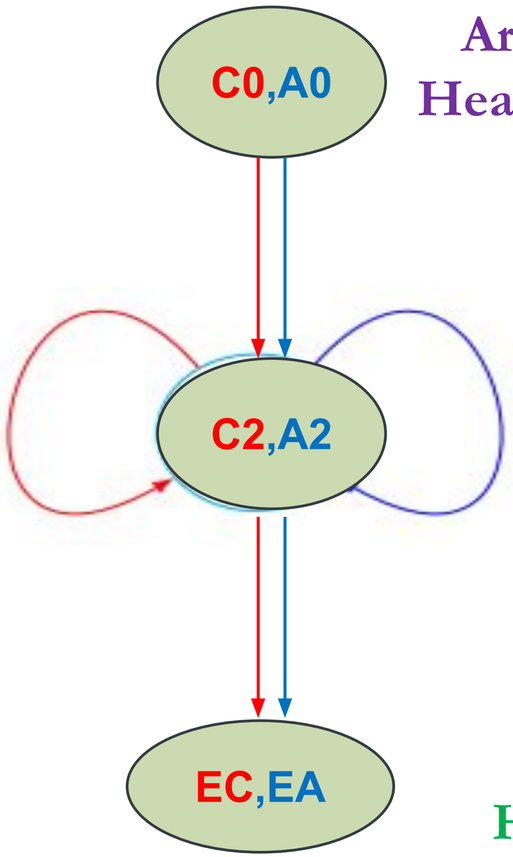


$Arg_C = Arg_A$
$Heap_C = Heap_A$

$i = r1;$
$Heap_C = Heap_A;$

$Ret_C = Ret_A$
$Heap_C = Heap_A$

Equivalence Checking



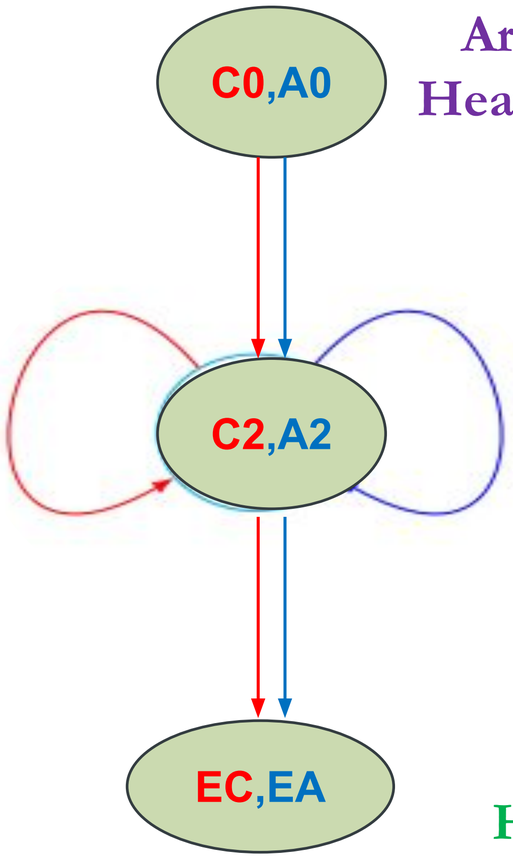
$Arg_C = Arg_A$
$Heap_C = Heap_A$

$i = r1;$
$Heap_C = Heap_A;$

$Ret_C = Ret_A$
$Heap_C = Heap_A$

Sound

Equivalence Checking



$Arg_C = Arg_A$
$Heap_C = Heap_A$

$i = r1;$
$Heap_C = Heap_A;$

$Ret_C = Ret_A$
$Heap_C = Heap_A$

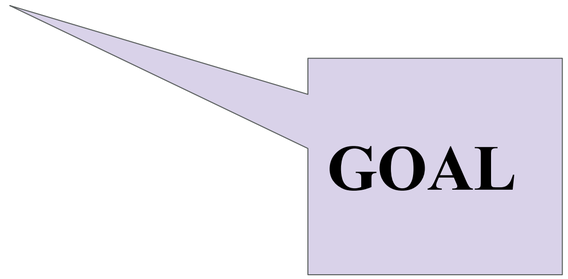
Undecidable

Identifying the correlated transitions

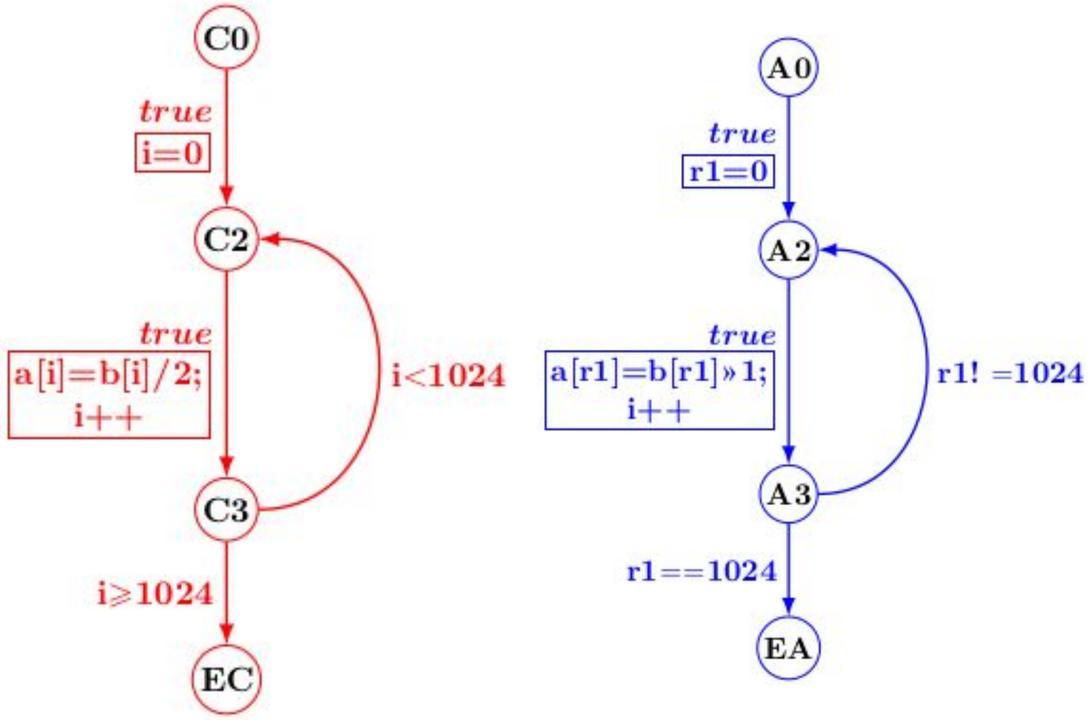
Identifying the Invariants

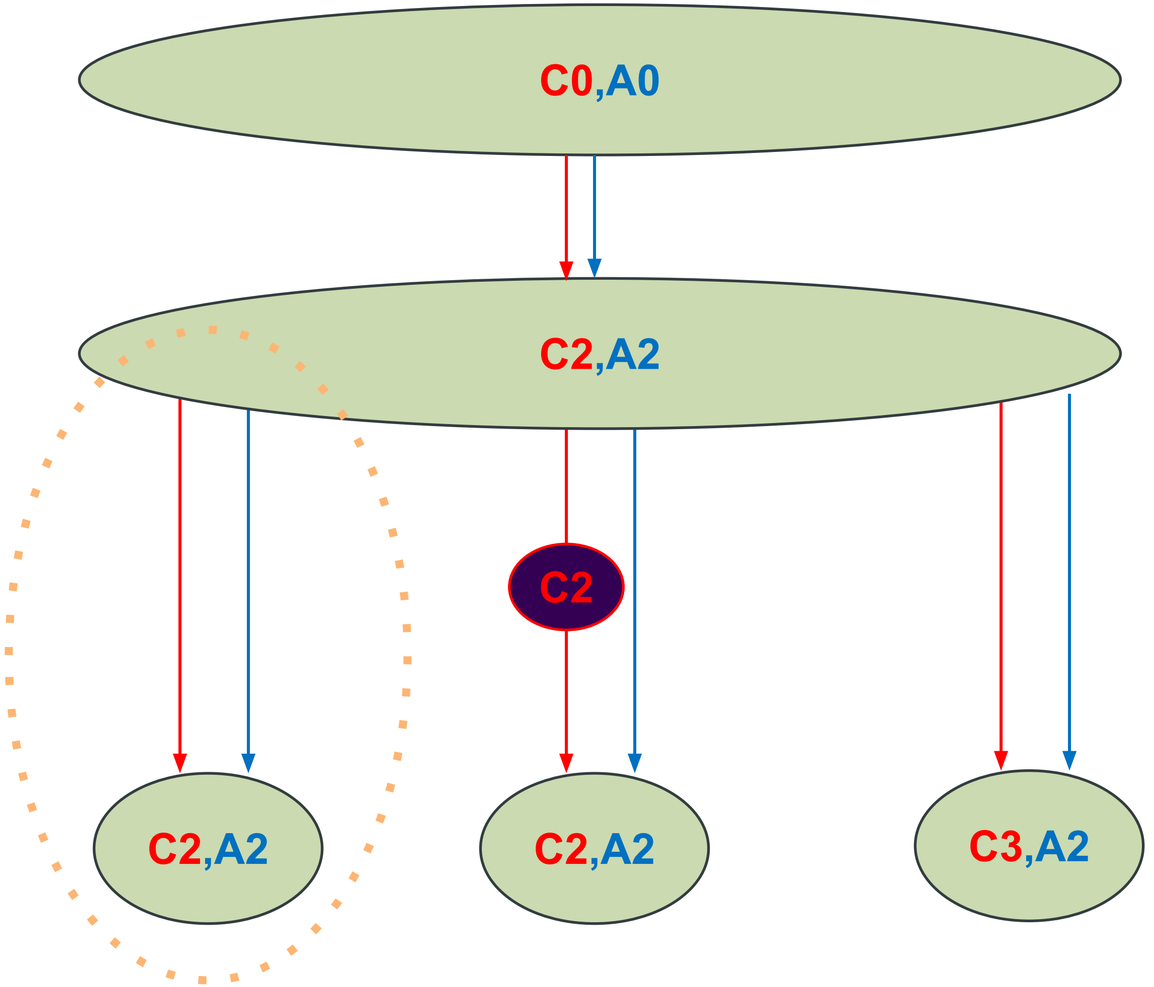
Equivalence Checking

Robustness

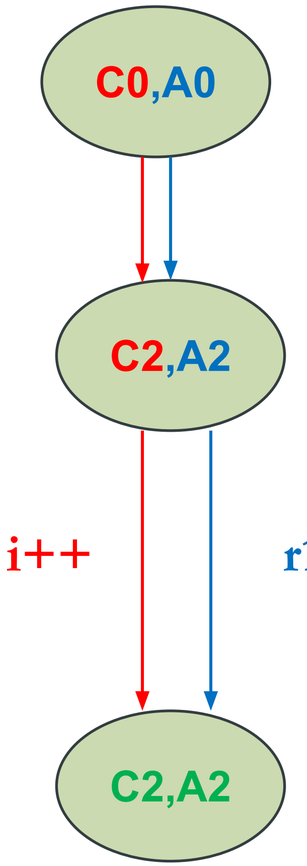


Enumerate multiple possible paths





Guessing the relations



$i=0; r1=0;$

$Heap_C = Heap_A;$

Affine relations

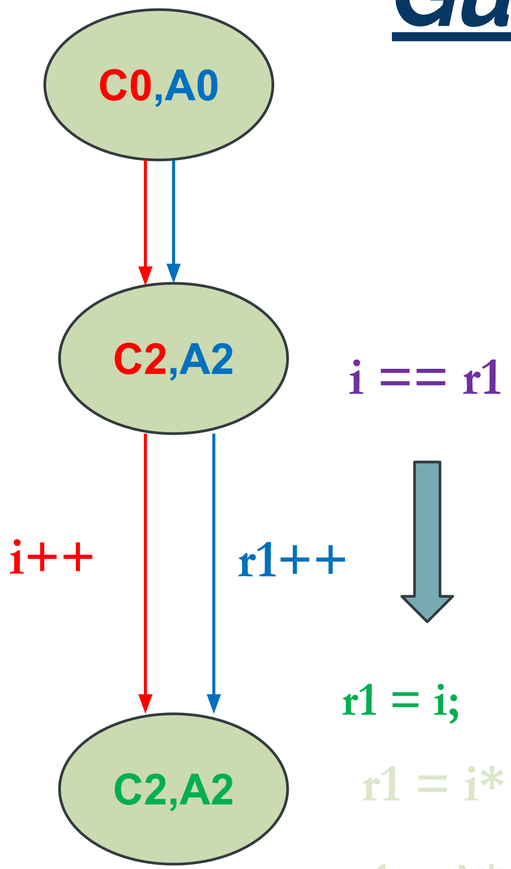
$r1 = i * 1;$ $\quad r1 + 5 = i;$ $\quad 2 * r1 = 5 * i;$

$r1 = i * 2;$ $\quad r1 = i + 3;$ $\quad 3 * r1 = 7 * i;$

$r1 = i * 100;$ $\quad r1 * 8 = i;$

$r1 = i * 240;$

Guessing the relations



i == r1

r1 = i;

r1 = i* 2;

r1 = i * 100;

r1 = i * 240;

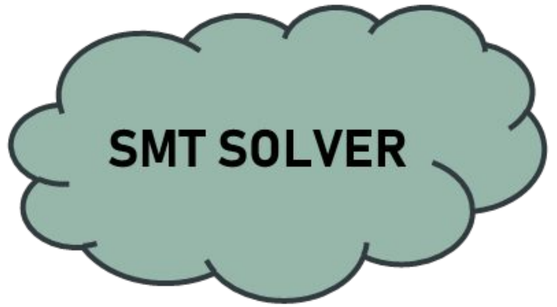
r1 + 5= i;

r1 = i + 3;

r1 * 8 = i;

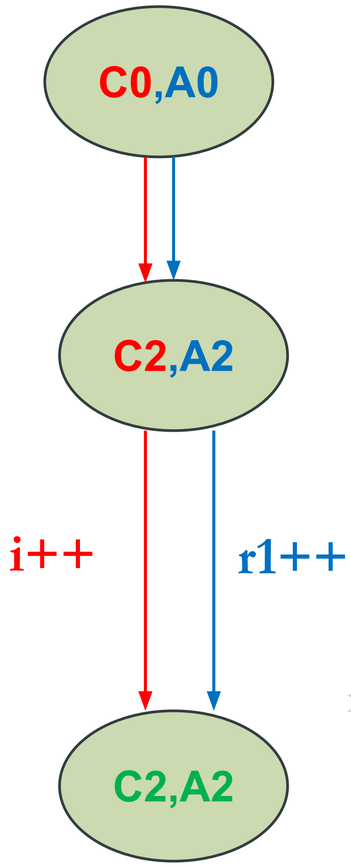
2 * r1 = 5 * i;

3 * r1 = 7 * i;



Higher-order theory

Guessing the relations



i = r1

r1 + 5= i;

r1 = i* 1;

r1 = i* 2;

r1 = i * 100;

r1 = i * 240;

r1 = i + 3;

r1 * 8 = i;

2 * r1 = 5 * i;

3 * r1 = 7 * i;

Not Provable Guess

(i == r1) => (i+1 == r1+5+1) ?

SMT SOLVER

NO

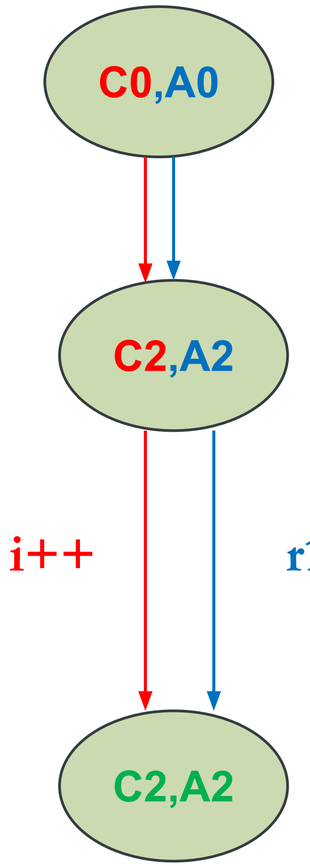
model: { i = 100; r1 = 100}

Counterexample

Research Contributions

Counterexample Guided
Invariant Inference

Invariant Inference



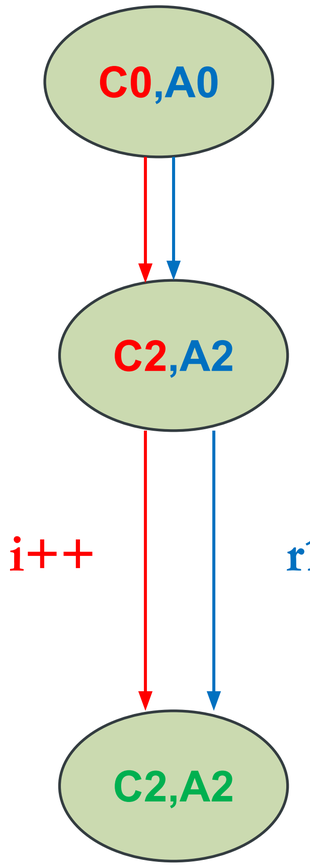
i=0; r1=0;
$Heap_C = Heap_A$;

i++

r1 +=2

Invariant : 2*i = r1

Invariant Inference



$i=0; r1=0;$
$Heap_C = Heap_A;$

i++

r1 +=2

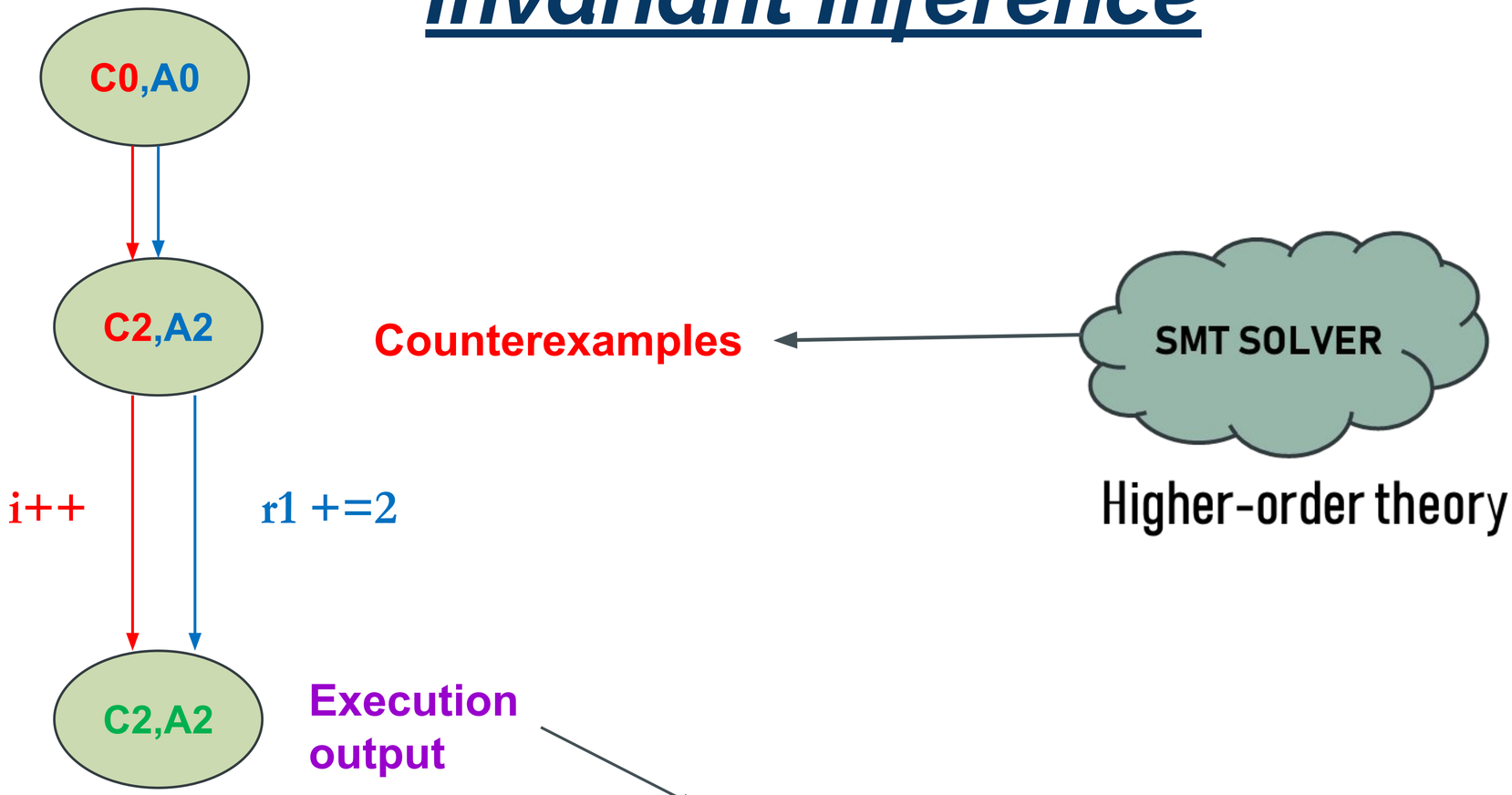
$0*i = r1$ $400*i = r1$

$1*i = r1$ $5678*i = r1$

$2*i = r1$ $2^{31}*i = r1$

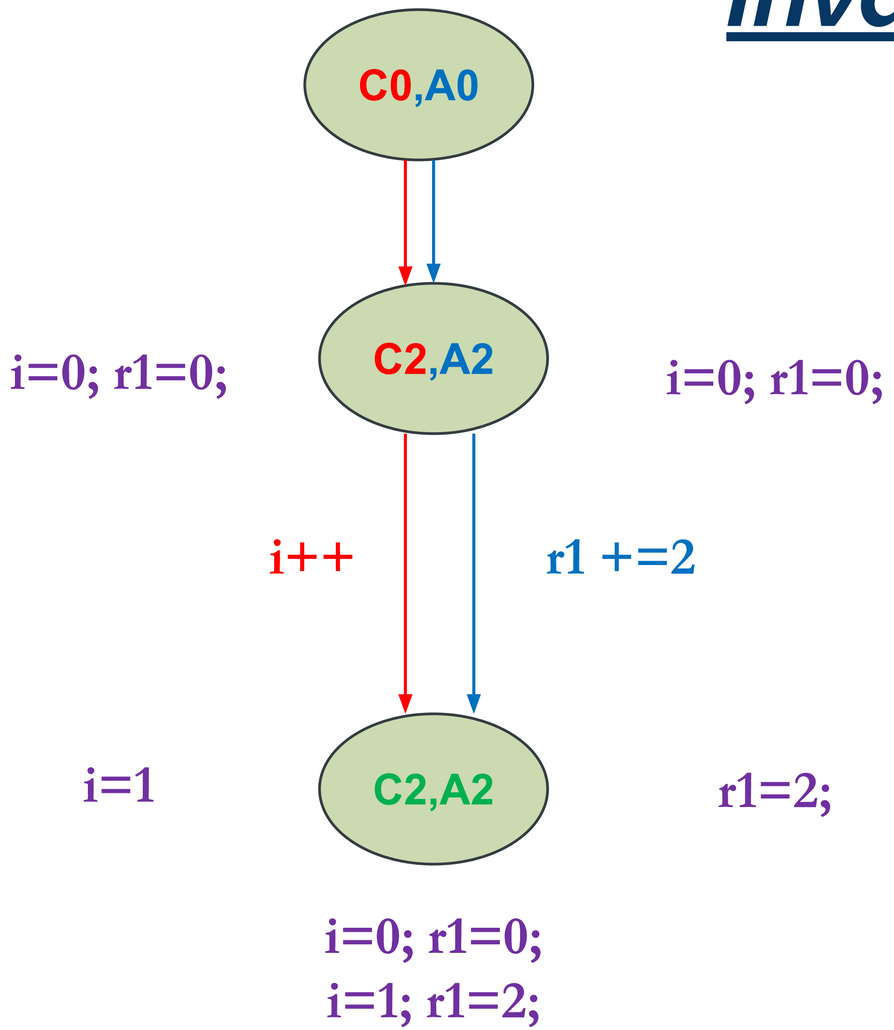
Invariant : $2*i = r1$

Invariant Inference



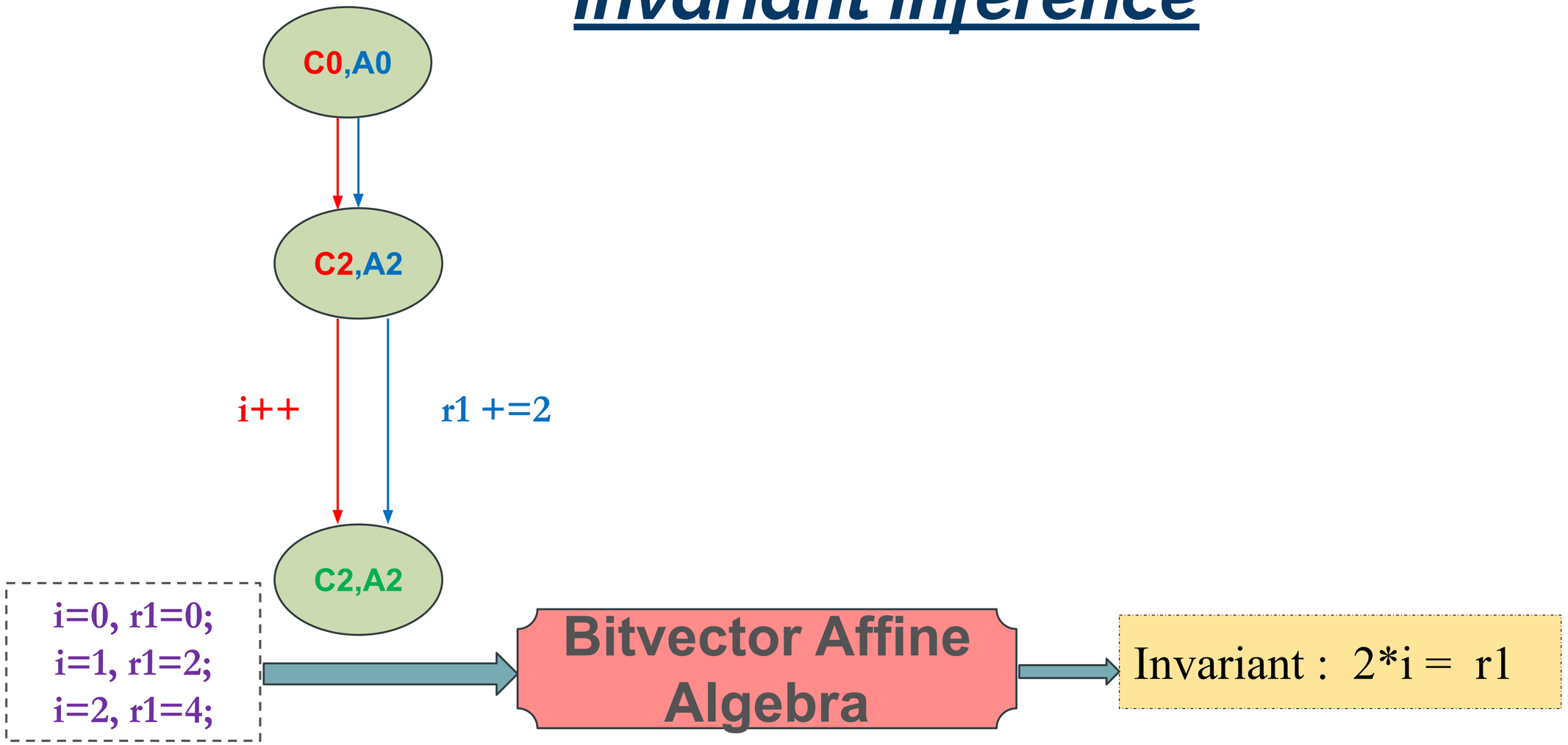
Invariant : 2*i = r1

Invariant Inference



Invariant : 2*i = r1

Invariant Inference

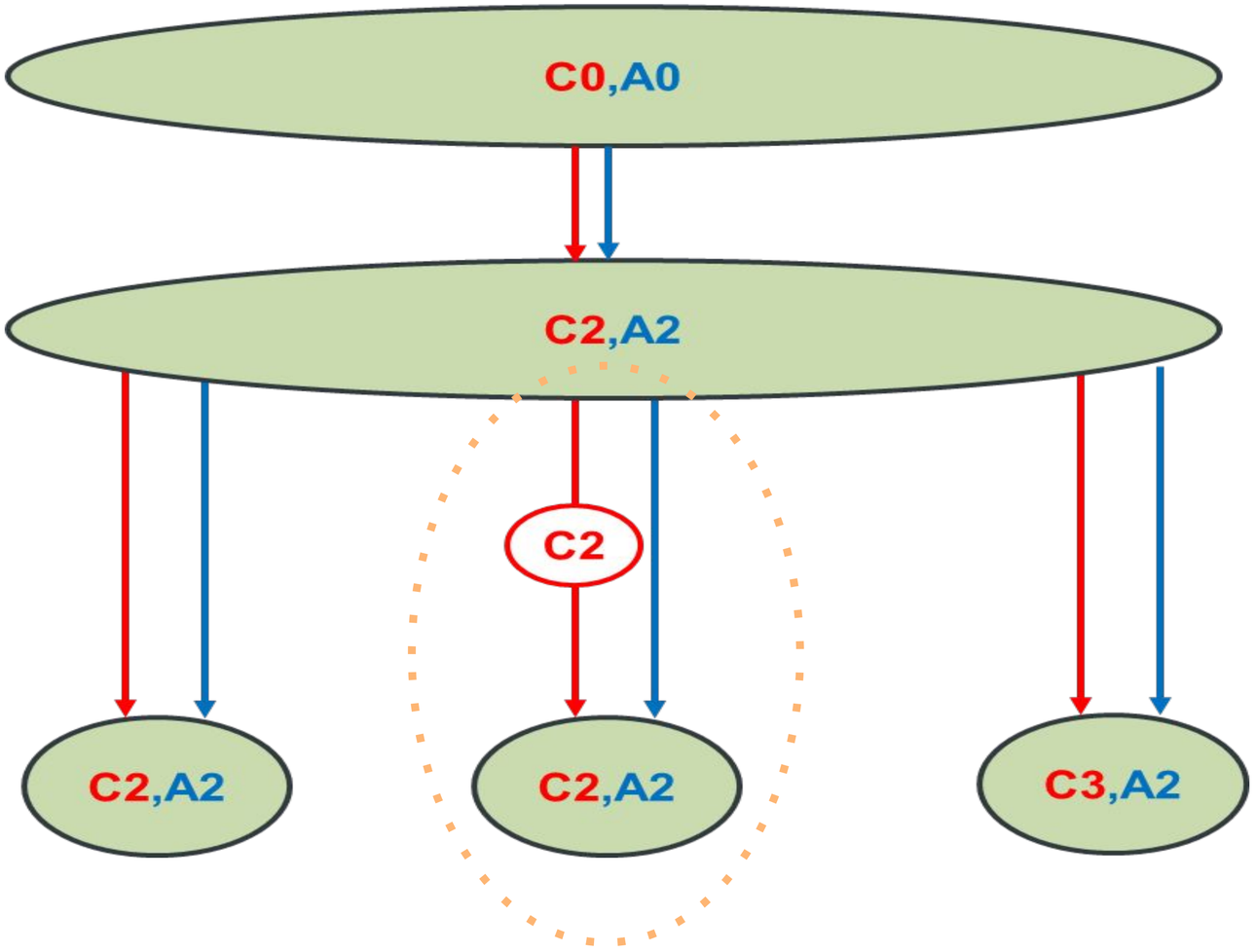


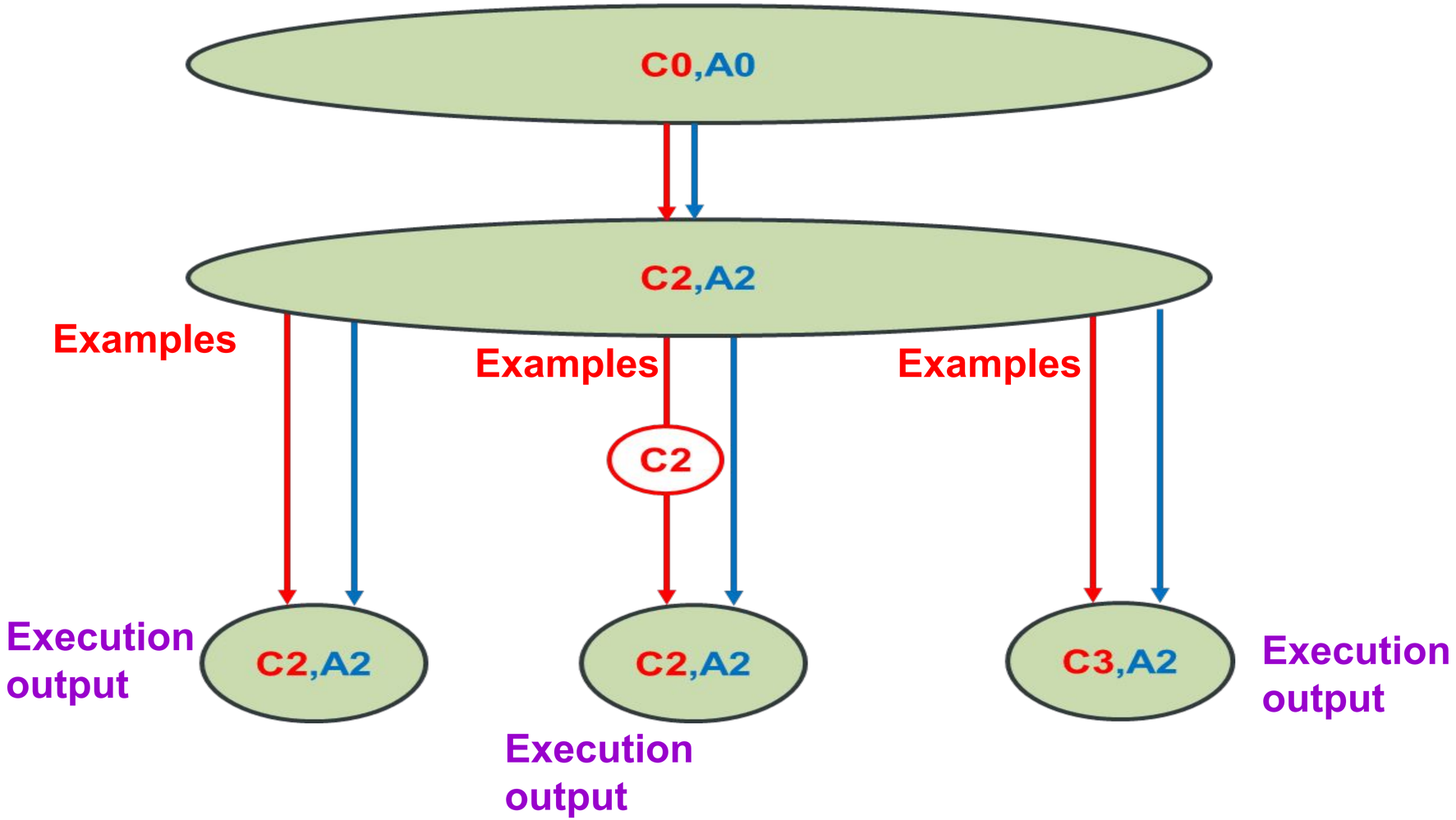
Effective use of SMT solvers for Program Equivalence Checking through Invariant-Sketching and Query-Decomposition (SAT2018)

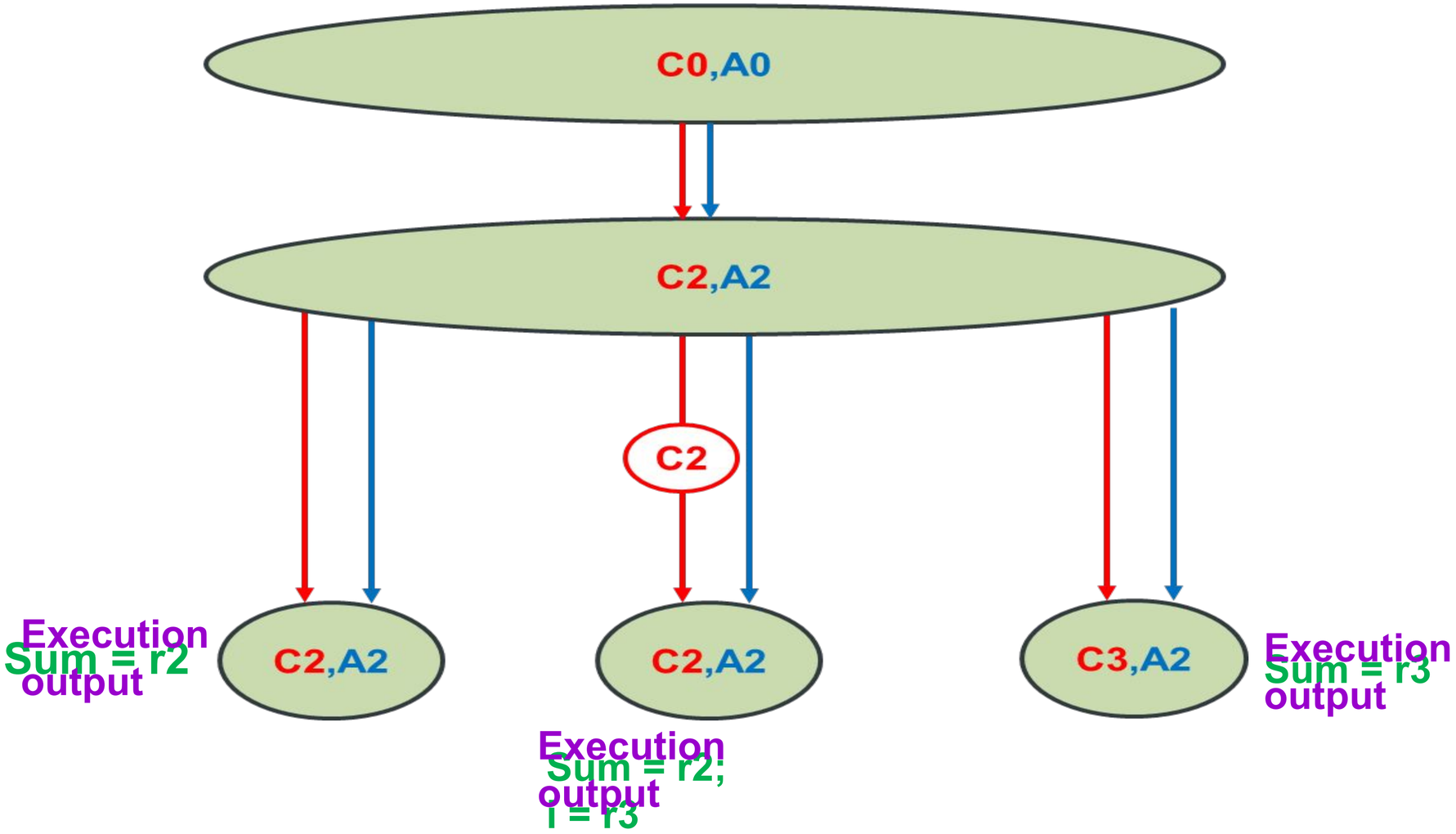
Shubhani Gupta, Aseem Saxena, Anmol Mahajan, Sorav Bansal
Indian Institute Of Technology Delhi

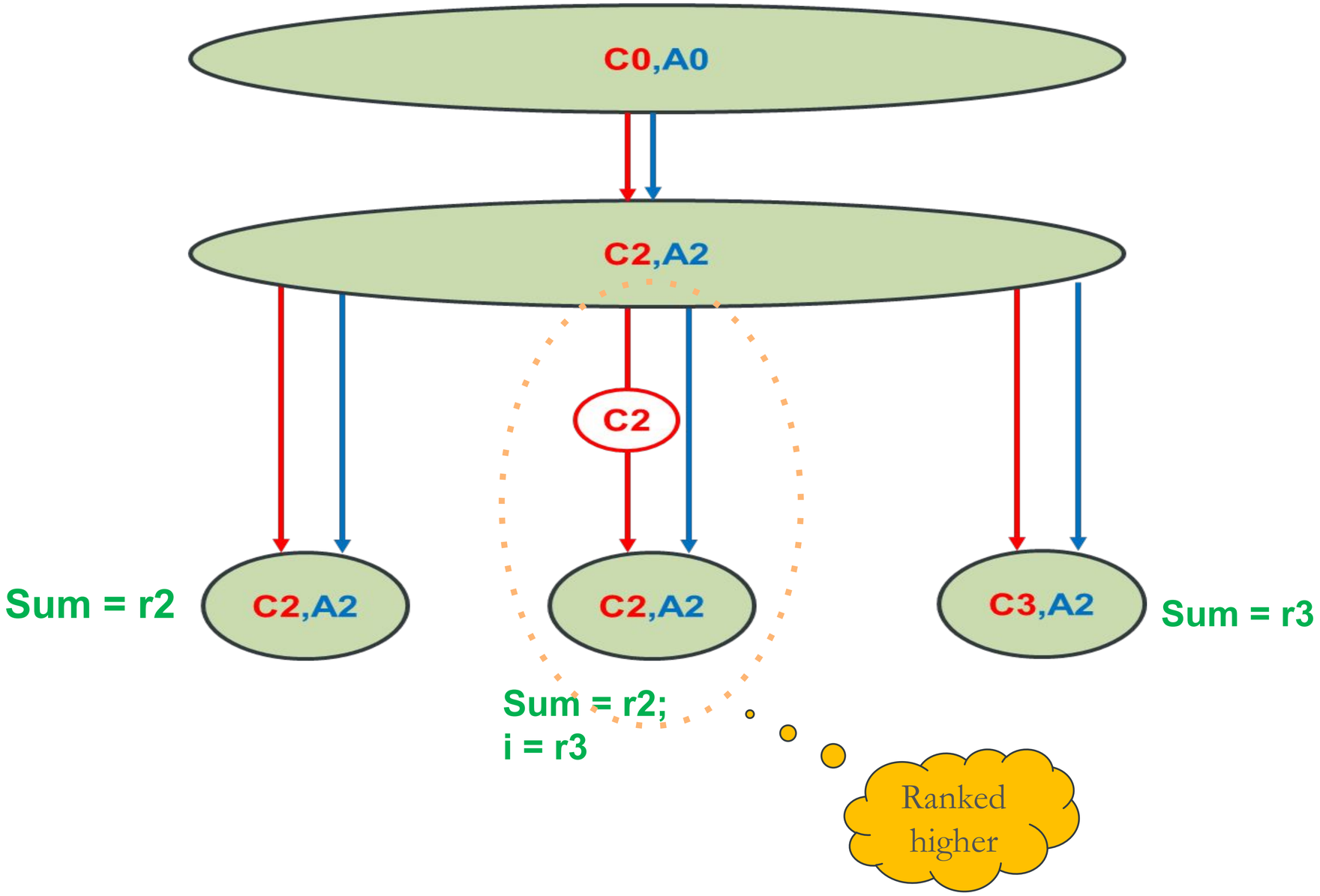
https://doi.org/10.1007/978-3-319-94144-8_22

Counterexample Guided **Correlation**


C0,A0
C2,A2
C2
C2,A2
C2,A2
C3,A2


C0,A0
C2,A2
Examples
Examples
Examples
C2
Execution output
C2,A2
C2,A2
C3,A2
Execution output
Execution output


C0,A0
C2,A2
C2
C2,A2
C2,A2
C3,A2
Execution
Sum = r2
output
Execution
Sum = r2;
output
i = r3
Execution
Sum = r3
output


C0,A0
C2,A2
C2
Sum = r2
C2,A2
C2,A2
C3,A2
Sum = r3
Sum = r2;
i = r3
Ranked
higher

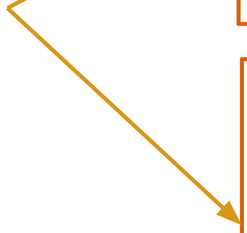
Counterexample-Guided Correlation Algorithm For Translation Validation (OOPSLA2020)

Shubhani, Abhishek Rose, Sorav Bansal
Indian Institute Of Technology Delhi

https://dl.acm.org/doi/pdf/10.1145/3428289

EXAMPLE 5 : LOOP TRANSFORMATIONS

```
#define LEN 1000
int original() {
   int sum = 0;
   int mid = LEN /2;
   for ( int i = 0; i < LEN ; i ++) {
     if ( i < mid ) sum += c[a[i]];
     if ( i >= mid ) sum += b[i];
   }
   return sum ;
}
```



```
int loopSplitting() {
   int sum = 0;
   int mid = LEN /2;
   for ( int i = 0; i < mid ; i ++) {
     if ( i < mid ) sum += c[a[i]];
     if ( i >= mid ) sum += b[i];
   }
   for ( int i = mid; i < LEN ; i ++) {
     if ( i < mid ) sum += c[a[i]];
     if ( i >= mid ) sum += b[i];
   }
   return sum ;
}
```

EXAMPLE 5 : LOOP TRANSFORMATIONS

```
int loopSplitting() {
   int sum = 0;
   int mid = LEN /2;
   for ( int i = 0; i < mid ; i ++) {
     if ( i < mid ) sum += c[a[i]];
     if ( i >= mid ) sum += b[i];
   }
   for ( int i = mid; i < LEN ; i ++) {
     if ( i < mid ) sum += c [a[i]];
     if ( i >= mid ) sum += b[i];
   }
   return sum ;
}
```

```
int loopUnswitching() {
   int sum = 0;
   int mid = LEN /2;
   for ( int i = 0; i < mid ; i++) {
     sum += c[a[i]];
   }
   for ( int i = mid; i < LEN ; i ++) {
     sum += b[i];
   }
   return sum ;
}
```

EXAMPLE 5 : LOOP TRANSFORMATIONS

```
int loopUnswitching() {
    int sum = 0;
    int mid = LEN /2;
    for ( int i = 0; i < mid ; i++) {
      sum += c[a[i]];
    }
    for ( int i = mid; i < LEN ; i ++) {
      sum += b[i];
    }
    return sum ;
}
```

```
int loopUnrolling() {
    int sum = 0;
    int mid = LEN /2;
    for ( int i = 0; i < mid ; i ++) {
      sum += c[a[i]];
    }
    for ( int i = mid; i < LEN ; i +=4) {
      sum += b[ i ];
      sum += b[ i+1 ];
      sum += b[ i +2];
      sum += b[ i +3];
    }
    return sum ;
}
```

EXAMPLE 5 : LOOP TRANSFORMATIONS

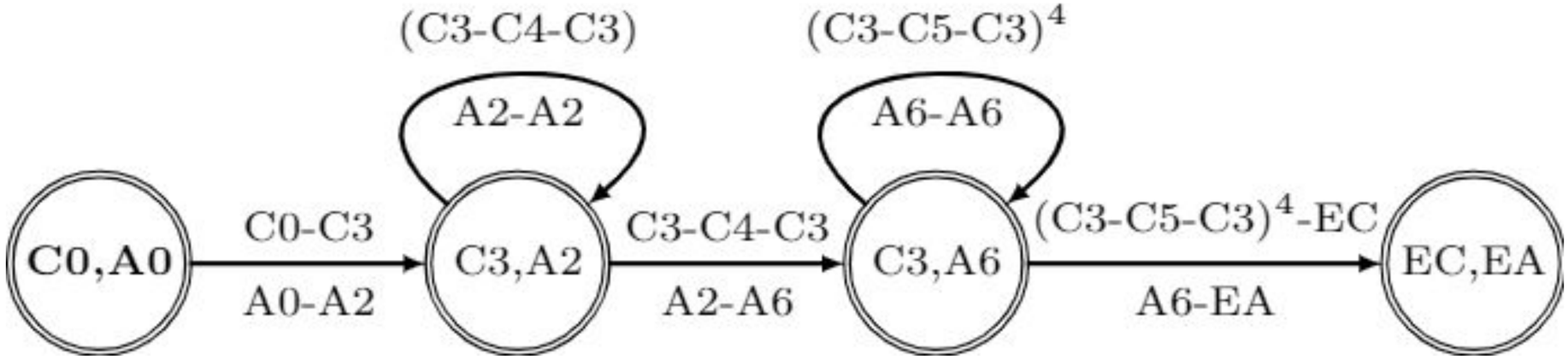
```
int loopUnrolling() {
   int sum = 0;
   int mid = LEN /2;
   for ( int i = 0; i < mid ; i ++) {
     sum += c[a[i]];
   }
   for ( int i = mid; i < LEN ; i +=4) {
     sum += b[ i ];
     sum += b[ i+1 ];
     sum += b[ i +2];
     sum += b[ i +3];
   }
   return sum ;
}
```

```
A0 : loopVectorizedAndRegAllocated :
A1 :   r1 = 0; r2 = 0;
A2 :     r2 += c [ a [ r1 ]]
A3 :     r1 ++
A4 :     if ( r1 != mid ) goto A2
A5 :   r1 = &b[mid]; r3=& b[LEN]; xmm0 = 0
A6 :     xmm0 += * r1 , .. , *( r1 +12)
A7 :     r1 += 16
A8 :     if ( r1 != r3 ) goto A6
A9 :   xmm0 += ( xmm0 >> 8)
A10 :  xmm0 += ( xmm0 >> 4)
A11 :  r2 += xmm0 [31:0]
EA   :  ret r2
```

End-to-End Equivalence Check

```
  #define LEN 1000
C0: int original() {
C1:   int sum = 0;
C2:   int mid = LEN /2;
C3:   for ( int i = 0; i < LEN ; i ++) {
C4:     if ( i < mid ) sum += c[a[ i ]];
C5:     if ( i >= mid ) sum += b[i];
C6:   }
EC:   return sum ;
  }
```

```
A0 : loopVectorizedAndRegAllocated :
A1 :   r1 = 0; r2 = 0;
A2 :     r2 += c [ a [ r1 ]]
A3 :     r1 ++
A4 :     if ( r1 != mid ) goto A2
A5 :   r1 = &b[mid]; r3=& b[LEN]; xmm0 = 0
A6 :     xmm0 += * r1 , .. , *( r1 +12)
A7 :     r1 += 16
A8 :     if ( r1 != r3 ) goto A6
A9 :   xmm0 += ( xmm0 >> 8)
A10 :  xmm0 += ( xmm0 >> 4)
A11 :  r2 += xmm0 [31:0]
EA   :  ret r2
```

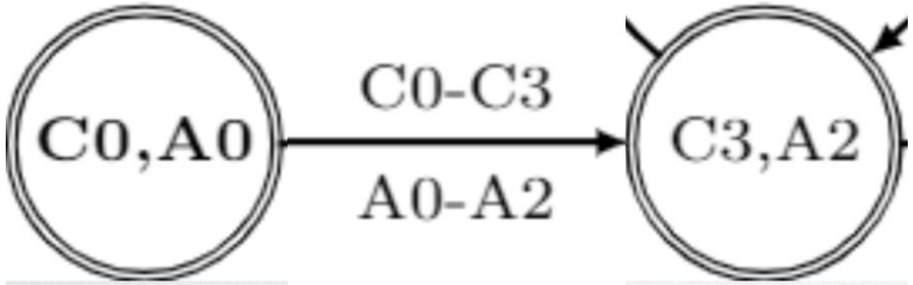


Incremental Construction of the Product CFG

Incremental Construction of the Product CFG

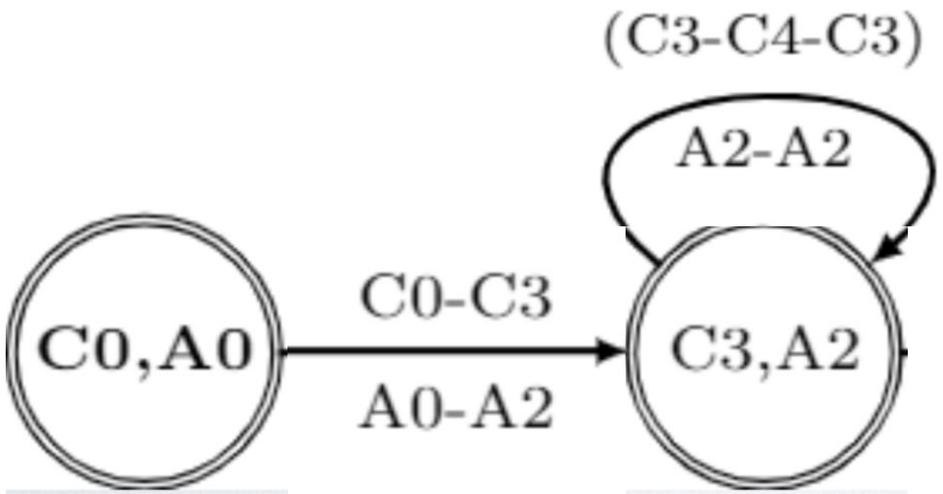
Incremental Construction of the Product CFG

Use off-the-shelf invariant inference algorithms to infer affine, equality and inequality invariants on bitvectors and memory states



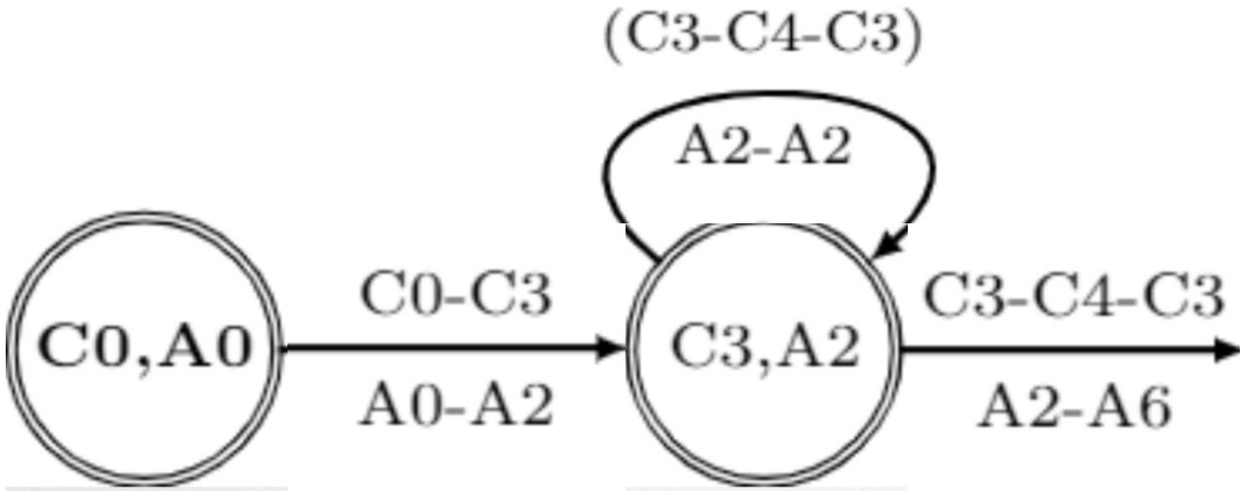
Infer Invariants at (C3,A2)

Incremental Construction of the Product CFG

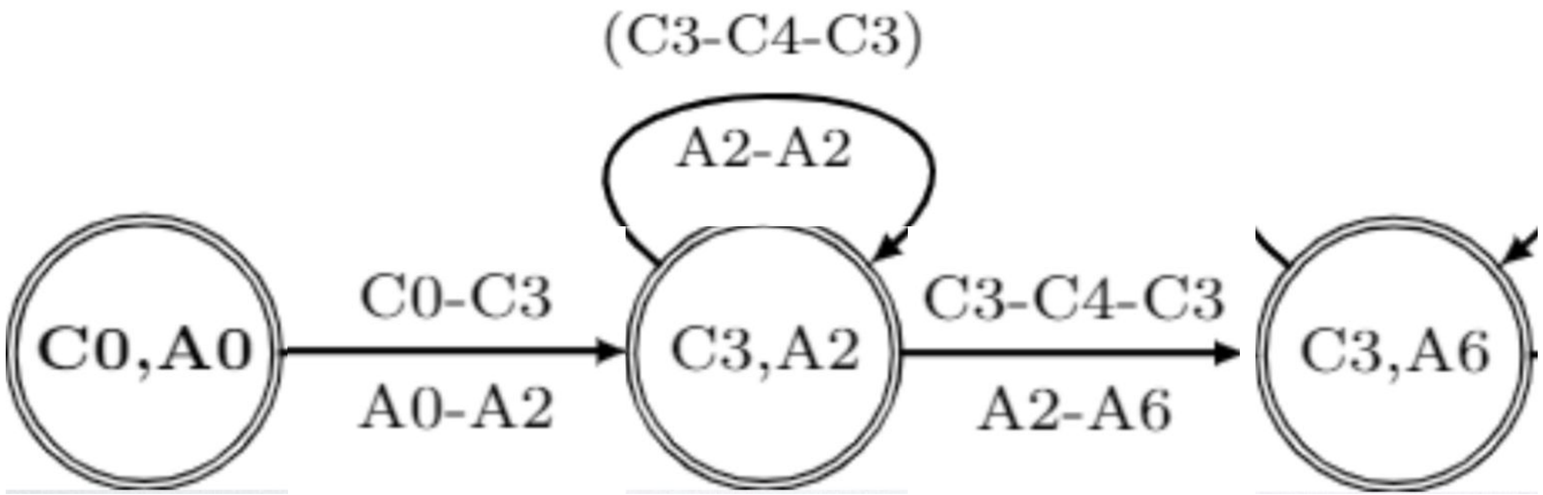


Relax Invariants
at (C3,A2)

Incremental Construction of the Product CFG

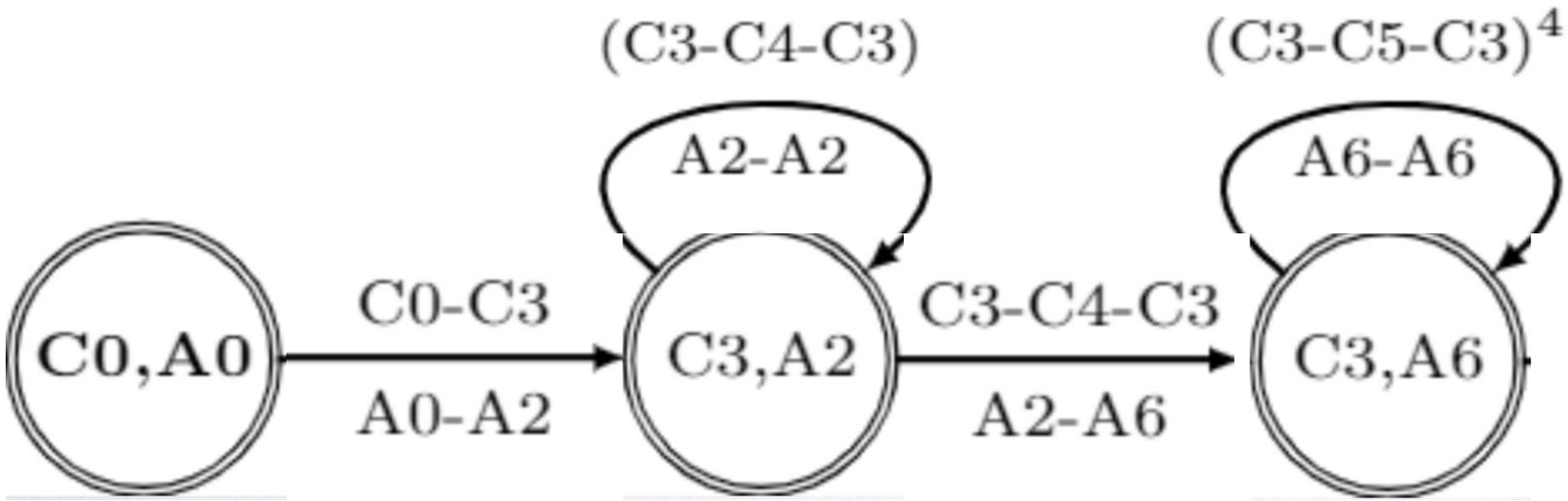


Incremental Construction of the Product CFG



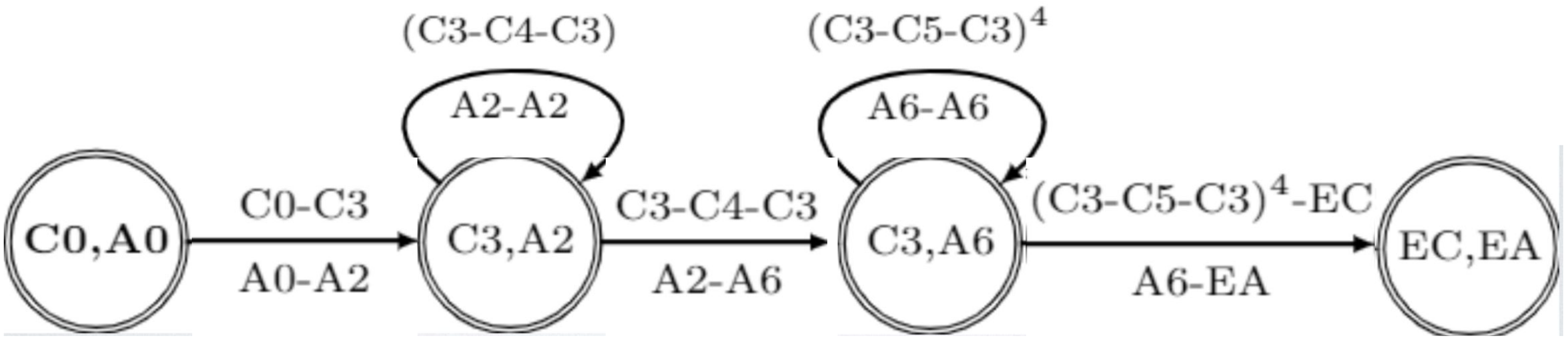
Infer Invariants at
(C3,A6)

Incremental Construction of the Product CFG



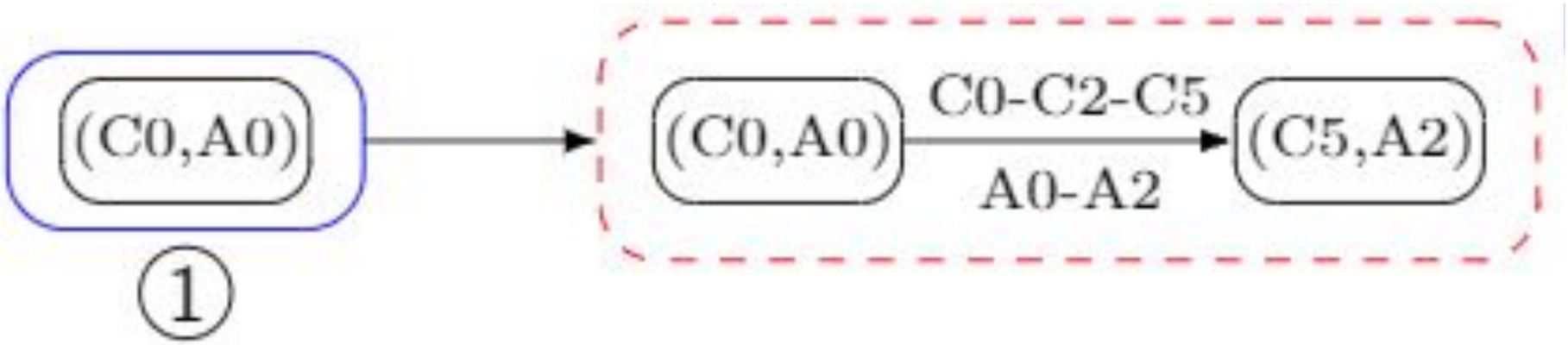
Relax Invariants
at (C3,A6)

Incremental Construction of the Product CFG

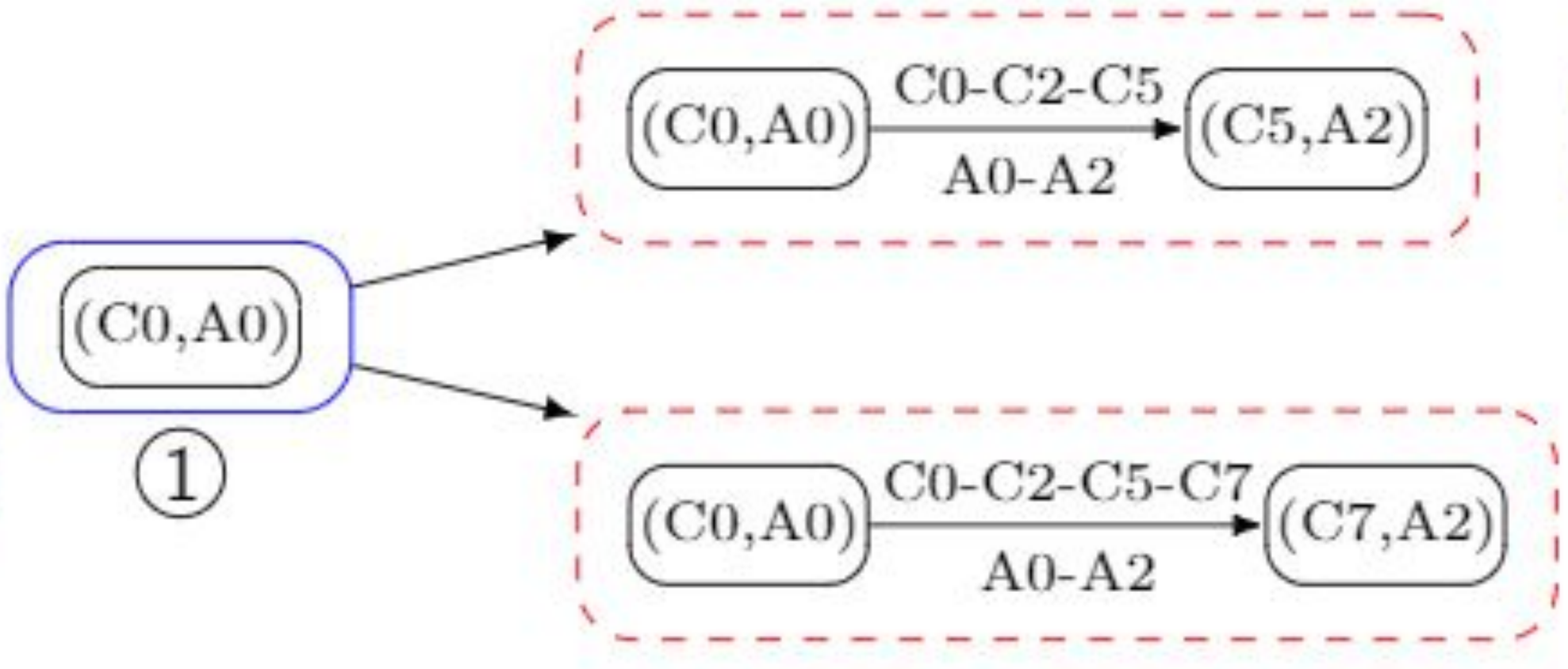


Check equivalence of return values under inferred invariants

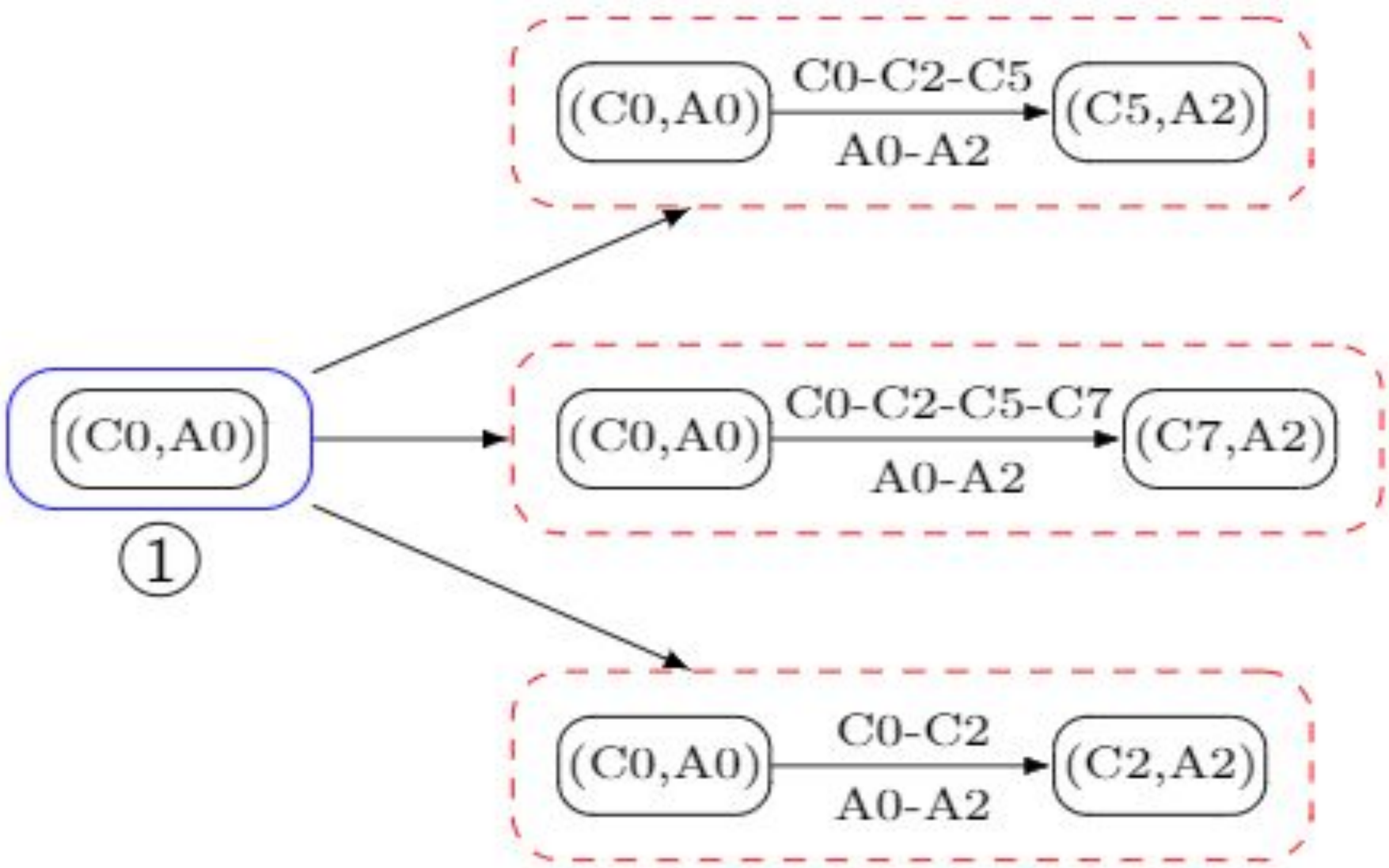
SEARCH SPACE



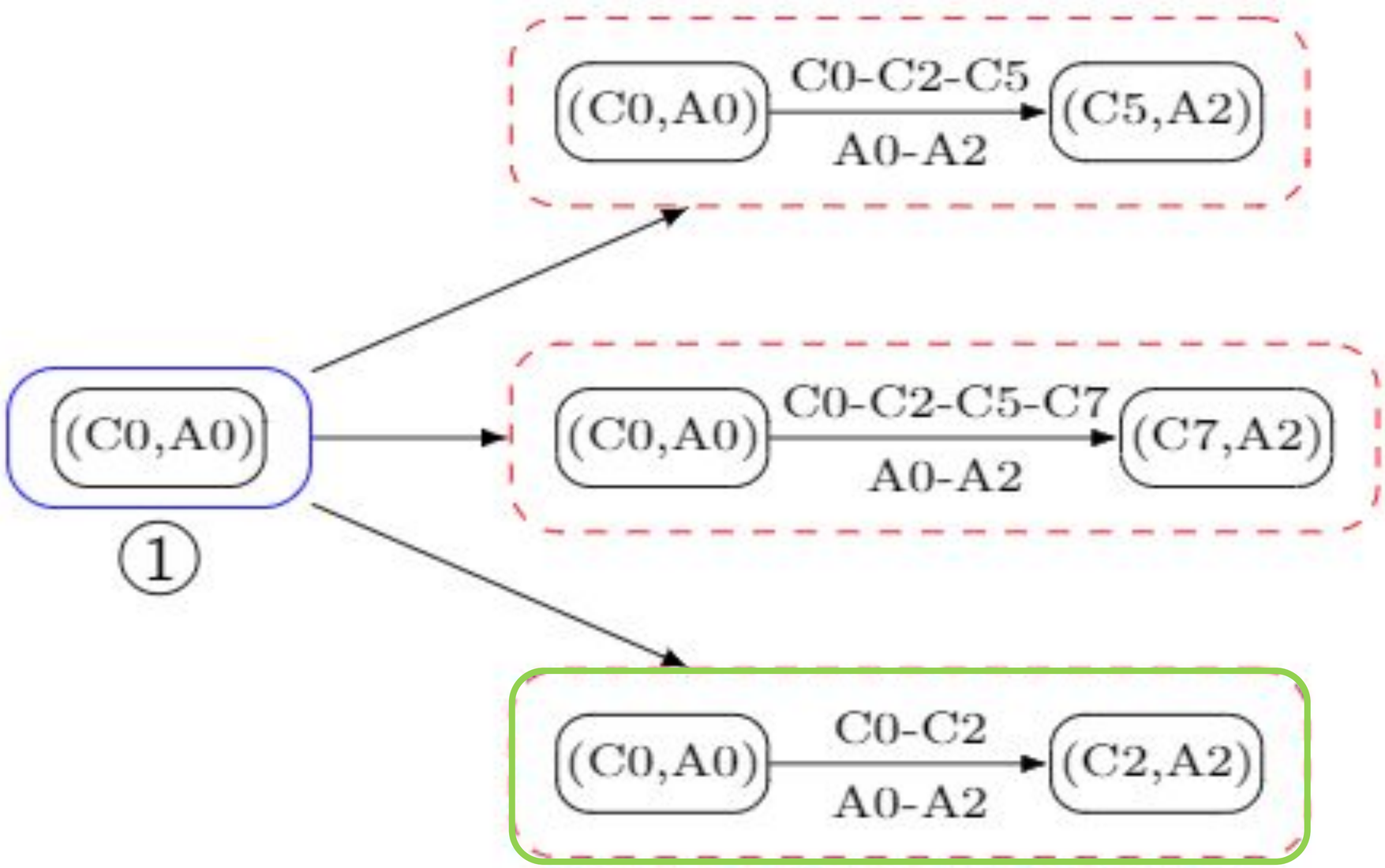
SEARCH SPACE



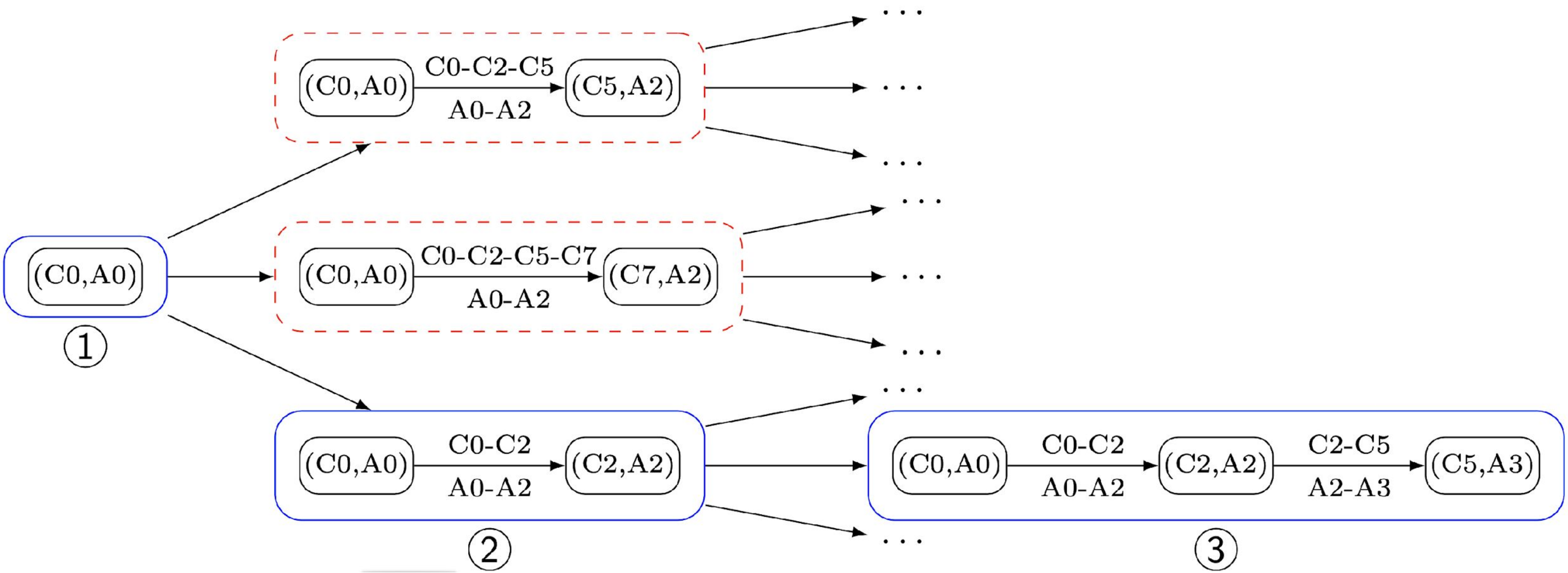
SEARCH SPACE



SEARCH SPACE



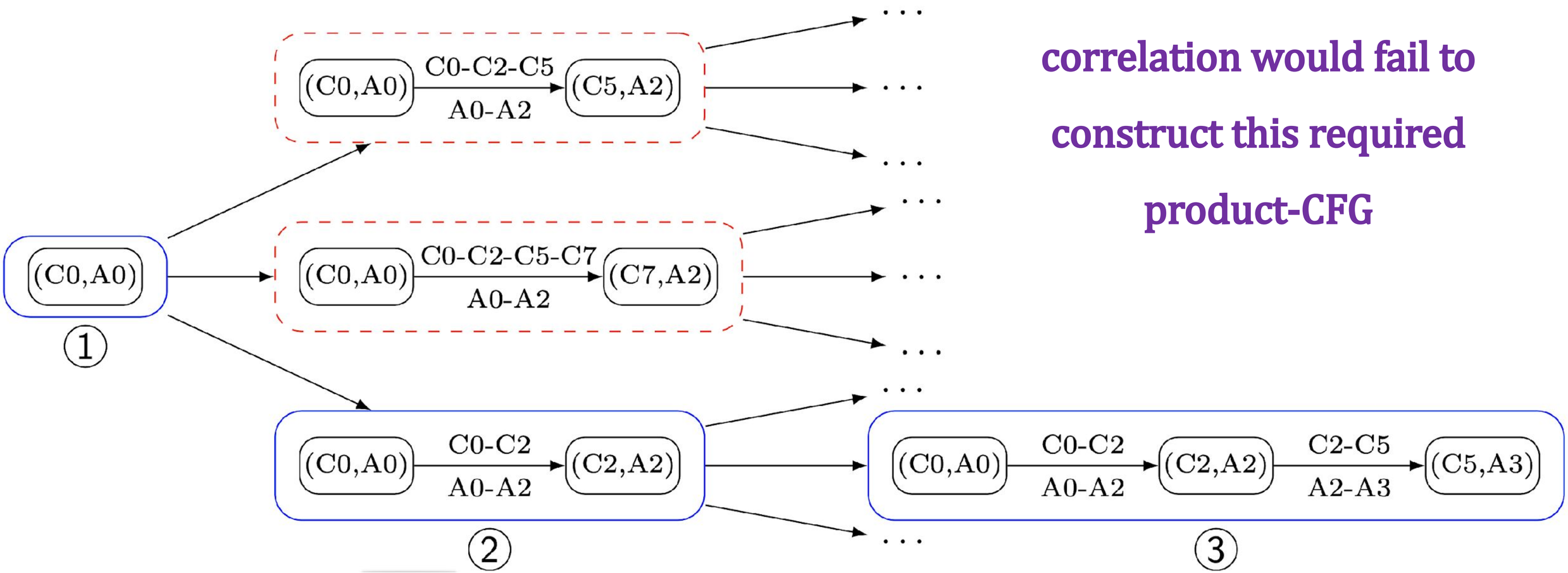
SEARCH SPACE



Exhaustive search would take millions of years to compute equivalence

SEARCH SPACE

Prior work on data driven correlation would fail to construct this required product-CFG

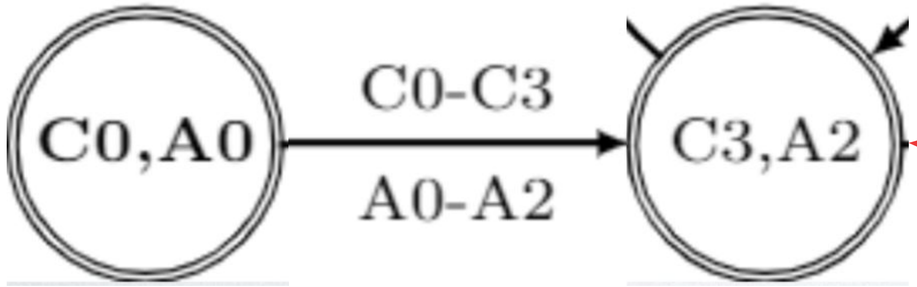


Exhaustive search would take millions of years to compute equivalence

Counterexamples

During invariant inference, we make potential GUESSes for invariants. We try to prove a GUESS using an SMT Solver.

- If the GUESS is provable, we have found an invariant.
- If not, the SMT solver returns a counterexample



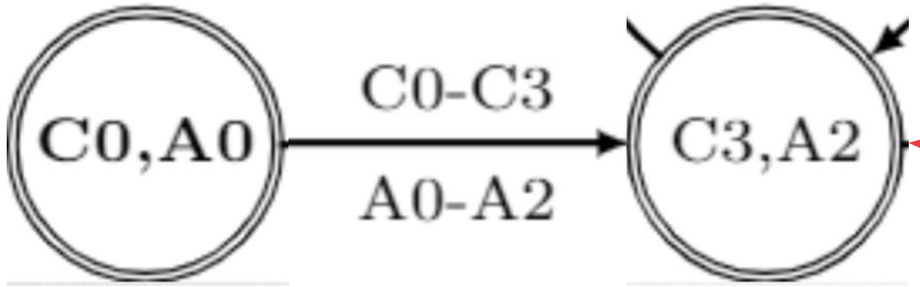
Counterexamples identified at (C3,A2) during invariant inference

Infer Invariants at (C3,A2)

Counterexamples

A counterexample at a node is a potential concrete machine state that may occur at that particular node during execution.

The concrete state would involve valuations for (related) variables of both C and A.

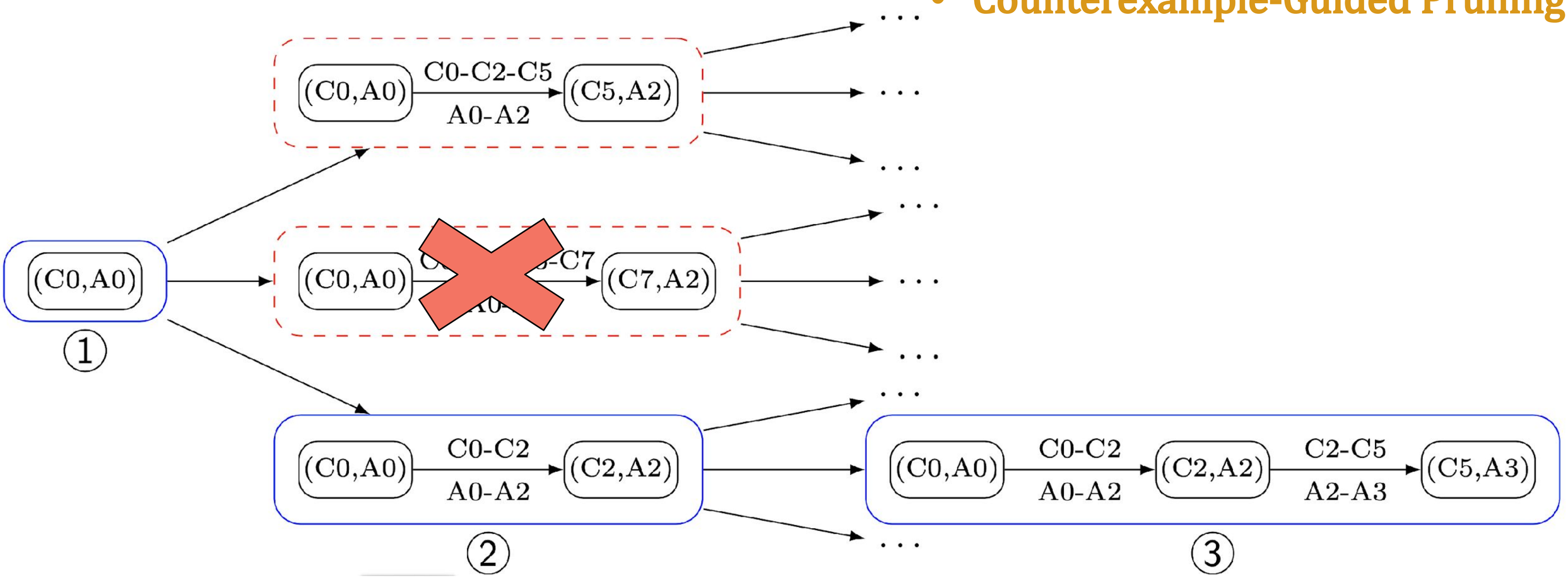


Counterexample1: I=5, R1=5
Counterexample2: I=1003, R1=1003

Infer Invariants at (C3,A2)

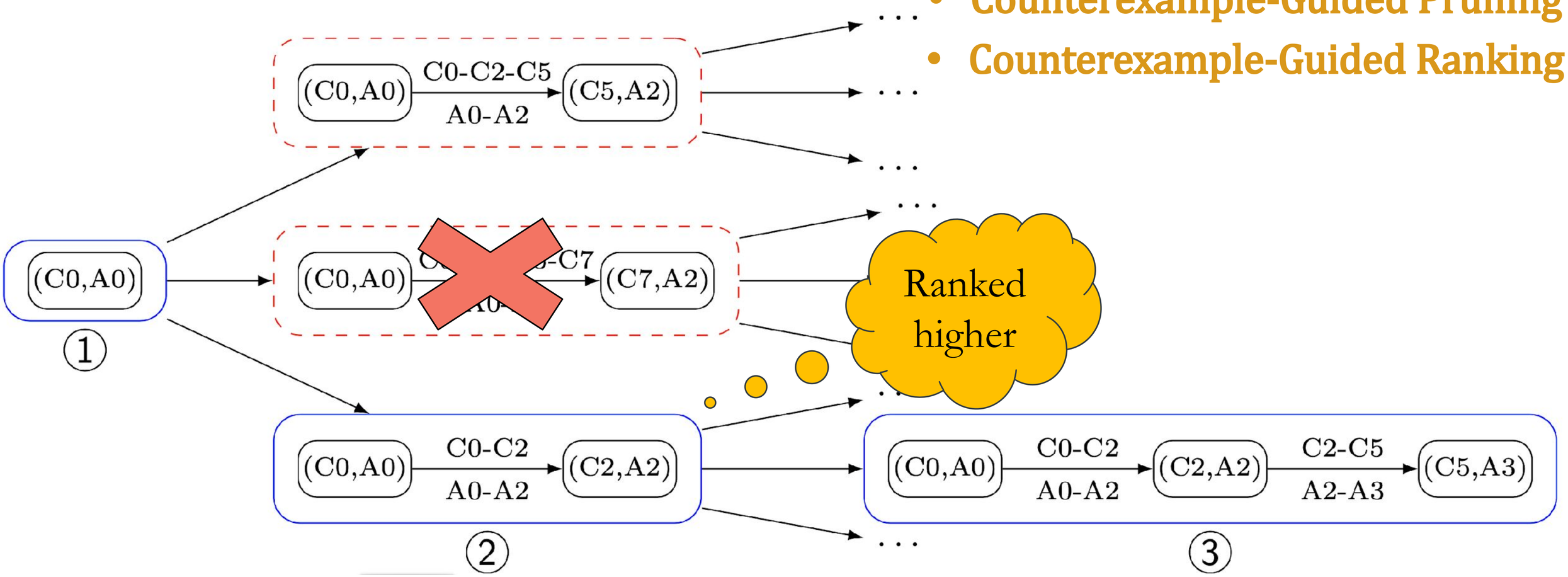
COUNTEREXAMPLE GUIDED BEST-FIRST SEARCH

- **Counterexample-Guided Pruning**

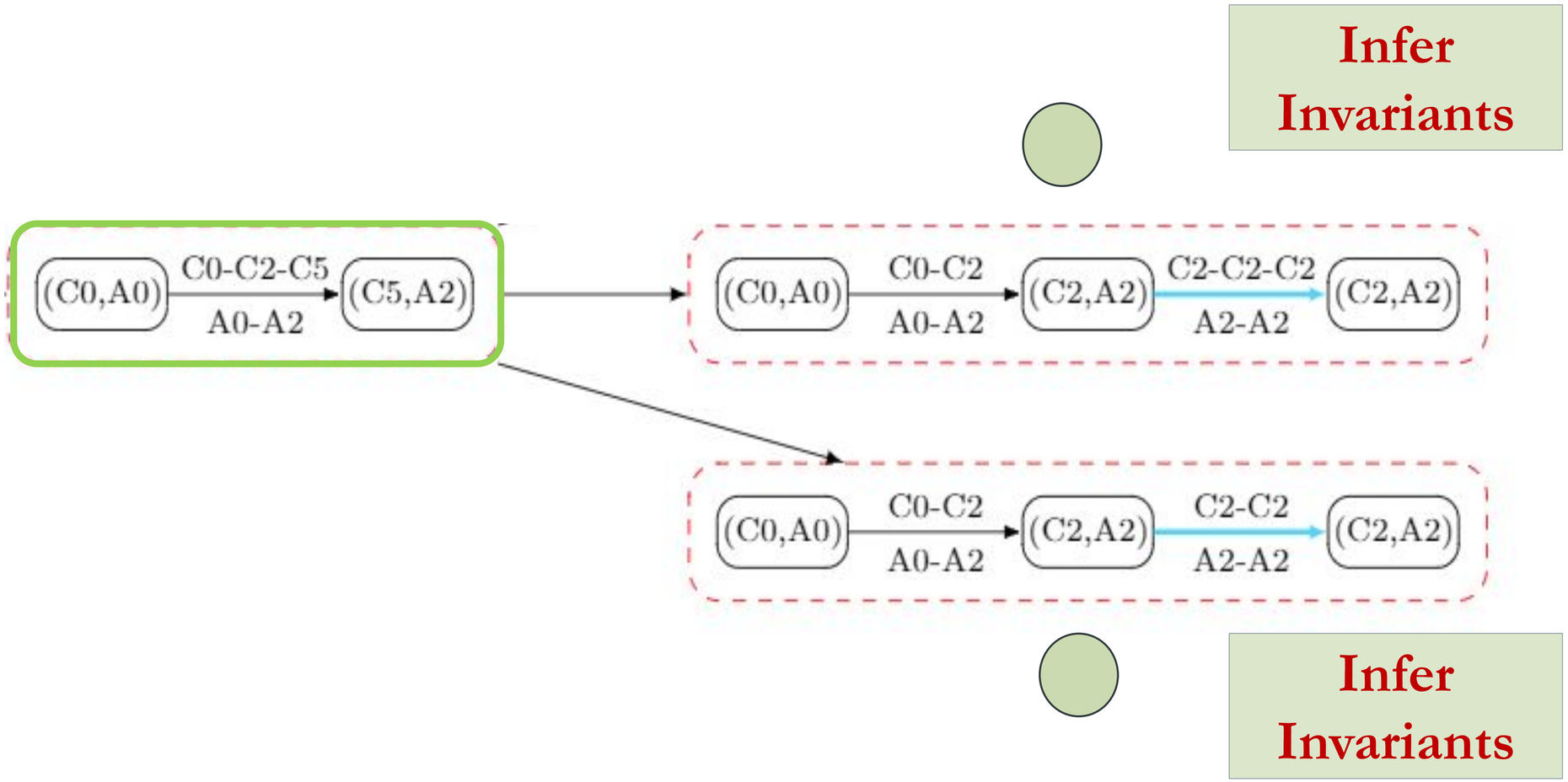


COUNTEREXAMPLE GUIDED BEST-FIRST SEARCH

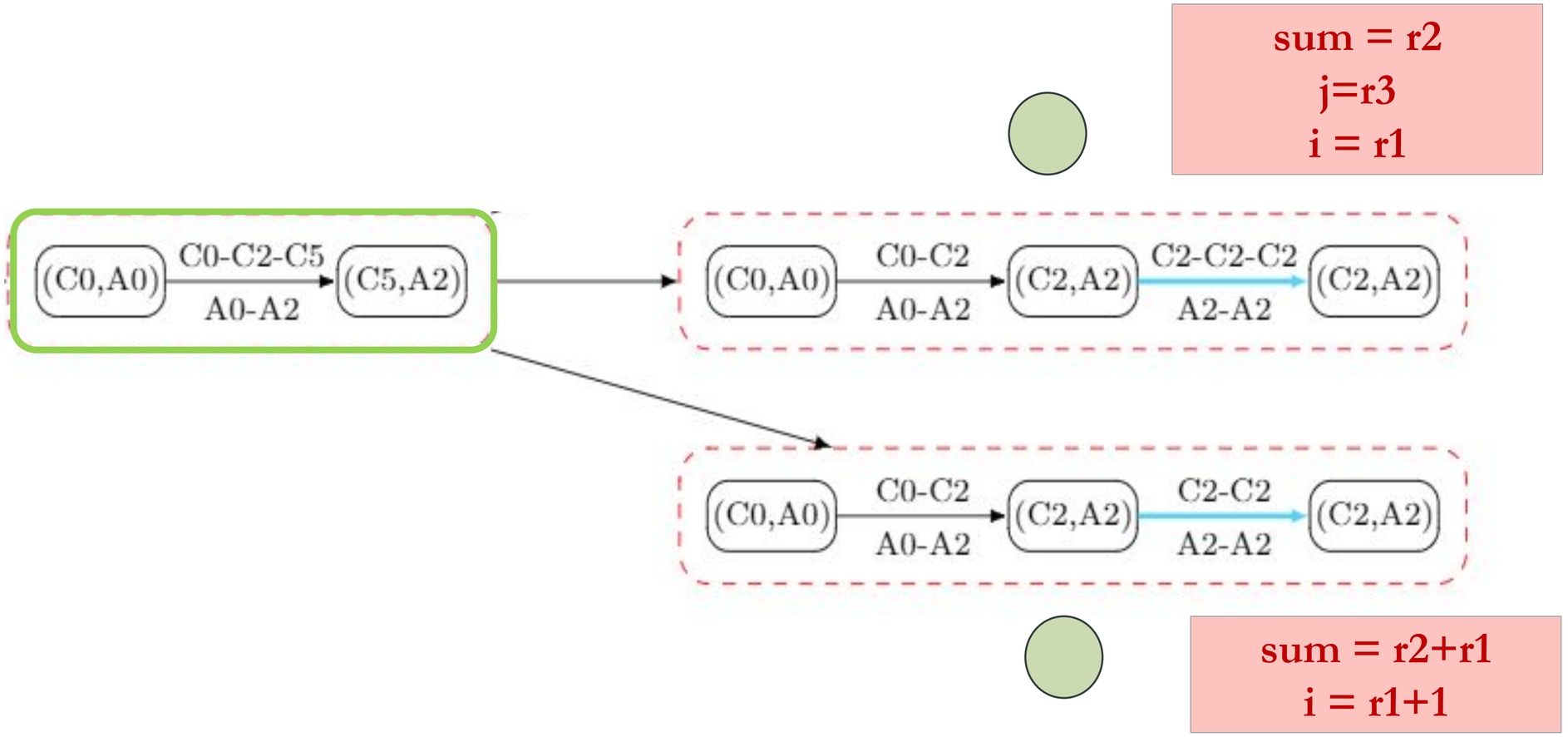
- **Counterexample-Guided Pruning**
- **Counterexample-Guided Ranking**



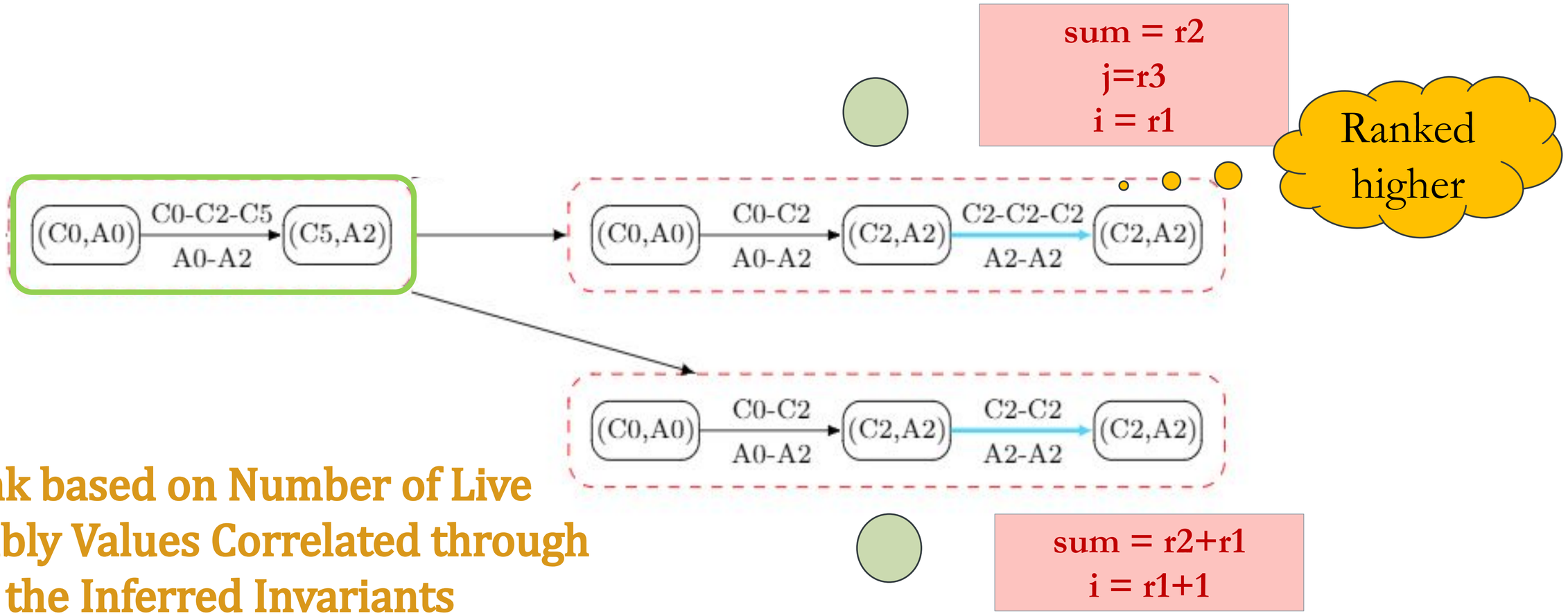
Infer Invariant Covers for Executed Counterexamples



Infer Invariant Covers for Executed Counterexamples

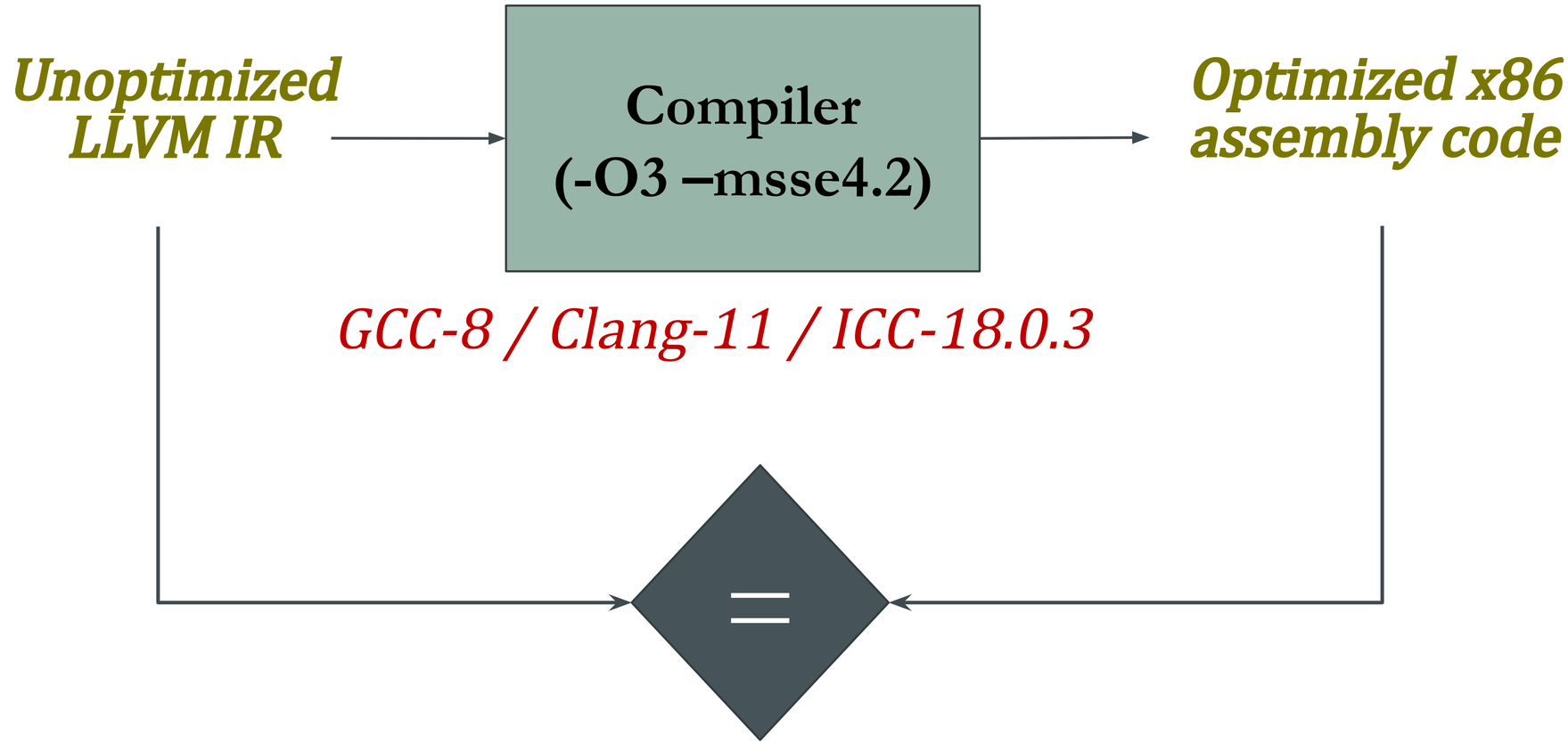


Infer Invariant Covers for Executed Counterexamples



Rank based on Number of Live Assembly Values Correlated through the Inferred Invariants

Counter Evaluation



Evaluated on Testsuite for Vectorizing Compilers

Bugs Discovered (https://compiler.ai/bugs))

- Bug in ICC-16.03 involving integer overflow
- Bug in ICC-16.03 related to incorrect reordering of memory accesses
- Bug in GCC-4.8 involving incorrect reordering of memory accesses
- Bug in Qemu machine emulator that is shipped with Linux/KVM hypervisor
- Several bugs in DietLibc related to missing unsigned-to-signed typecasts
- Bug in the Yices SMT Solver related to incorrect query result
- Bug in strrchr() function in klibc-2.0.11 related to missing handling of c == 0 (kernel)
- Bug in swab() function in NetBSD's C library (3290fbc). Fix on Github
- Bug in the memccpy() function in Newlib-4.2.0 (embedded systems C library)

Automatic Generation of Debug Headers

through Blackbox Equivalence Checking

An Example Debug Session

```
#define LEN 32000
int X[LEN] , Y[LEN], val;
C0:  void addAndCopy (  ) {
C1:      for (int i=0; i < LEN; i++) {
C2:          X[i] = Y[i] +  val;
C3:      }
EC:  }
```

C Program

An Example Debugging Session

```
#define LEN 32000
int X[LEN] , Y[LEN], val;
C0:  void addC( ) {
C1:      for (int i=0; i < LEN; i++) {
C2:          X[i] = Y[i] +  val;
C3:      }
EC:  }
```

C Program

```
(gdb) break addC:C2

(gdb) run
Starting program: addC
Breakpoint 1, addC () at addC.c:C3
            X[i] = Y[i] + val;
(gdb) print i
$1 = 0
(gdb) continue
Continuing.
Breakpoint 1, addC () at addC.c:C3
            X[i] = Y[i] + val;
(gdb) print i
$2 = 0
```

i appears to be always 0

An Example Debugging Session

```
#define LEN 32000
int X[LEN] , Y[LEN], val;
C0:  void addC( ) {
C1:      for (int i=0; i < LEN; i++) {
C2:          X[i] = Y[i] +  val;
C3:      }
EC:  }
```

C Program

```
(gdb) break addC:C2

(gdb) run
Starting program: addC
Breakpoint 1, addC () at addC.c:C3
                X[i] = Y[i] + val;
(gdb) print i
<value optimized out>
(gdb) continue
Continuing.
Breakpoint 1, addC () at addC.c:C3
                X[i] = Y[i] + val;
(gdb) print i
<value optimized out>
```

Debug Headers : Src names → Asm names

```
# define LEN 32000
int X [ LEN ], Y [ LEN ], val ;
C0:  void foo (){
C1:    int i =0;
C2:    for ( ; i < LEN ; i ++)
C3:      X [ i ] = Y [ i ] + val ;
EC:  }
```

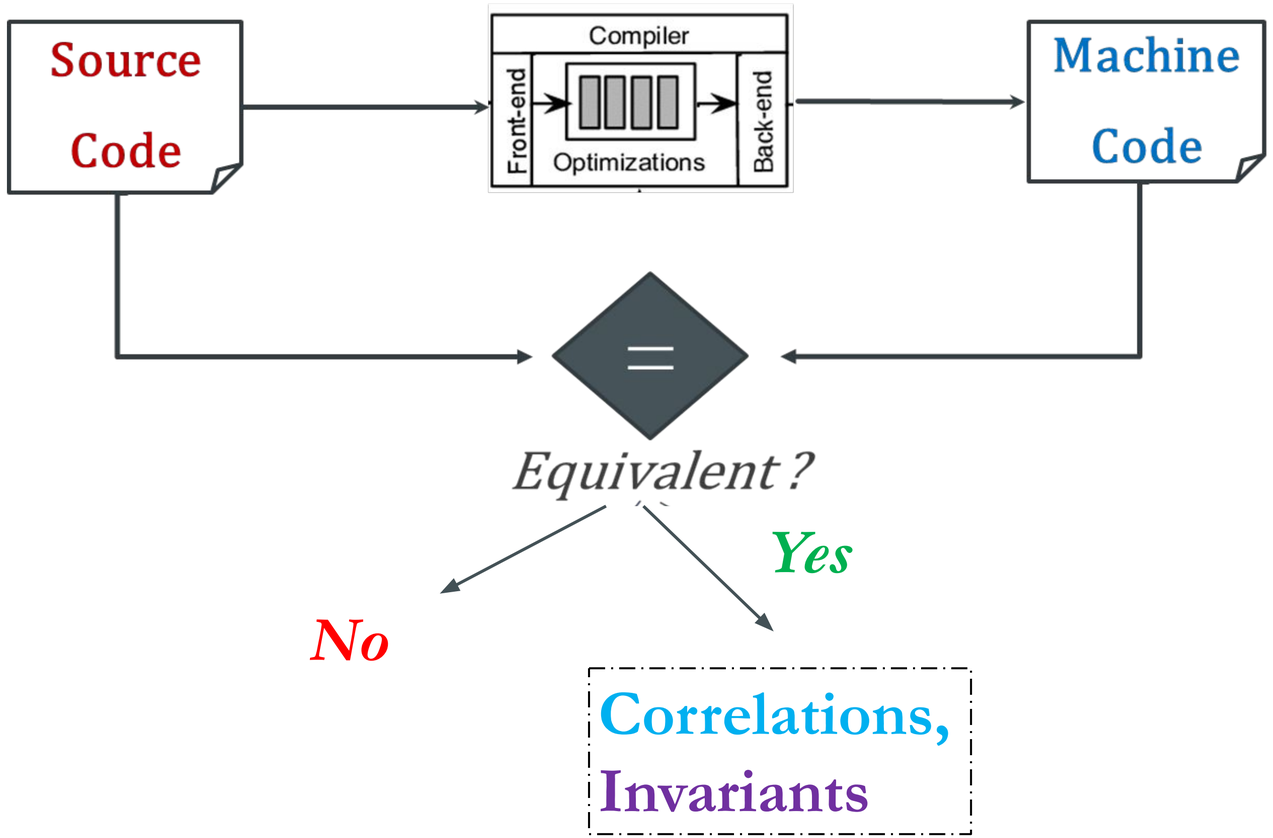
C Program

Hard for developers to maintain debugging information in the presence of aggressive optimization

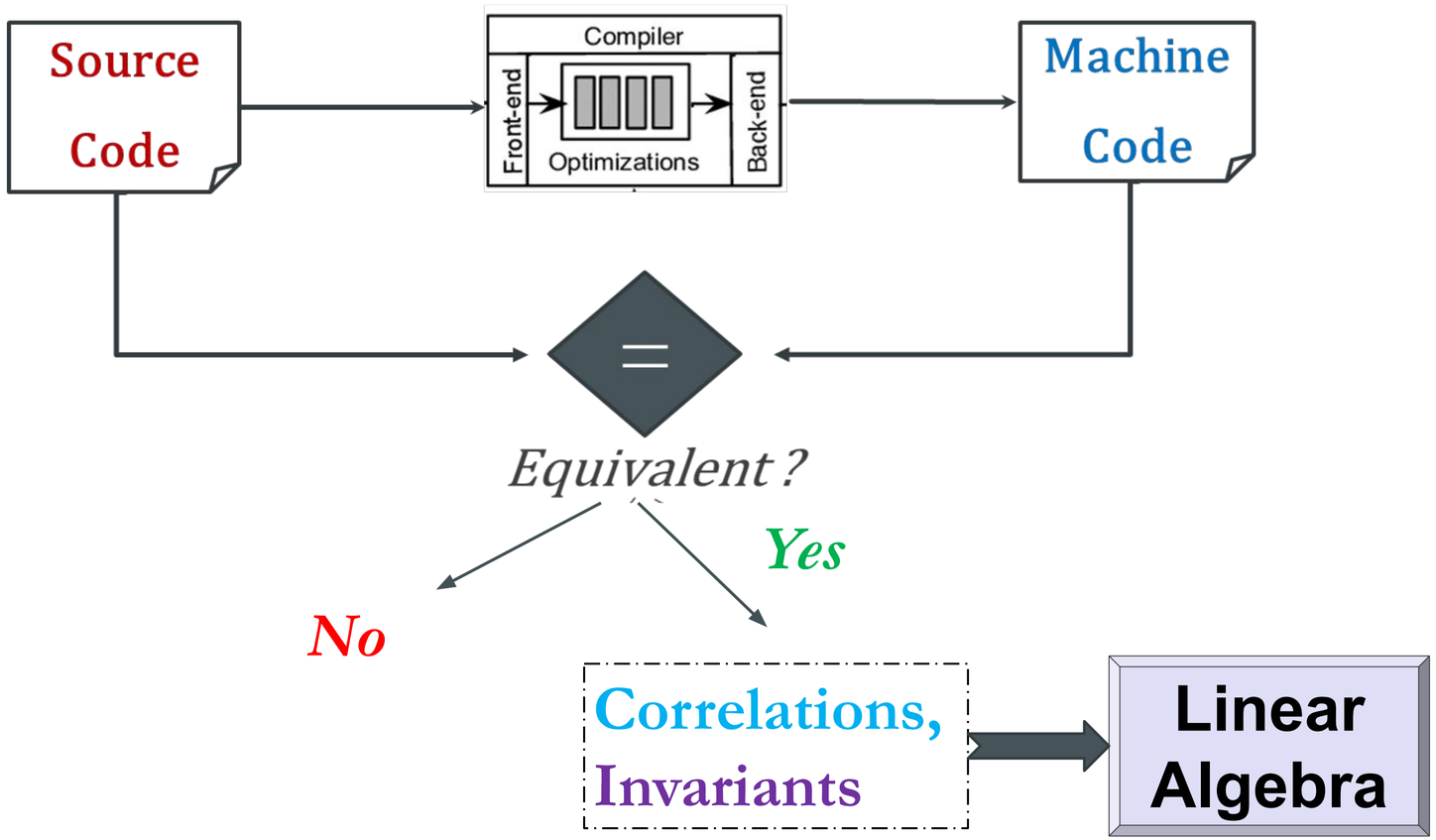
```
A0: foo:
A1:   r1 =& X [0]; r2 =& Y [0]
A2:   r3 = val
A3:   r4 = r1 + 4* LEN
A4:   mem [r1] = mem [r2] + r3
A5:    r1 += 4;  r2 += 4
A6:    if( r1 != r4 ) goto A4
EA:  ret
```

(abstracted) Assembly

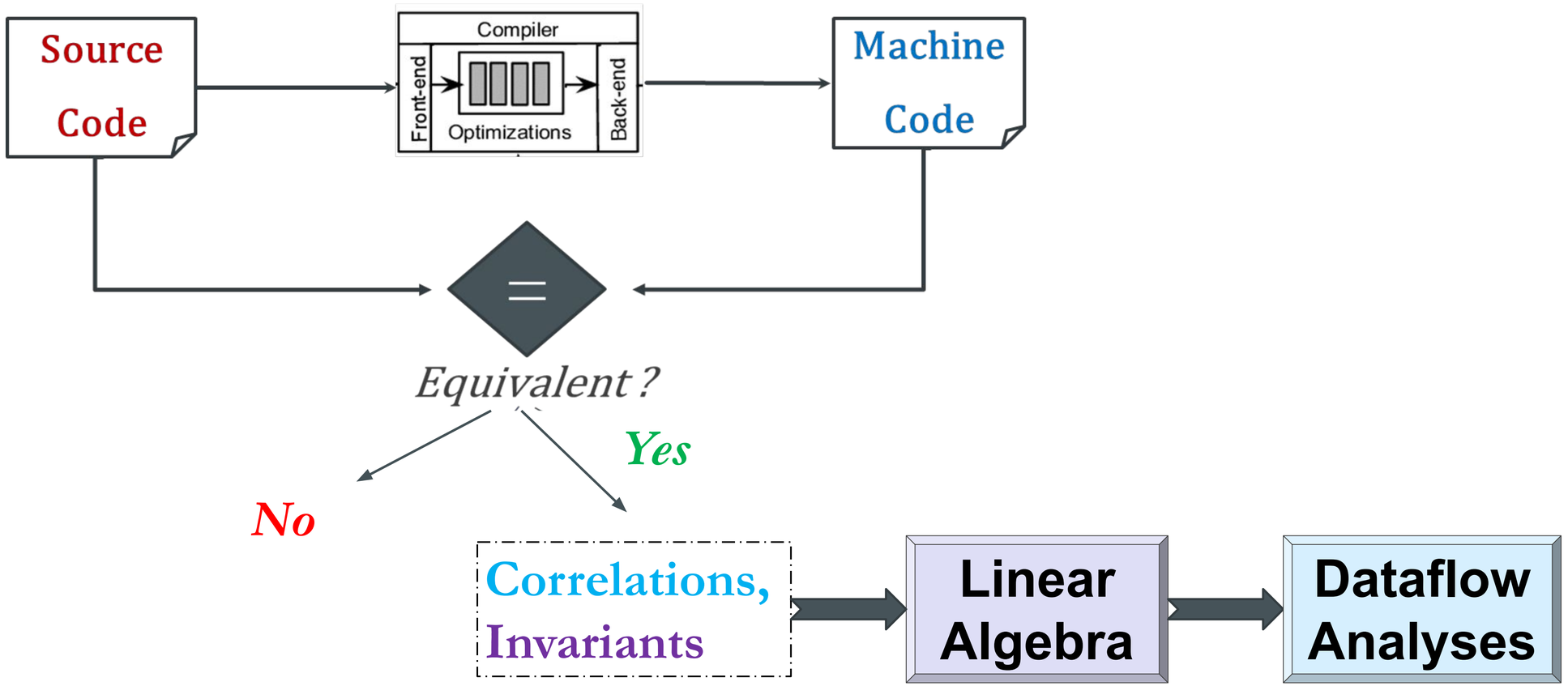
Automatic Generation of Debug Headers



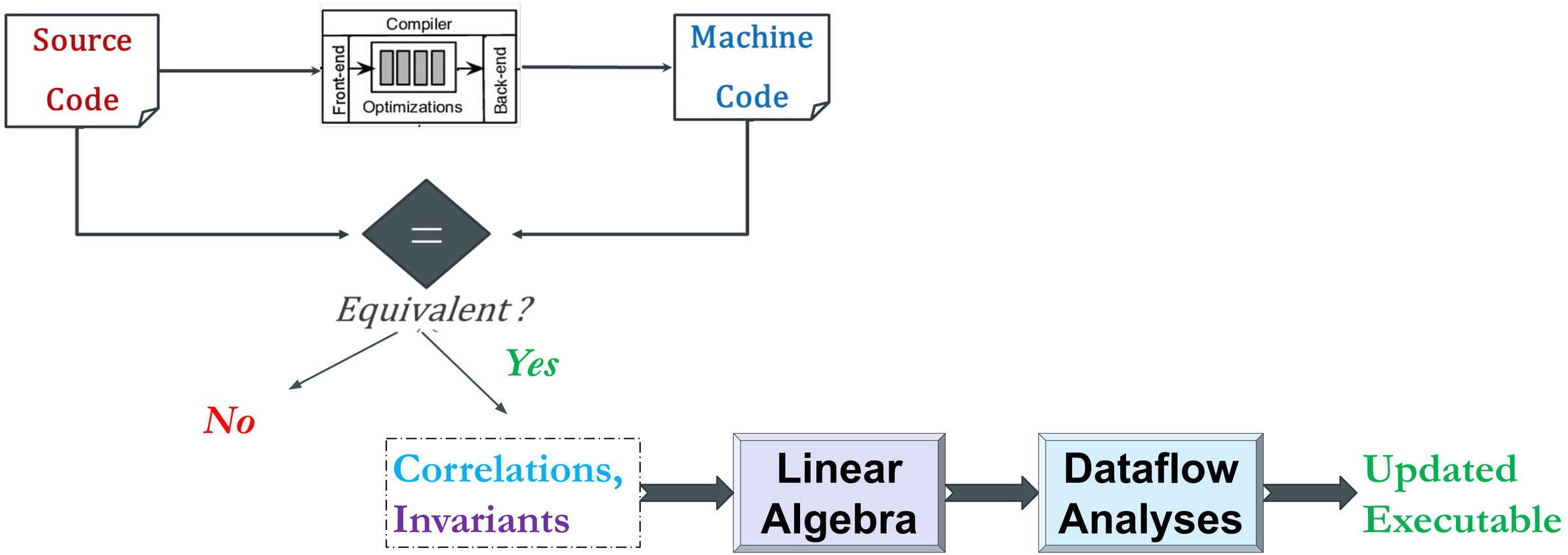
Automatic Generation of Debug Headers



Automatic Generation of Debug Headers



Automatic Generation of Debug Headers



Summary of Results

On the Testsuite for Vectorizing Compilers

Clang/LLVM	GCC	IntelCC
73 %	75 %	12 %

Percentage of PC-variable pairs where the debugging information was improved by this approach

Automatic Generation of Debug Headers through Blackbox Equivalence Checking (CGO 2022)

Vaibhav Kurhe, Pratik Karia, Shubhani, Abhishek Rose, Sorav Bansal
Indian Institute Of Technology Delhi

Counterexample-guided Verification of Imperative Programs Against Functional Specification

Problem Setting

- Specification – functional + strongly typed + safe e.g., Haskell
  - Easier to verify correctness
  - Slow to execute

- Implementation – imperative + explicit memory + control e.g., C
  - More optimization possibilities (manual + compiler)
  - Hard to verify correctness

Is implementation ‘equivalent to’ specification?

Applications

- Program verification
  - Manual spec + manual impl
  - Verify impl against spec

- Translation validation
  - Manual spec + (spec ⇒ impl) translator
  - Verify correctness of translator

Scope

- Vision…
  - Translation validation
  - Manually written spec in existing language (Haskell?)
  - A self-verifying optimizing compiler to C

- Our target…
  - Program verification
  - A minimal specification language called “Spec”
  - Manually written C impl
  - Small programs involving simple data structures

Spec Language

A minimal language called “Spec”

- Subset of functional features
- Scalar types
  - bool + bitvector
  - logical + arithmetic ops
- Algebraic data types (ADT)
  - constructors
  - pattern matching

```
type List = Nil
          | Cons (val:i32,tail:List).

fn rec(n:i32,i:i32,l:List) : List =
    if i ⩾ n then l
    else rec(n,i+1,Cons(i, l)).

fn mk_list (n:i32) : List =
    rec(n, 0, Nil).
```

Spec v. C – mk_list

```
type List = Nil
          | Cons (val:i32,tail:List).

fn rec(n:i32,i:i32,l:List) : List =
    if i ⩾ n then l
    else rec(n,i+1,Cons(i, l)).

fn mk_list (n:i32) : List =
    rec(n, 0, Nil).
```

```
typedef struct lnode {
    unsigned val;
    struct lnode* next;
} lnode;

lnode* mk_list(unsigned n) {
    lnode* l = NULL;
    for (unsigned i = 0; i < n; ++i) {
        lnode* p = malloc(sizeof lnode);
        p→val = i;
        p→next = l;
        l = p;
    }
    return l;
}
```

Spec v. C – strlen

```
type Str = Nil
         | Cons(ch:i8,tail:Str).

fn rec (s:Str,len:i32) : i32 =
    match s with
    | Nil ⇒ len
    | Cons(c,t) ⇒ rec(t,len+1).

fn strlen (s:Str) = rec(s,0).
```

```
size_t strlen(char* s) {
    size_t i = 0;
    for (i = 0;s[i];++i);
    return i;
}
```

Optimized strlen

- Chunked linked list layout

```
typedef struct clnode {
    char chunk[4];
    struct clnode* next;
} clnode;
```

- Optimized algorithm for x86
  - combined check for 4 bytes
  - used by glibc strlen impl

```
size_t strlen(clnode* s) {
  unsigned long himagic,lomagic,chunk,*chunk_ptr;
  himagic = 0x80808080; lomagic = 0x01010101;
  for (size_t i = 0;;s=s->next;i+=4) {
    // read next chunk (4 bytes)
    chunk_ptr = s->chunk;
    chunk = *chunk_ptr;
    // check for a zero byte in the chunk
    if (((chunk-lomagic) & ~chunk & himagic)!=0) {
      if (s->chunk[0] == 0) return i;
      if (s->chunk[1] == 0) return i+1;
      if (s->chunk[2] == 0) return i+2;
      if (s->chunk[3] == 0) return i+3;
    }
  }
}
```

Objectives

- Prove equivalence – Spec v. C
  - Equivalent ⇔ ‘identical’ observable behavior
    ⇔ ‘equal’ input ⇒ ‘equal’ outputs
  - Assume C allocations always succeed

- Proof method – bisimulation
  - Program ⇒ deterministic labeled transition system (IR)
  - Encode lock-step execution between both IR

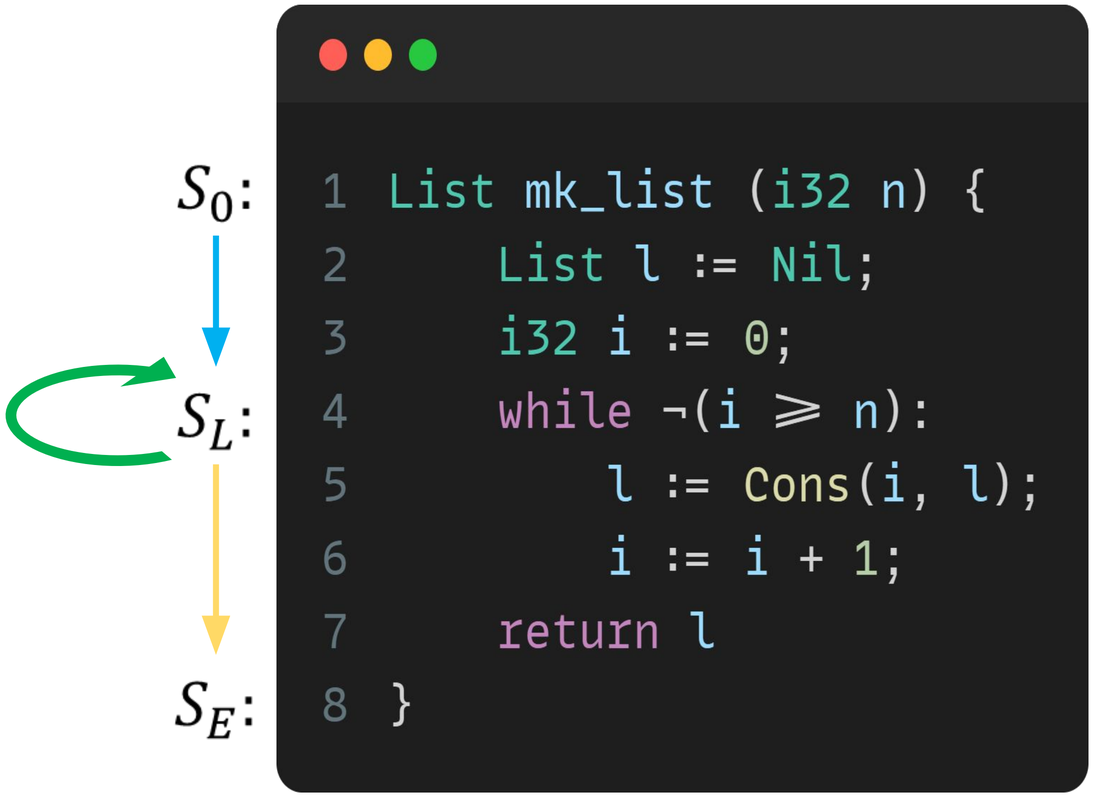
Spec + C IR Example – mk_list

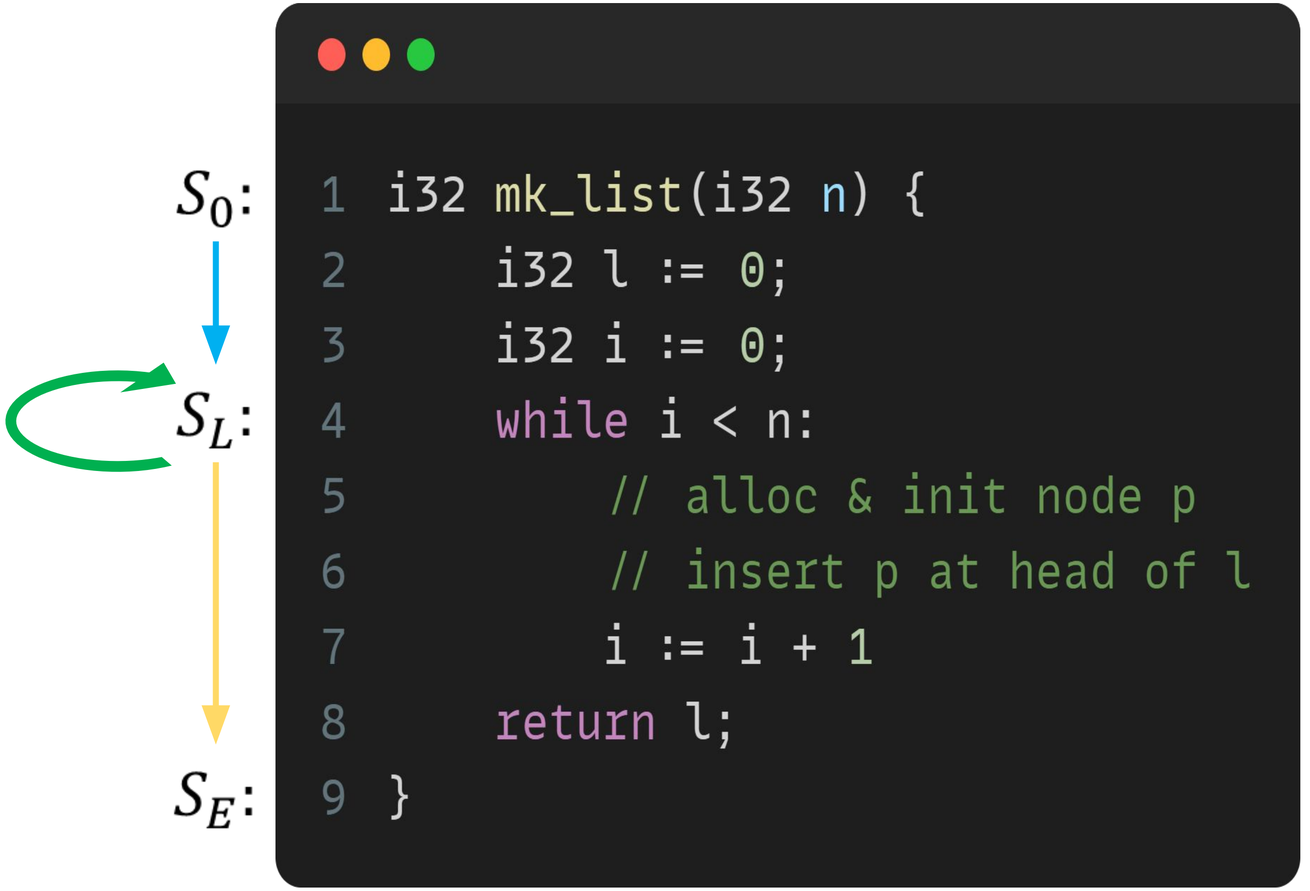
```
1 List mk_list (i32 n) {
2     List l := Nil;
3     i32 i := 0;
4     while ¬(i ⩾ n):
5         l := Cons(i, l);
6         i := i + 1;
7     return l
8 }
```

```
1  i32 mk_list(i32 n) {
2      i32 l := 0;
3      i32 i := 0;
4      while i < n:
5          i32 p := malloc(sizeof lnode);
6          m := m[&p→val ⇐= i];
7          m := m[&p→next ⇐= l];
8          l := p;
9          i := i + 1;
10     return l;
11 }
```

Bisimulation Example – mk_list

- Lock-step execution
- relations at S_0, S_L, S_E pairs





Subgoals

- A0: Spec + C ⇒ IR translator
  - Make C deterministic (also called C for simplicity)
- A1: Correlation search algorithm
  - Correlate lock-step transitions between Spec and C
- A2: Invariant inference algorithm
  - Find invariants at correlated points
- A3: Proof discharge algorithm
  - Prove queries generated by A1+A2

Contributions

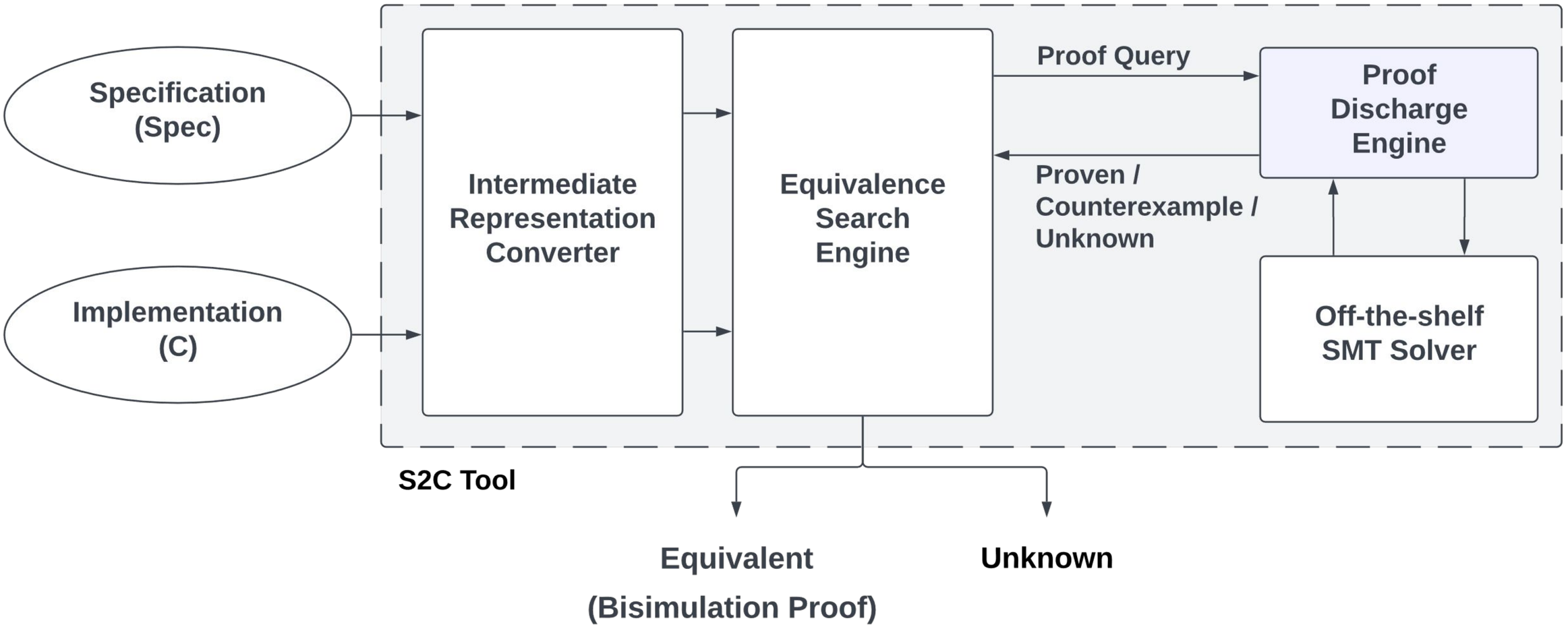
- Proof discharge algorithm (A3)
  - Queries contain ‘recursive relations’ (RR)
  - Returns proven | counterexample | ¯_(ツ)_/¯
- Automatic Spec-to-C equivalence checker tool (S2C)
  - A1+A2 ⇒ based on prior work on counterexample-guided C-to-ASM equivalence checker [1]
  - Uses A3 for discharging queries

A3 may make equivalence subqueries to S2C

- “higher order” equivalence…

[1] Shubhani Gupta et al. 2020. Counterexample-guided correlation algorithm for translation validation. Proc. ACM Program. Lang. 4, OOPSLA

S2C Diagram


Specification
(Spec)
Implementation
(C)
Intermediate
Representation
Converter
Equivalence
Search
Engine
Proof Query
Proven /
Counterexample /
Unknown
Proof
Discharge
Engine
Off-the-shelf
SMT Solver
S2C Tool
Equivalent
(Bisimulation Proof)
Unknown

Soundness of A3 and S2C

- Output of Proof discharge algorithm (A3)

  - Proven ⇒ iff query is provable
    - ensures… soundness of S2C
  - Counterexample Γ ⇒ iff Γ falsifies the query
    - increases… scalability of S2C
  - Unknown ⇒ query may or may not be true
    - decreases… completeness of S2C

Spec – Intermediate Representation

- Pattern matching ⇒ if-then-else
- Inline & loopify (tail) calls
  - limitation of our tool
- Algebra contains…
  - bool + bitvector types + ops
  - constructors ⇒ Cons(x, y)
  - sum_is ⇒ x is Nil
  - prod_get ⇒ x.tail

```
1 List mk_list (i32 n) {
2       List l := Nil;
3       i32 i := 0;
4       while ¬(i ⩾ n):
5             l := Cons(i, l);
6             i := i + 1;
7       return l
8 }
```

Spec v. Spec IR

```
type List = Nil
          | Cons (val:i32,tail:List).

fn rec(n:i32,i:i32,l:List) : List =
    if i ⩾ n then l
    else rec(n,i+1,Cons(i, l)).

fn mk_list (n:i32) : List =
    rec(n, 0, Nil).
```

```
List mk_list (i32 n) {
    List l := Nil;
    i32 i := 0;
    while ¬(i ⩾ n):
        l := Cons(i, l);
        i := i + 1;
    return l
}
```

C – Intermediate Representation

- Concretize type size + layout
  - pointer ⇒ i32
- Explicit memory with r/w
  - memory (m) ⇒ Array(i32,i8)
- Algebra contains…
  - bool + bitvector types + ops
  - bitvector array select + store

```
1  i32 mk_list(i32 n) {
2      i32 l := 0;
3      i32 i := 0;
4      while i < n:
5          i32 p := malloc(sizeof lnode);
6          m := m[&p→val ⇐ i];
7          m := m[&p→next ⇐ l];
8          l := p;
9          i := i + 1;
10     return l;
11 }
```

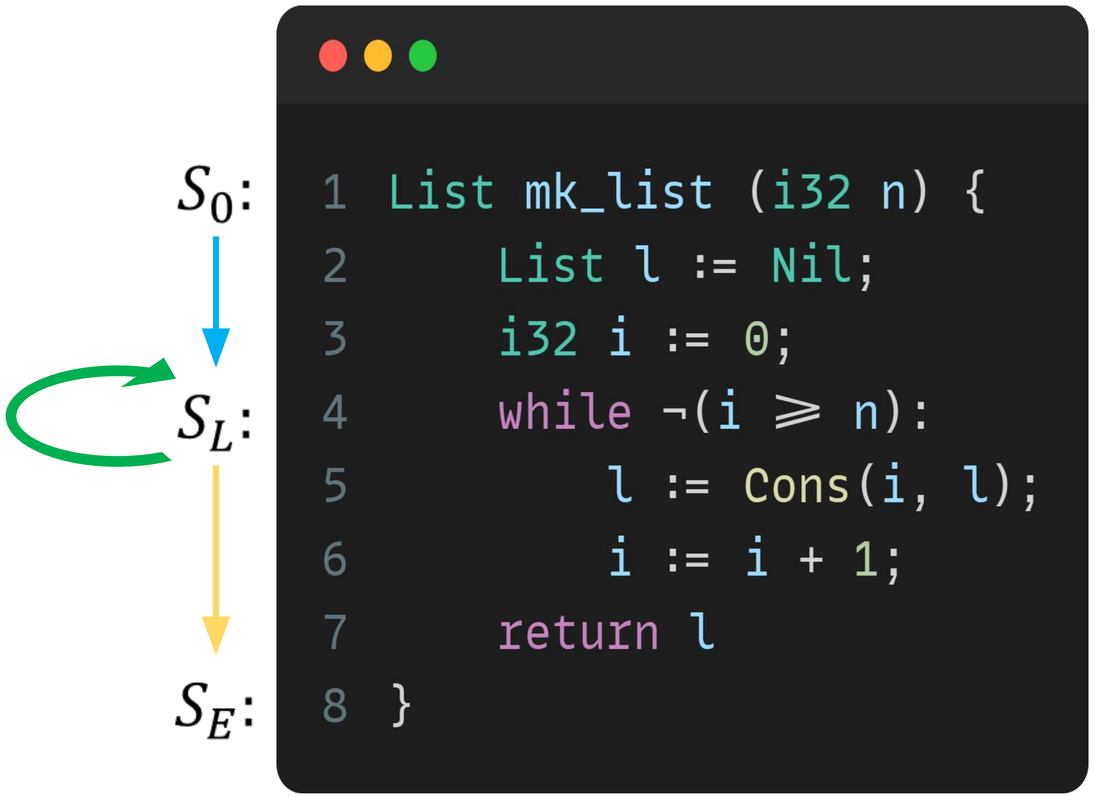
C v. C IR

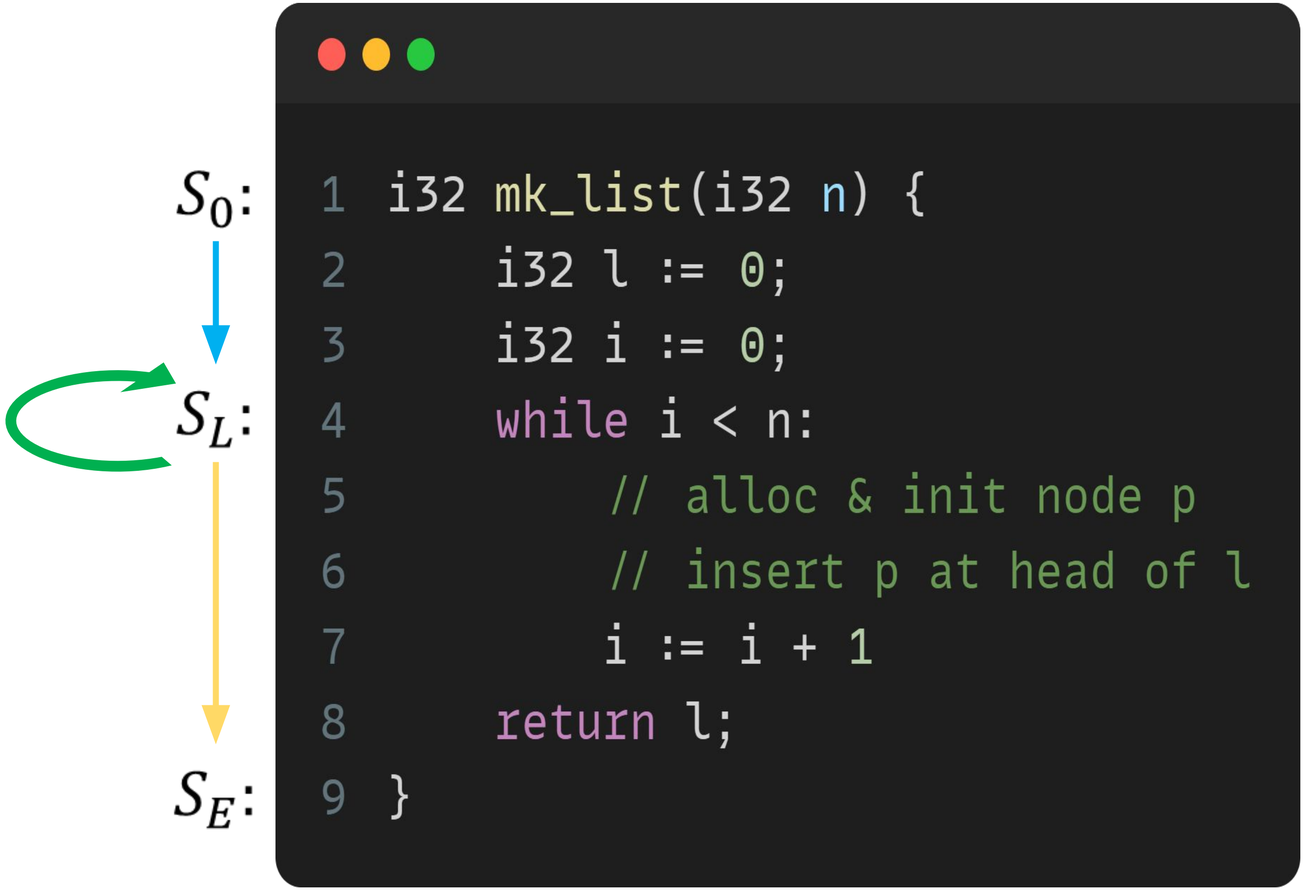
```c
typedef struct lnode {
    unsigned val;
    struct lnode* next;
} lnode;

lnode* mk_list(unsigned n) {
    lnode* l = NULL;
    for (unsigned i = 0; i < n; ++i) {
        lnode* p = malloc(sizeof lnode);
        p→val = i;
        p→next = l;
        l = p;
    }
    return l;
}
```

```
i32 mk_list(i32 n) {
    i32 l := 0;
    i32 i := 0;
    while i < n:
        i32 p := malloc(sizeof lnode);
        m := m[&p→val ⟸ i];
        m := m[&p→next ⟸ l];
        l := p;
        i := i + 1;
    return l;
}
```

Revisiting Bisimulation

Express equal “l” values ⇒ List v. i32





Invariants

- Entry Precondition at (S_0, C_0): $n = n$
- Loop Invariants at (S_L, C_L): $n = n \wedge i = i \wedge l \sim Clist_{\mathbf{m}}(l)$
- Exit Postcondition at (S_E, C_E): $ret \sim Clist_{\mathbf{m}}(ret)$

Recursive Relations (RR) $\Rightarrow l \sim Clist_{\mathbf{m}}(l)$

- ADT equality between Spec & C values

Lifting Constructor...

$$Clist_{\mathbf{m}}(l\colon i32) \triangleq \underline{\text{if}}\ (l = 0)\ \underline{\text{then}}\ Nil \\ \underline{\text{else}}\ Cons(l \rightarrow val, Clist_{\mathbf{m}}(l \rightarrow next))$$

Hoare Triple

Prove invariants along a correlated finite path…

- Proof Query to Hoare triple… $\{P\}\ C\ \{Q\}$
- Lowers to… $P \wedge pathcond(C) \Rightarrow weakest\_precond_C(Q)$
- Expressed as… $LHS \Rightarrow RHS$

$$l \sim Clist_{\mathbf{m}}(l) \text{ along } (S_0 \to S_L, C_0 \to C_L)$$

$$\Updownarrow$$

$$\{n = n\}(S_0 \to S_L, C_0 \to C_L)\{l \sim Clist_{\mathbf{m}}(l)\}$$

$$\Updownarrow$$

$$n = n \ \Rightarrow Nil \sim Clist_{\mathbf{m}}(0)$$

Query algebra

- Spec + C IR introduces…
  - bool + bitvector types + ops
  - ADT constructor application (⋆)
  - sum_is + prod_get (⋆)
  - array over bitvector + select & store ops
- Invariants relating Spec ADT values with C values require…
  - ADT equality of the form $S \sim lift(C)$ (⋆)
    - S ⇒ Spec ADT value
    - C ⇒ Lifted C value (bool + bitvector)
- Cannot be handled directly by SMT solvers…

Proof Discharge Algorithm

Decomposition of Recursive Relations

- Decompose $l_1 \sim l_2$ into an equivalent set of clauses
  - Clauses have the shape…
    - $P \Rightarrow (A = B)$
    - $P \Rightarrow (A \sim B)$

$$Cons(a, b) \sim Cons(c, d)$$

$$\Updownarrow$$

$$a = c \wedge b \sim d$$

Decomposition of Recursive Relations

- Decompose in presence of lifting constructors…

$$Cons(a, b\ ) \sim Clist_{\mathbf{m}}(l)$$

$$\Updownarrow$$

$$\wedge \begin{cases} l \neq 0 \\ l \neq 0 \Rightarrow a = l \rightarrow val \\ l \neq 0 \Rightarrow b \sim Clist_{\mathbf{m}}(l \rightarrow next) \end{cases}$$

$$Clist_{\mathbf{m}}(l\colon i32) \triangleq \underline{\text{if}}\ (l = 0)\ \underline{\text{then}}\ Nil \\ \underline{\text{else}}\ Cons(l \rightarrow val, Clist_{\mathbf{m}}(l \rightarrow next))$$

Proof Query Example

$$Clist_{\mathbf{m}}(l{:}\,i32) \triangleq$$
$$\underline{\text{if}}\ (l = 0)\ \underline{\text{then}}\ Nil$$
$$\underline{\text{else}}\ Cons(l \to val, Clist_{\mathbf{m}}(l \to next))$$

- Consider the proof query…

$$l \sim Clist_{\mathbf{m}}(l) \text{ along } (S_0 \to S_L, C_0 \to C_L)$$

$$\Updownarrow$$

$$\{n = n\}(S_0 \to S_L, C_0 \to C_L)\{l \sim Clist_{\mathbf{m}}(l)\}$$

$$\Updownarrow$$

$$n = n \Rightarrow Nil \sim Clist_{\mathbf{m}}(0)$$

$$\Updownarrow \quad \text{decompose once…}$$

$$n = n \Rightarrow (0 = 0) \qquad \text{no recursive relations!}$$

Solved by SMT Solver… (trivial in this case)

Types of Proof Queries

- Decompose $LHS \Rightarrow RHS$ 'k' times...
  - $LHS_k \Rightarrow RHS_k$

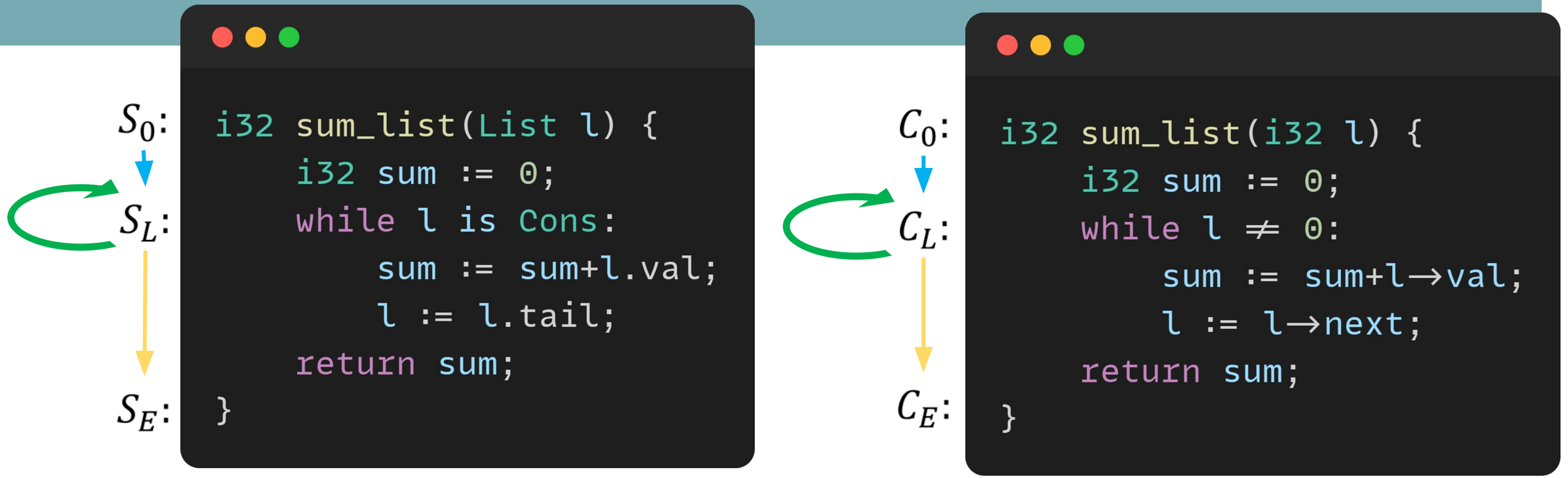
Type		
Type I	✖	✖
Type II	✔	✖
Type III	–	✔

Spec vs C – sum_list

```
type List = Nil
          | Cons (val:i32,tail:List).

fn rec(l:List,sum:i32) : i32 =
    match l with
    | Nil ⇒ sum
    | Cons(v,t) ⇒ rec(t, sum+v).

fn sum_list (l:List) : i32 =
    rec(l, 0).
```

```
typedef struct lnode {
    unsigned val;
    struct lnode* next;
} lnode;

unsigned sum_list(lnode* l) {
    unsigned sum = 0;
    while (l) {
        sum += →val;
        l = l→next;
    }
    return l;
}
```

Bisimulation – sum_list



Entry Precondition at (S_0, C_0): $l \sim Clist_{\mathbf{m}}(l)$

Loop Invariants at (S_L, C_L): $l \sim Clist_{\mathbf{m}}(l) \wedge sum = sum$

Exit Postcondition at (S_E, C_E): $ret = ret$

list

"$l.val$" represents an ADT prod_get expr

$l : List$

$l.val : i32$

$\Updownarrow$

$prod\_get(l, val)$

```
while l is :
    sum := sum+l.val;
    l := l.tail;
return sum;
}
```

S_L:

S_E:

"$l \rightarrow val$" represents a C memory read expr

$m : Array(i32, i8)$

$l \rightarrow val : i32$

$\Updownarrow$

$$bvconcat \begin{pmatrix} m[l \rightarrow val \quad] \\ m[l \rightarrow val + 1] \\ m[l \rightarrow val + 2] \\ m[l \rightarrow val + 3] \end{pmatrix}$$

```
while l ≠ 0:
    sum := sum+l→val;
    l := l→next;
return sum;
}
```

C_L:

C_E:

Entry Precondition at (S_0, C_0): $l \sim Clist_{\mathbf{m}}(l)$

Loop Invariants at (S_L, C_L): $l \sim Clist_{\mathbf{m}}(l) \wedge sum = sum$

Exit Postcondition at (S_E, C_E): $ret = ret$

Type II Query - Only LHS has $\sim$

$$\begin{array}{l} Clist_{\mathbf{m}}(l{:}\,i32) \triangleq \\ \underline{\text{if}}\ (l = 0)\ \underline{\text{then}}\ Nil \\ \underline{\text{else}}\ Cons(l \to val, Clist_{\mathbf{m}}(l \to next)) \end{array}$$

Consider the proof query…

$$sum = sum \text{ along } (S_L \to S_L, C_L \to C_L)$$

$$\Updownarrow$$

$$\{l \sim Clist_{\mathbf{m}}(l) \wedge sum = sum\}(S_0 \to S_L, C_0 \to C_L)\{sum = sum\}$$

$$\Updownarrow$$

$$\underline{l \sim Clist_{\mathbf{m}}(l)} \wedge sum = sum \Rightarrow (sum + l.val) = (sum + l \to val)$$

Decomposition of $l \sim Clist_{\mathbf{m}}(l)$ always contains recursive relations!

- l has arbitrary length…

Type II Query : Over-Approximation

- $l \sim Clist_{\mathbf{m}}(l) \quad \Leftrightarrow \quad$ infinite set of equalities

Over-approximate LHS to LHS_o

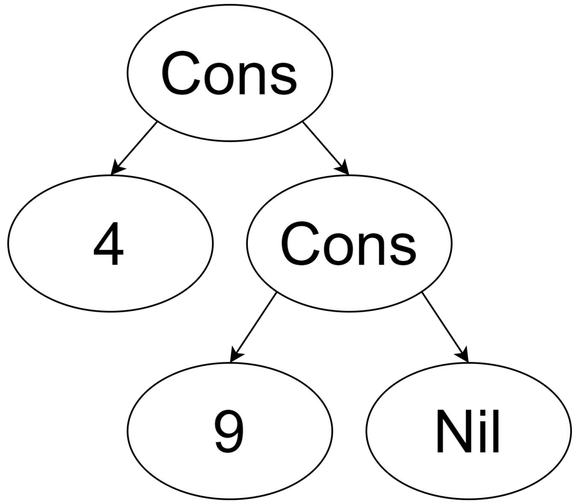
- Only include scalar equalities till *depth* ‘d’
- Weaker condition
- Written $\sim_d$

If $LHS_o \Rightarrow RHS$ is provable then so is $LHS \Rightarrow RHS$!

Type II : Depth of Values

$$Clist_{\mathbf{m}}(l{:}\,i32) \triangleq \\ \underline{\text{if}}\ (l = 0)\ \underline{\text{then}}\ Nil \\ \underline{\text{else}}\ Cons(l \to val, Clist_{\mathbf{m}}(l \to next))$$

- Depth of ADT values…
  - Depth of expression tree diagram
    - $Cons(42, Nil) \Rightarrow 1$
    - $Cons(4, Cons(9, Nil)) \Rightarrow 2$



$$l \sim_1 Clist_{\mathbf{m}}(l) \Leftrightarrow \wedge \begin{cases} (l\ is\ Nil) = (l = 0) \\ (l\ \text{is}\ Cons) \wedge (l \neq 0) \Rightarrow l.val = l \to val \end{cases}$$

Type II : Under-Approximation

- Under-approximate LHS to LHS_u
  - Include scalar equalities till depth ‘d’ $\equiv\ l_1 \sim_1 l_2$
  - Assert that both values have a max depth of ‘d’ $\equiv\ \Gamma_{\mathrm{d}}(l_1) \wedge \Gamma_{\mathrm{d}}(l_2)$
  - Stronger condition
  - Written $\approx_d$

If $LHS_u \Rightarrow RHS$ is disprovable then so is $LHS \Rightarrow RHS$!

- If Γ is a counterexample to $LHS_u \Rightarrow RHS$
  then Γ is also a counterexamples to $LHS \Rightarrow RHS$

Type II : Under-Approximation

$$Clist_{\mathbf{m}}(l{:}\,i32) \triangleq$$
$$\underline{\text{if}}\ (l = 0)\ \underline{\text{then}}\ Nil$$
$$\underline{\text{else}}\ Cons(l \to val, Clist_{\mathbf{m}}(l \to next))$$

Under-approximation formula…

$$l_1 \approx_d l_2 \Leftrightarrow l_1 \sim_d l_2 \wedge \Gamma_d(l_1) \wedge \Gamma_d(l_2)$$

Example…

$$l \approx_2 Clist_{\mathbf{m}}(l) \Leftrightarrow l \sim_2 Clist_{\mathbf{m}}(l) \wedge \Gamma_2(l) \wedge \Gamma_2\big(Clist_{\mathbf{m}}(l)\big)$$

$$\Gamma_2\big(Clist_{\mathbf{m}}(l)\big) \Leftrightarrow \vee \begin{cases} l = 0 \\ l \neq 0 \wedge l \to next = 0 \end{cases}$$

Revisiting Query algebra

- Spec + C IR introduces…
  - bool + bitvector types + ops
  - ADT constructor application ($\star$)
  - sum_is + prod_get ($\star$)
  - array over bitvector + select & store ops
- Invariants relating Spec ADT values with C values require…
  - ADT equality of the form $S \sim lift(C)$ ($\star$)
    - $S \Rightarrow$ Spec ADT value
    - $C \Rightarrow$ Lifted C value (bool + bitvector)
- Cannot be handled directly by SMT solvers…

SMT Encoding & Reconciliation

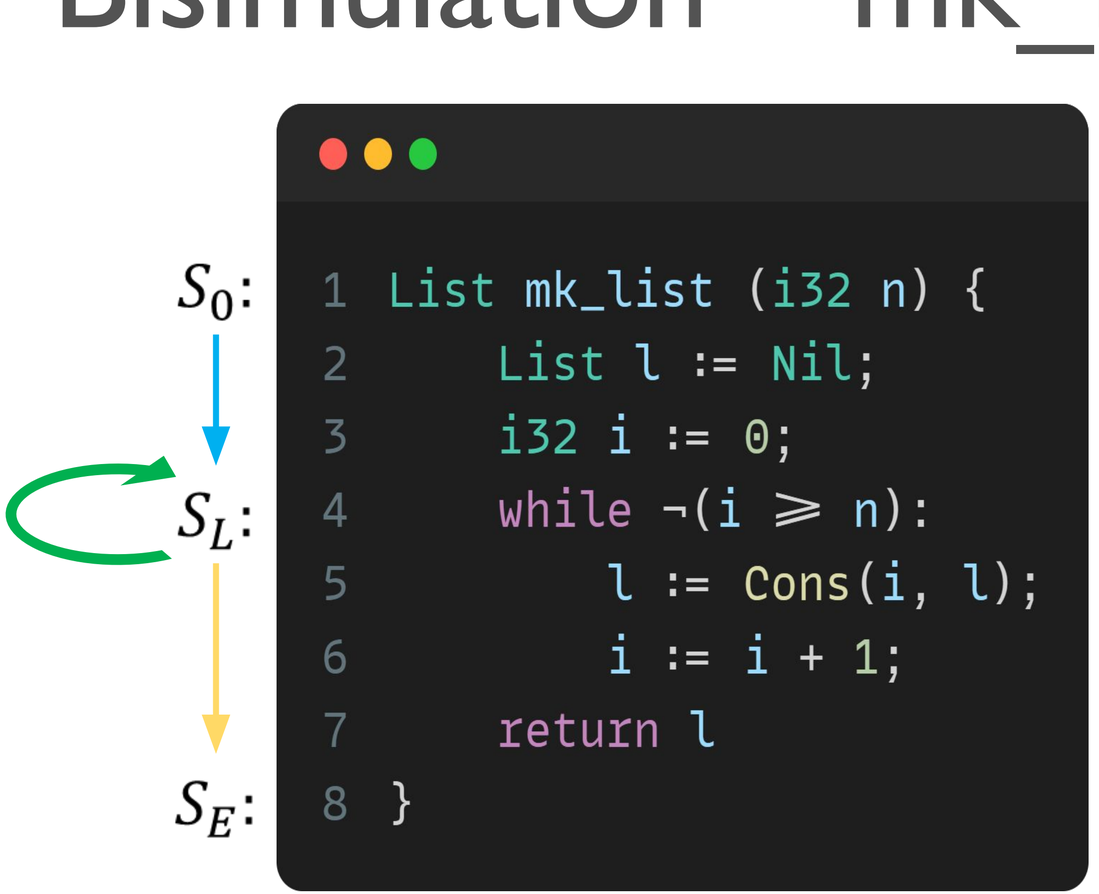
- Queries solved by SMT solvers
  - Type I
  - Type II underapprox + overapprox
- Algebra contains…
  - ADT constructor application (⋆)
  - sum_is + prod_get (⋆)
  - No ADT equalities… only scalar equalities…
- Convert to equivalent SMT logic ⇒ encode
- Recover counterexample for original query ⇒ reconcile

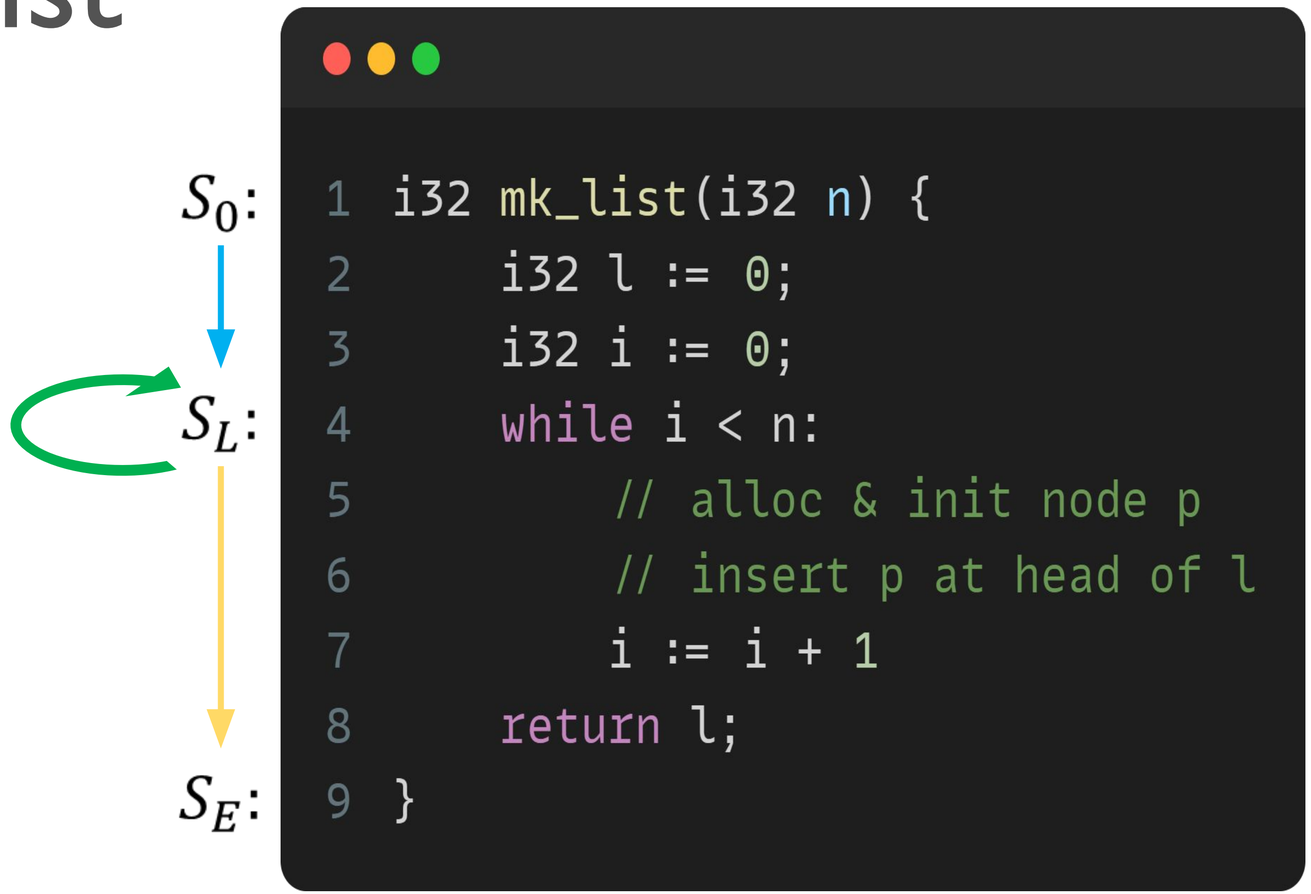
SMT Encoding & Reconciliation

- Normalize & encode…
  - “l is Nil” ⇒ “$l\_tag = Nil$”
  - “$l.val$” ⇒ “l_val”
  - “$Cons(4, l).tail.val$” ⇒ “$l.val$” ⇒ “l_val”

- Reconcile assignments…
  - $\{l\_tag \rightarrow Cons, l\_val \rightarrow 23, l\_tail\_tag \rightarrow Nil\}$
    $\Downarrow$
  - $\{l \rightarrow Cons(23, Nil)\}$

Bisimulation – mk_list





Entry Precondition at (S_0, C_0): $n = n$

Loop Invariants at (S_L, C_L): $n = n \wedge i = i \wedge l \sim Clist_{\mathbf{m}}(l)$

Exit Postcondition at (S_E, C_E): $ret \sim Clist_{\mathbf{m}}(ret)$

Type III Query - RHS has ~

$$Clist_{\mathbf{m}}(l{:}\,i32) \triangleq \\ \underline{\text{if}}\ (l = 0)\ \underline{\text{then}}\ Nil \\ \underline{\text{else}}\ Cons(l \to val, Clist_{\mathbf{m}}(l \to next))$$

- Consider the proof query…

$$l \sim Clist_{\mathbf{m}}(l) \text{ along } (S_L \to S_L, C_L \to C_L)$$

$$\Updownarrow$$

$$\{i = i \wedge l \sim Clist_{\mathbf{m}}(l)\}(S_0 \to S_L, C_0 \to C_L)\{l \sim Clist_{\mathbf{m}}(l)\}$$

$$\Updownarrow$$

$$i = i \wedge \underline{l \sim Clist_{\mathbf{m}}(l)} \wedge p = malloc() \Rightarrow \underline{Cons(i, l) \sim Clist_{\mathbf{m}'}(p)}$$

$$\mathbf{m}' = \mathbf{m}[\&p \to val \Leftarrow i][\&p \to next \Leftarrow l]$$

$$\mathbf{m}' = \mathbf{m}[\&p \to val \Leftarrow i][\&p \to next \Leftarrow l]$$

Type III : RHS Decomposition

- $i = i \wedge l \sim Clist_{\mathbf{m}}(l) \wedge p = malloc() \Rightarrow Cons(i, l) \sim Clist_{\mathbf{m}'}(p)$

$$\Updownarrow$$

$$i = i \wedge l \sim Clist_{\mathbf{m}}(l) \wedge p = malloc() \Rightarrow Cons(i, Clist_{\mathbf{m}}(l)) \sim Clist_{\mathbf{m}'}(p)$$

Decompose RHS…

- $LHS \Rightarrow \neg(p = 0)$
- $LHS \Rightarrow \neg(p = 0) \Rightarrow (i = p \overset{\mathbf{m}'}{\to} val)$

} Type II

- $LHS \Rightarrow \neg(p = 0) \Rightarrow \underline{Clist_{\mathbf{m}}(l) \sim Clist_{\mathbf{m}'}(p \overset{\mathbf{m}'}{\to} next)}$

Type III – RHS has ~

$$\mathbf{m}' = \mathbf{m}[\&p \rightarrow val \Leftarrow i][\&p \rightarrow next \Leftarrow l]$$

-

$$Clist_{\mathbf{m}}(l) \sim Clist_{\mathbf{m}'}(p \xrightarrow{\mathbf{m}'} next)$$

$\Updownarrow$ simplification using **m**’

$$Clist_{\mathbf{m}}(l) \sim Clist_{\mathbf{m}'}(l)$$

Reinterpret lifting constructors as programs…

Equality of Values

$\Updownarrow$

Equivalence of their ‘deconstruction programs’

Deconstruction program

$$\mathcal{C}list_{\mathbf{m}}(l{:}\,i32) \triangleq$$
$$\underline{\text{if}}\ (l = 0)\ \underline{\text{then}}\ Nil$$
$$\underline{\text{else}}\ Cons(l \to val,\ Clist_{\mathbf{m}}(l \to next))$$

```
List Clist(i32 l,Array(i32,i8) m) {
    if (l == 0) {
        return Nil;
    } else {
        val := l→val;
        tail := Clist(l→next, m);
        return Cons(val, tail);
    }
}
```

Bisimulation

$$Clist_{\mathbf{m}}(l) \sim Clist_{\mathbf{m}'}(l)$$

Entry precondition $\Leftrightarrow l^f = l = l^s \wedge \mathrm{m}^f = \mathbf{m} \wedge m^s = \mathbf{m}'$

```
List Clist(i32 l,Array(i32,i8) m) {
    if (l == 0) {
        return Nil;
    } else {
        val := l→val;
        tail := Clist(l→next, m);
        return Cons(val, tail);
    }
}
```

```
List Clist(i32 l,Array(i32,i8) m) {
    if (l == 0) {
        return Nil;
    } else {
        val := l→val;
        tail := Clist(l→next, m);
        return Cons(val, tail);
    }
}
```

Bisimulation

$$\mathbf{m}' = \mathbf{m}[\&p \rightarrow val \Leftarrow i][\&p \rightarrow next \Leftarrow l]$$

Proof Obligations…

- $\left(l^f = 0\right) = (l^s = 0)$ – follows from the precondition
- $\left(l^f \neq 0\right) \wedge (l^s \neq 0) \Rightarrow l^f \xrightarrow{\mathbf{m}} val = l^s \xrightarrow{\mathbf{m}'} val$
- $\left(l^f \neq 0\right) \wedge (l^s \neq 0) \Rightarrow l^f \xrightarrow{\mathbf{m}} next = l^s \xrightarrow{\mathbf{m}'} next$

} read on write!

- Writes do not alias with reads ⇒ pointer analysis

Pointer Analysis

- May-point-to analysis
  - Forward DFA
  - Segment the memory into…
    - Two regions ⇒ (1) most recent call (2) rest
      - Allocation site A ⇒ $\{A_1, A_{2+}\}$ regions
    - All other regions ⇒ $\mathcal{H}(heap)$
  - Run on the C IR before equivalence check begins…
  - Run on the deconstruction programs before equivalence…

Pointer Analysis – C

- Allocation sites $\Rightarrow \{B\}$
- Regions $\Rightarrow \bot \triangleq \{B_1, B_2, \mathcal{H}\}$
- Notation $\Rightarrow ptr \rightsquigarrow R$ at X

- Examples…
  - $n \rightsquigarrow \top$ at #1
  - $l \rightsquigarrow \phi$ at #3

```
1  i32 mk_list(i32 n) {
2      i32 l := 0;
3      i32 i := 0;
4      while i < n:
5          i32 p := malloc(sizeof lnode);
6          m := m[&p→val ⇐ i];
7          m := m[&p→next ⇐ l];
8          l := p;
9          i := i + 1;
10     return l;
11 }
```

Bisimulation Proof

$$\mathbf{m}' = \mathbf{m}[\&p \rightarrow val \Leftarrow i][\&p \rightarrow next \Leftarrow l]$$

Revisiting proof obligations…

- $\left(l^f \neq 0\right) \wedge \left(l^s \neq 0\right) \Rightarrow l^f \overset{\mathbf{m}}{\rightarrow} val = l^s \overset{\mathbf{m}'}{\rightarrow} val$
- $\left(l^f \neq 0\right) \wedge \left(l^s \neq 0\right) \Rightarrow l^f \overset{\mathbf{m}}{\rightarrow} next = l^s \overset{\mathbf{m}'}{\rightarrow} next$
- Points-to info…

$$l^s \rightsquigarrow \{A_2\} \text{ but } p \rightsquigarrow \{A_1\} \text{ in } \mathrm{m}^s$$

- Provable with the points-to information…

Proof Discharge Algorithm - Summary

- Decompose and categorize $LHS \Rightarrow RHS$

Type I

- Solve using SMT Solvers

Type II

- Overapproximate LHS into LHS_o and attempt to prove $LHS_o \Rightarrow RHS$
- Underapproximate LHS into LHS_u and attempt to disprove $LHS_u \Rightarrow RHS$

Type III

- LHS-to-RHS substitute and decompose RHS
- Solve recursive relations using bisimulation of reconstruction programs
- Solve rest using Type II algorithm

Evaluation

Evaluation - Config

SMT Solvers – timeout of 1 second

- z3-4.8.7
- z3-4.8.14
- yices2-45e38fc
- cvc4-1.7

S2C tool parameters…

- Unroll factor ⇒ 4 – loop unrolling in C
- Approximation depths ⇒ 8 – type II query
- Categorization parameter ⇒ 5 – identify query type

Experiments

Specification ⇒ manually written

Implementations ⇒

- Taken from C libs (e.g., glibc, klibc, newlibc, openbsd)
- Manually written (e.g., tree traversal, matrix frequency count)

Str = SInvalid | SNil | SCons(i8, Str)	strlen, strchr, strcmp, strspn, strcspn, strpbrk	null character terminated (array, linked list, chunked linked list)
List = LNil | LCons(i32, List)	listsum, listdot	variable length array, linked list, chunked linked list
Tree = TNil | TCons(i32, Tree, Tree)	treesum	pointer node-based, array index-based
Matrix = MNil | MCons(List, Matrix)	matfreq	2D array, row-major, column-major, array of chunked linked lists etc..

Spec v. C – strlen

```
type Str = Nil
         | Cons(ch:i8,tail:Str).

fn rec (s:Str,len:i32) : i32 =
    match s with
    | Nil ⇒ len
    | Cons(c,t) ⇒ rec(t,len+1).

fn strlen (s:Str) = rec(s,0).
```

```
size_t strlen(char* s) {
    size_t i = 0;
    for (i = 0;s[i];++i);
    return i;
}
```

Optimized strlen

- Chunked linked list layout

```c
typedef struct clnode {
    char chunk[4];
    struct clnode* next;
} clnode;
```

- Optimized algorithm for x86
  - combined check for 4 bytes
  - used by glibc strlen impl

```c
size_t strlen(clnode* s) {
  unsigned long himagic,lomagic,chunk,*chunk_ptr;
  himagic = 0x80808080; lomagic = 0x01010101;
  for (size_t i = 0;;s=s->next;i+=4) {
    // read next chunk (4 bytes)
    chunk_ptr = s->chunk;
    chunk = *chunk_ptr;
    // check for a zero byte in the chunk
    if (((chunk-lomagic) & ~chunk & himagic)!=0) {
      if (s->chunk[0] == 0) return i;
      if (s->chunk[1] == 0) return i+1;
      if (s->chunk[2] == 0) return i+2;
      if (s->chunk[3] == 0) return i+3;
    }
  }
}
```

Strlen Examples – Lifting Constructors

String C memory layouts

- Null character terminated character array…

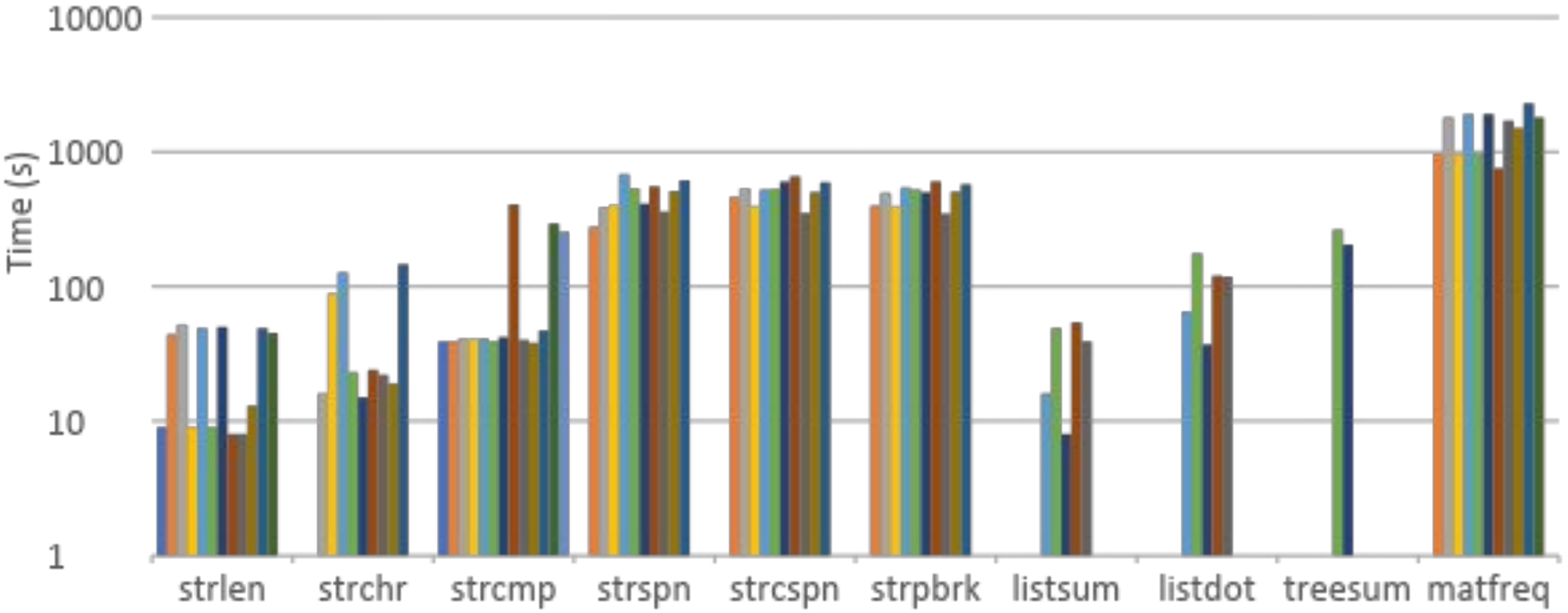
$$Cstr_{\mathbf{m}}(\text{char}^{*}s) \triangleq \underline{\text{if}}\ (s[0] = 0)\ \underline{\text{then}}\ Nil\ \underline{\text{else}}\ Cons\big(s[0], Cstr_{\mathbf{m}}(s+1)\big)$$

- Null character terminated chunked linked list…

$$Cstr'_{\mathbf{m}}(\text{lnode}^{*}s, \text{i2}\ i) \triangleq$$
$$\underline{\text{if}}\ (s \to chunk[i] = 0)\ \underline{\text{then}}\ Nil$$
$$\underline{\text{else}}\ Cons\big(s \to chunk[i], Cstr'_{\mathbf{m}}(i = 3\ ?\ s \to next : s, i+1)\big)$$

Results

~90 equivalence checks – max time of 38 minutes



String Lifting Constructors

Lifting Constructor	Definition
	(T1) `Str = SInvalid | SNil | SCons(i8, Str)`
$\texttt{Cstr}_{\texttt{m}}^{\texttt{u8[]}}(p:\texttt{i32})$	$\underline{\texttt{if}}\ p = 0_{\texttt{i32}}\ \underline{\texttt{then}}\ \texttt{SInvalid}$ $\underline{\texttt{elif}}\ p[0_{\texttt{i32}}]_{\texttt{m}}^{i8} = 0_{\texttt{i8}}\ \underline{\texttt{then}}\ \texttt{SNil}$ $\underline{\texttt{else}}\ \texttt{SCons}(p[0_{\texttt{i32}}]_{\texttt{m}}^{i8}, \texttt{Cstr}_{\texttt{m}}^{\texttt{u8[]}}(p + 1_{\texttt{i32}}))$
$\texttt{Cstr}_{\texttt{m}}^{\texttt{lnode(u8)}}(p:\texttt{i32})$	$\underline{\texttt{if}}\ p = 0_{\texttt{i32}}\ \underline{\texttt{then}}\ \texttt{SInvalid}$ $\underline{\texttt{elif}}\ p \overset{\texttt{m}}{\rightarrow}_{\texttt{lnode}} \texttt{val} = 0_{\texttt{i8}}\ \underline{\texttt{then}}\ \texttt{SNil}$ $\underline{\texttt{else}}\ \texttt{SCons}(p \overset{\texttt{m}}{\rightarrow}_{\texttt{lnode}} \texttt{val}, \texttt{Cstr}_{\texttt{m}}^{\texttt{lnode(u8)}}(p \overset{\texttt{m}}{\rightarrow}_{\texttt{lnode}} \texttt{next}))$
$\texttt{Cstr}_{\texttt{m}}^{\texttt{clnode(u8)}}(p:\texttt{i32}, i:\texttt{i2})$	$\underline{\texttt{if}}\ p = 0_{\texttt{i32}}\ \underline{\texttt{then}}\ \texttt{SInvalid}$ $\underline{\texttt{elif}}\ p \overset{\texttt{m}}{\rightarrow}_{\texttt{lnode}} \texttt{chunk}[i]_{\texttt{m}}^{i8} = 0_{\texttt{i8}}\ \underline{\texttt{then}}\ \texttt{SNil}$ $\underline{\texttt{else}}\ \texttt{SCons}(p \overset{\texttt{m}}{\rightarrow}_{\texttt{lnode}} \texttt{chunk}[i]_{\texttt{m}}^{i8}, \texttt{Cstr}_{\texttt{m}}^{\texttt{clnode(u8)}}(i = 3_{\texttt{i2}} ? p \overset{\texttt{m}}{\rightarrow}_{\texttt{clnode}} \texttt{next} : p, i + 1_{\texttt{i2}}))$

List & Tree Lifting Constructors

Lifting Constructor	Definition
(T2) `List = LNil | LCons(i32, List)`	
$\texttt{Clist}_{\texttt{m}}^{\texttt{u32[]}}(p\ i\ n{:}\texttt{i32})$	$\underline{\texttt{if}}\ i \geq_u n\ \ \underline{\texttt{then}}\ \texttt{LNil}$ $\underline{\texttt{else}}\ \texttt{LCons}(p[i]_{\texttt{m}}^{i32}, \texttt{Clist}_{\texttt{m}}^{\texttt{u32[]}}(p, i + 1_{\texttt{i32}}, n))$
$\texttt{Clist}_{\texttt{m}}^{\texttt{lnode(u32)}}(p{:}\texttt{i32})$	$\underline{\texttt{if}}\ p = 0_{\texttt{i32}}\ \ \underline{\texttt{then}}\ \texttt{LNil}$ $\underline{\texttt{else}}\ \texttt{LCons}(p \overset{\texttt{m}}{\rightarrow}_{\texttt{lnode}} \texttt{val}, \texttt{Clist}_{\texttt{m}}^{\texttt{lnode}}(p \overset{\texttt{m}}{\rightarrow}_{\texttt{lnode}} \texttt{next}))$
$\texttt{Clist}_{\texttt{m}}^{\texttt{clnode(u32)}}(p{:}\texttt{i32}, i{:}\texttt{i2})$	$\underline{\texttt{if}}\ p = 0_{\texttt{i32}}\ \ \underline{\texttt{then}}\ \texttt{LNil}$ $\underline{\texttt{else}}\ \texttt{LCons}(p \overset{\texttt{m}}{\rightarrow}_{\texttt{clnode}} \texttt{chunk}[i]_{\texttt{m}}^{i32}, \texttt{Clist}_{\texttt{m}}^{\texttt{clnode}}(i = 3_{\texttt{i2}}?p \overset{\texttt{m}}{\rightarrow}_{\texttt{clnode}} \texttt{next} : p, i + 1_{\texttt{i2}}))$
(T3) `Tree = TNil | TCons(i32, Tree, Tree)`	
$\texttt{Ctree}_{\texttt{m}}^{\texttt{u32[]}}(p\ i\ n{:}\texttt{i32})$	$\underline{\texttt{if}}\ i \geq_u n\ \ \underline{\texttt{then}}\ \texttt{TNil}$ $\underline{\texttt{else}}\ \texttt{TCons}(p[i]_{\texttt{m}}^{i32}, \texttt{Ctree}_{\texttt{m}}^{\texttt{u32[]}}(p, 2_{\texttt{i32}} \times i + 1_{\texttt{i32}}, n), \texttt{Ctree}_{\texttt{m}}^{\texttt{u32[]}}(p, 2_{\texttt{i32}} \times i + 2_{\texttt{i32}}, n))$
$\texttt{Ctree}_{\texttt{m}}^{\texttt{tnode(u32)}}(p{:}\texttt{i32})$	$\underline{\texttt{if}}\ p = 0_{\texttt{i32}}\ \ \underline{\texttt{then}}\ \texttt{TNil}$ $\underline{\texttt{else}}\ \texttt{TCons}(p \overset{\texttt{m}}{\rightarrow}_{\texttt{tnode}} \texttt{val}, \texttt{Ctree}_{\texttt{m}}^{\texttt{tnode(u32)}}(p \overset{\texttt{m}}{\rightarrow}_{\texttt{tnode}} \texttt{left}), \texttt{Ctree}_{\texttt{m}}^{\texttt{tnode(u32)}}(p \overset{\texttt{m}}{\rightarrow}_{\texttt{tnode}} \texttt{right}))$

Matrix Lifting Constructors

Lifting Constructor	Definition
(T4) Matrix = MNil | MCons(List, Matrix)	
$\texttt{Cmat}_{\mathtt{m}}^{\texttt{u32[][]}}(p\ i\ u\ v{:}\texttt{i32})$	$\underline{\texttt{if}}\ i \geq_u u\ \ \underline{\texttt{then}}\ \texttt{MNil}$ $\underline{\texttt{else}}\ \texttt{MCons}(\texttt{Clist}_{\mathtt{m}}^{\texttt{u32[]}}(p[i]_{\mathtt{m}}^{i32}, 0_{\texttt{i32}}, v), \texttt{Cmat}_{\mathtt{m}}^{\texttt{u32[][]}}(p, i+1_{\texttt{i32}}, u, v))$
$\texttt{Clist}_{\mathtt{m}}^{\texttt{u32[r]}}(p\ i\ j\ u\ v{:}\texttt{i32})$	$\underline{\texttt{if}}\ j \geq_u v\ \ \underline{\texttt{then}}\ \texttt{LNil}$ $\underline{\texttt{else}}\ \texttt{LCons}(p[i \times v + j]_{\mathtt{m}}^{i32}, \texttt{Clist}_{\mathtt{m}}^{\texttt{u32[r]}}(p, i, j+1_{\texttt{i32}}, u, v))$
$\texttt{Cmat}_{\mathtt{m}}^{\texttt{u32[r]}}(p\ i\ u\ v{:}\texttt{i32})$	$\underline{\texttt{if}}\ i \geq_u u\ \ \underline{\texttt{then}}\ \texttt{MNil}$ $\underline{\texttt{else}}\ \texttt{MCons}(\texttt{Clist}_{\mathtt{m}}^{\texttt{u32[r]}}(p, i, 0_{\texttt{i32}}, u, v), \texttt{Cmat}_{\mathtt{m}}^{\texttt{u32[r]}}(p, i+1_{\texttt{i32}}, u, v))$
$\texttt{Clist}_{\mathtt{m}}^{\texttt{u32[c]}}(p\ i\ j\ u\ v{:}\texttt{i32})$	$\underline{\texttt{if}}\ j \geq_u v\ \ \underline{\texttt{then}}\ \texttt{LNil}$ $\underline{\texttt{else}}\ \texttt{LCons}(p[i + j \times u]_{\mathtt{m}}^{i32}, \texttt{Clist}_{\mathtt{m}}^{\texttt{u32[c]}}(p, i, j+1_{\texttt{i32}}, u, v))$
$\texttt{Cmat}_{\mathtt{m}}^{\texttt{u32[c]}}(p\ i\ u\ v{:}\texttt{i32})$	$\underline{\texttt{if}}\ i \geq_u u\ \ \underline{\texttt{then}}\ \texttt{MNil}$ $\underline{\texttt{else}}\ \texttt{MCons}(\texttt{Clist}_{\mathtt{m}}^{\texttt{u32[c]}}(p, i, 0_{\texttt{i32}}, u, v), \texttt{Cmat}_{\mathtt{m}}^{\texttt{u32[c]}}(p, i+1_{\texttt{i32}}, u, v))$
$\texttt{Cmat}_{\mathtt{m}}^{\texttt{lnode(u32[])}}(p\ v{:}\texttt{i32})$	$\underline{\texttt{if}}\ p = 0_{\texttt{i32}}\ \underline{\texttt{then}}\ \texttt{MNil}$ $\underline{\texttt{else}}\ \texttt{MCons}(\texttt{Clist}_{\mathtt{m}}^{\texttt{u32[]}}(p \xrightarrow{\mathtt{m}}_{\texttt{lnode}} \texttt{val}, 0_{\texttt{i32}}, v), \texttt{Cmat}_{\mathtt{m}}^{\texttt{lnode(u32[])}}(p \xrightarrow{\mathtt{m}}_{\texttt{lnode}} \texttt{next}, v))$
$\texttt{Cmat}_{\mathtt{m}}^{\texttt{lnode(u32)[]}}(p\ i\ u{:}\texttt{i32})$	$\underline{\texttt{if}}\ i \geq_u u\ \underline{\texttt{then}}\ \texttt{MNil}$ $\underline{\texttt{else}}\ \texttt{MCons}(\texttt{Clist}_{\mathtt{m}}^{\texttt{lnode(u32)}}(p[i]_{\mathtt{m}}^{i32}), \texttt{Cmat}_{\mathtt{m}}^{\texttt{lnode(u32)[]}}(p, i+1_{\texttt{i32}}, u))$
$\texttt{Cmat}_{\mathtt{m}}^{\texttt{clnode(u32)}}(p\ i\ u{:}\texttt{i32})$	$\underline{\texttt{if}}\ i \geq_u u\ \underline{\texttt{then}}\ \texttt{MNil}$ $\underline{\texttt{else}}\ \texttt{MCons}(\texttt{Clist}_{\mathtt{m}}^{\texttt{clnode(u32)}}(p[i]_{\mathtt{m}}^{i32}, 0_{\texttt{i2}}), \texttt{Cmat}_{\mathtt{m}}^{\texttt{clnode(u32)[]}}(p, i+1_{\texttt{i32}}, u))$