Formal Methods Update Meeting 2022

Equivalence Checking as the Backbone of Compiler Development

Sorav Bansal
IIT Delhi and CompilerAI Labs

Joint work with Alex Aiken, Manjeet Dahiya, Shubhani, Abhishek Rose
and several other past members of our research group

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…

Easier to Program ↑ / Performance ↓

- Haskell
- Javascript
- TensorFlow
- Java/Py
- C/C++
- WebAsm
- Asm

Increasing number of Languages

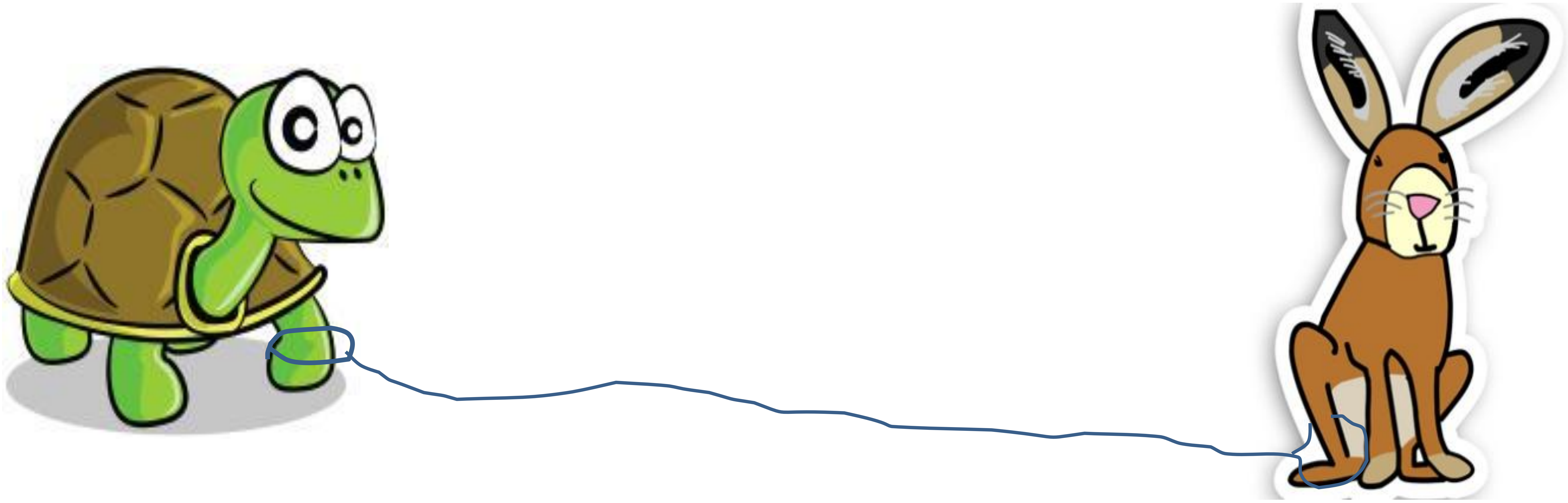
Compiler

Easier to Program ↑ / Performance ↓

- RISC
- CISC
- Manycore Hetro
- GPUs
- SecurityAcc
- Neuromorphic
- FPGAs

Increasing number of Architectures

Architects and Compilers

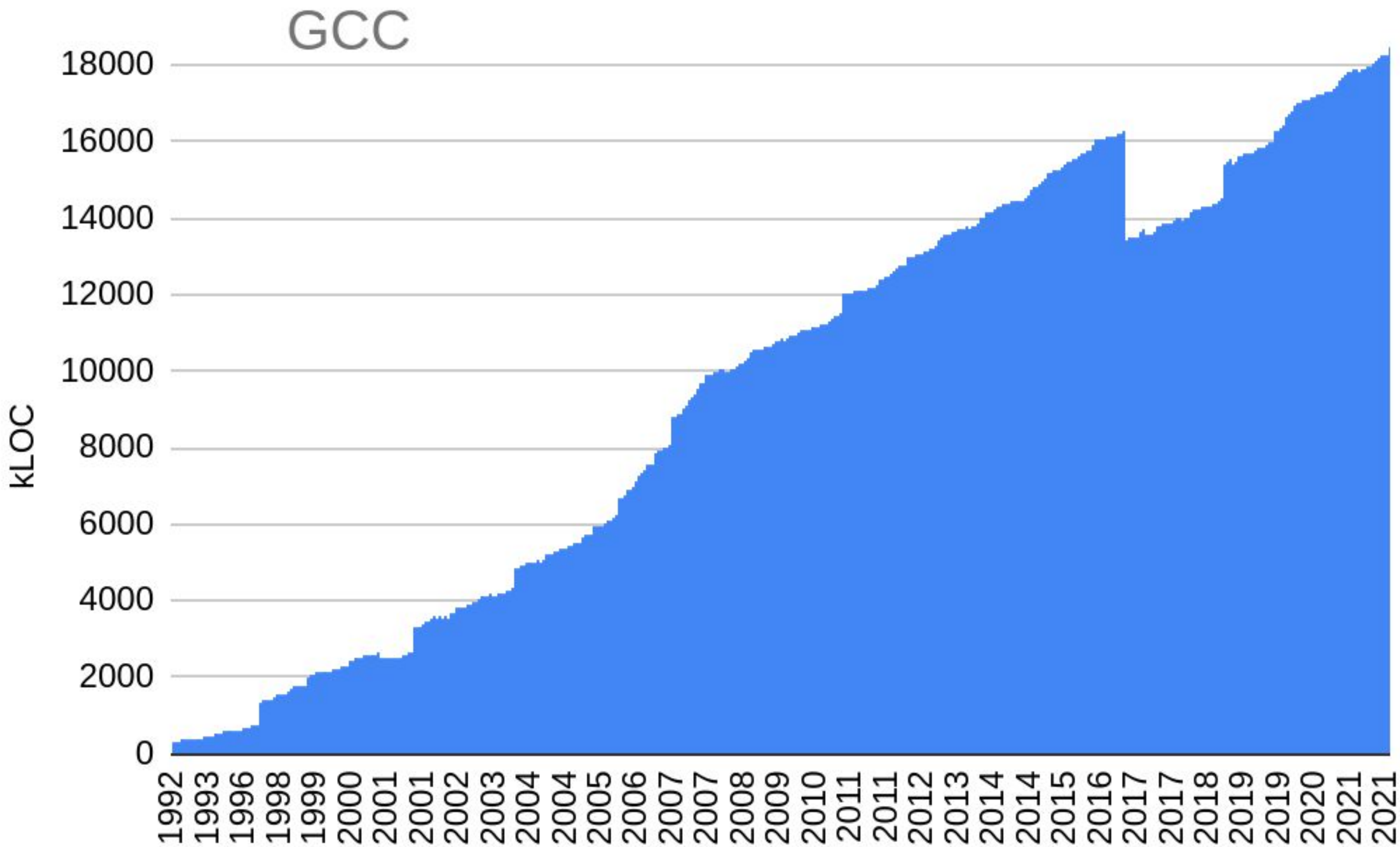


Compilers

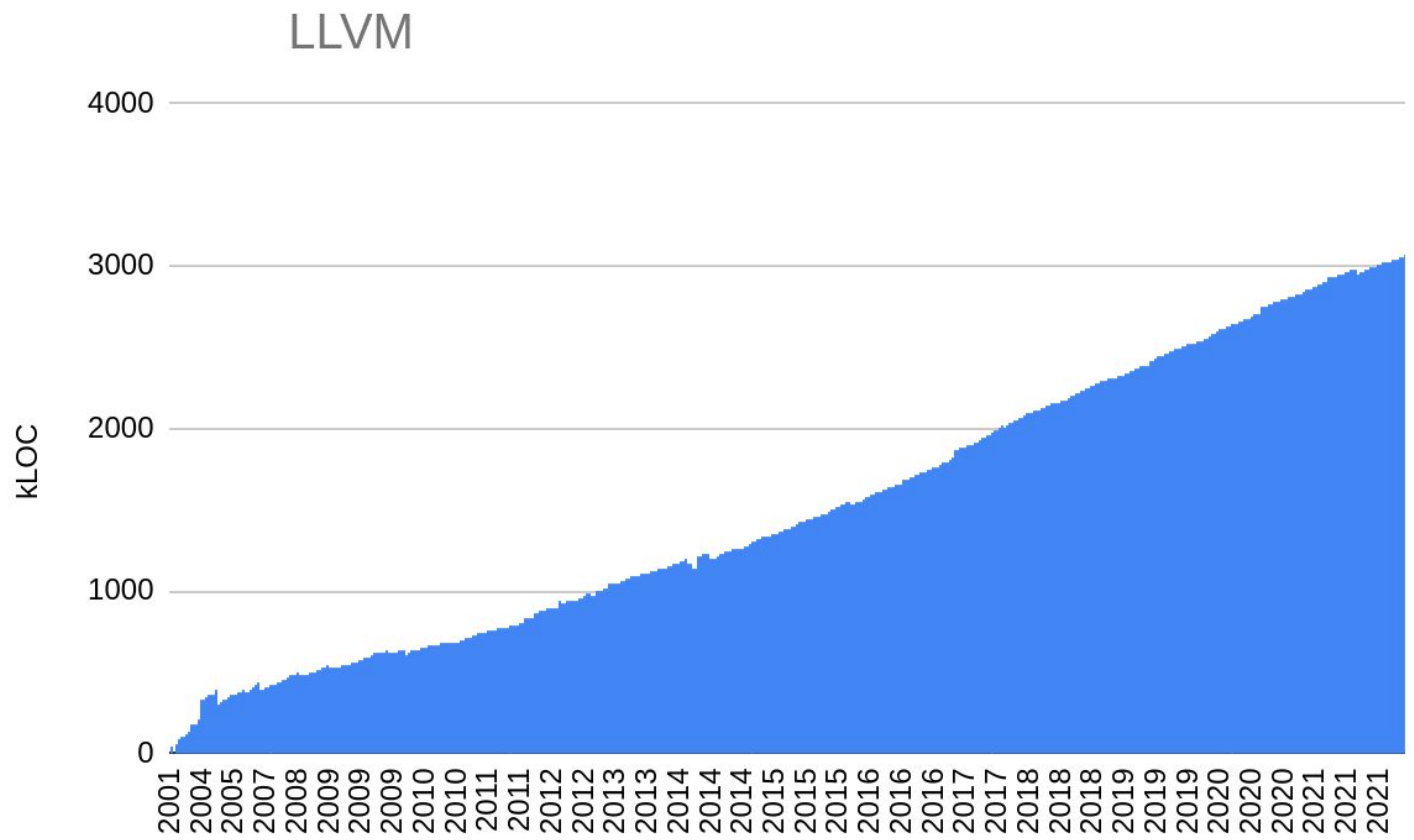
Architects

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

Compiler Complexity Growth

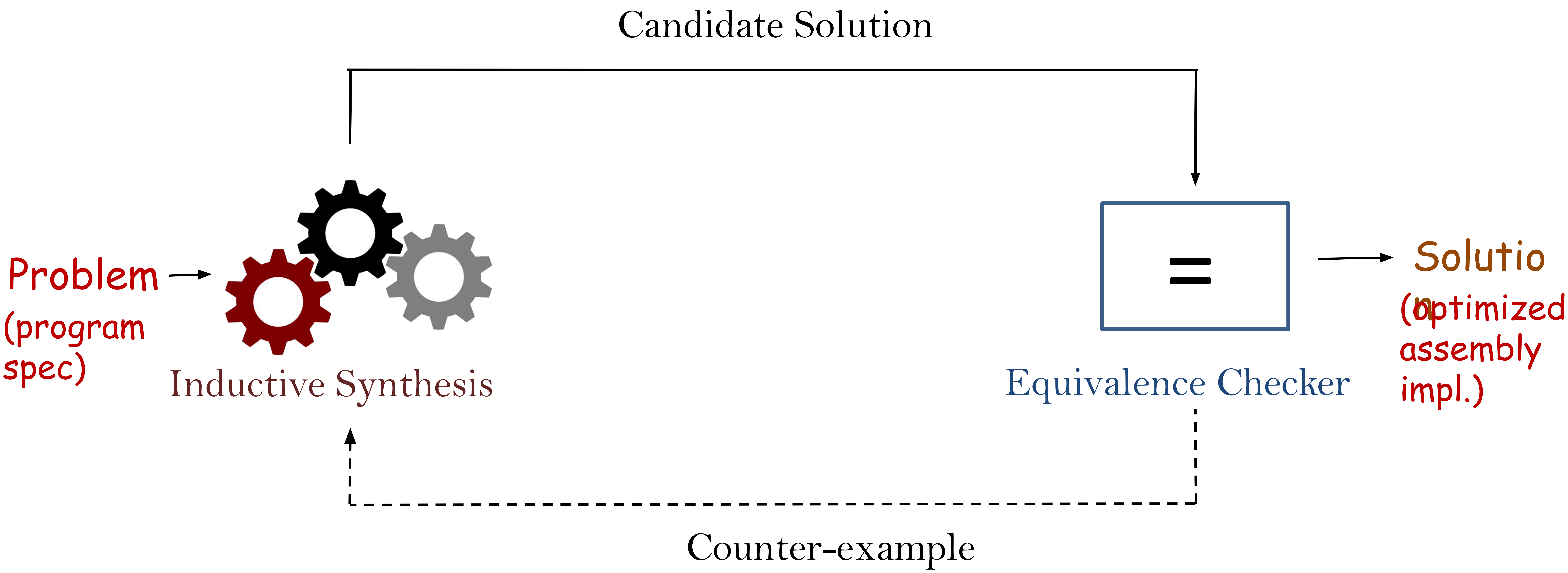


Compiler Complexity Growth



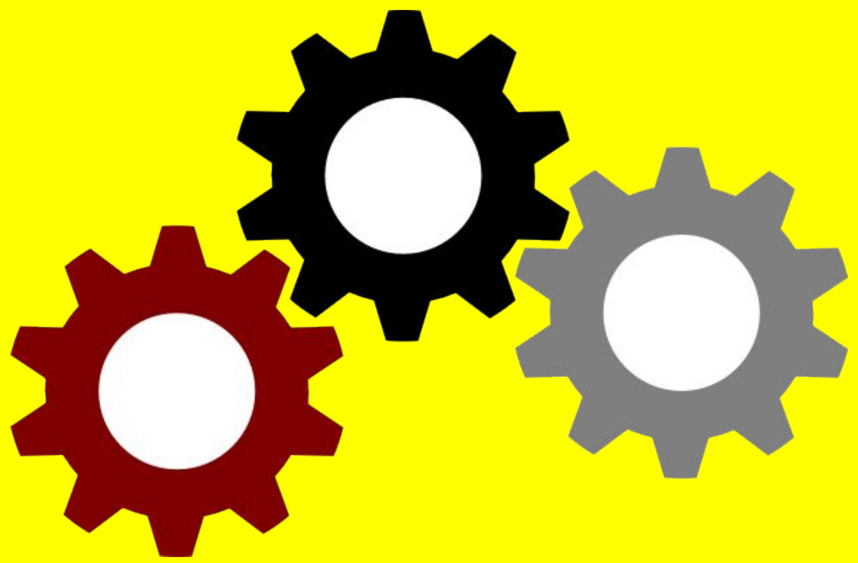
Superoptimization

Superoptimization



Superoptimization

Problem
(program spec)

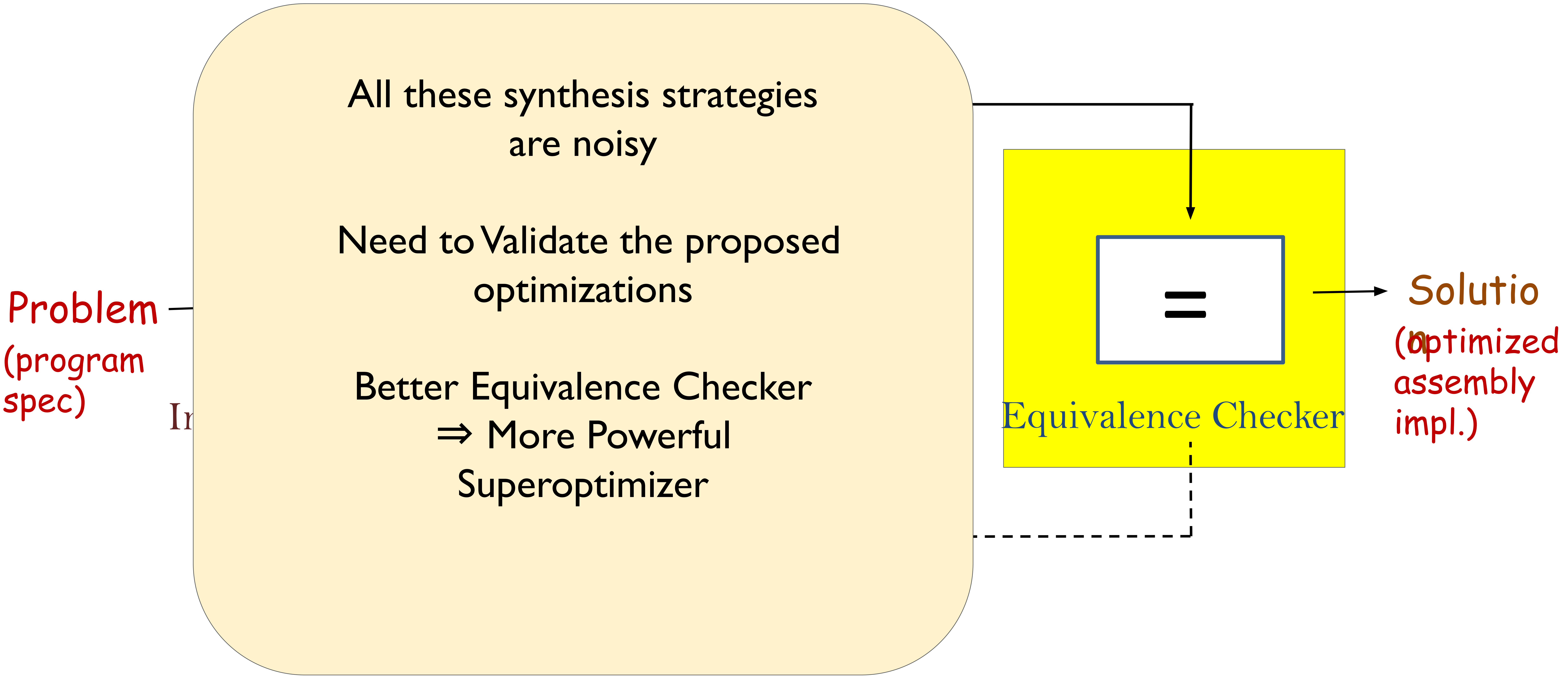


Inductive Synthesis

- Enumeration
- Stochastic Search
- Learning from Examples
- Mutation-based learning
- Constraint Solving
- Neural Networks
- <your favourite fancy technique>
- …

Solution
(optimized assembly impl.)

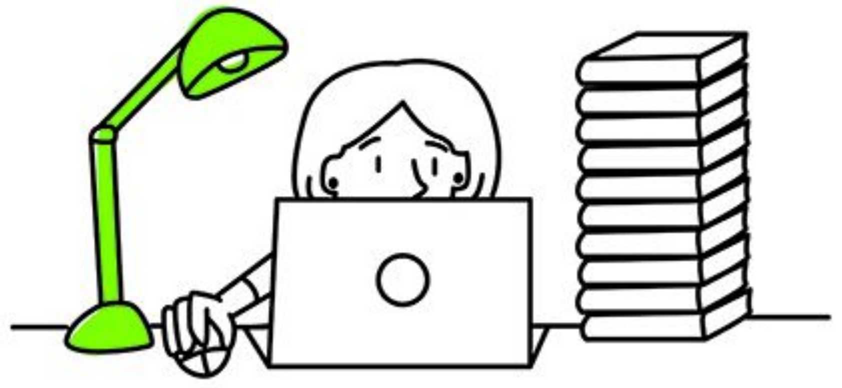
Superoptimization



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

An engineer gets an optimization idea

Codes it up

Reviewed by experts

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

An engineer gets an optimization idea

Codes it up

Reviewed by experts

Too much human effort

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

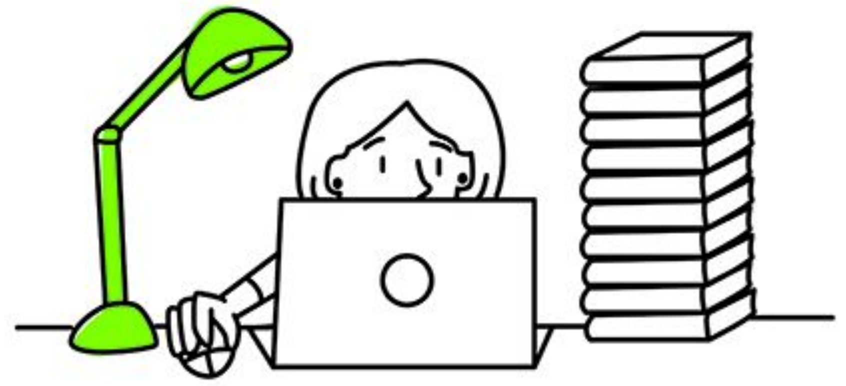
Codes it up

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;`
`}`
	<u>On Motorola 68020</u>:
	`    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*

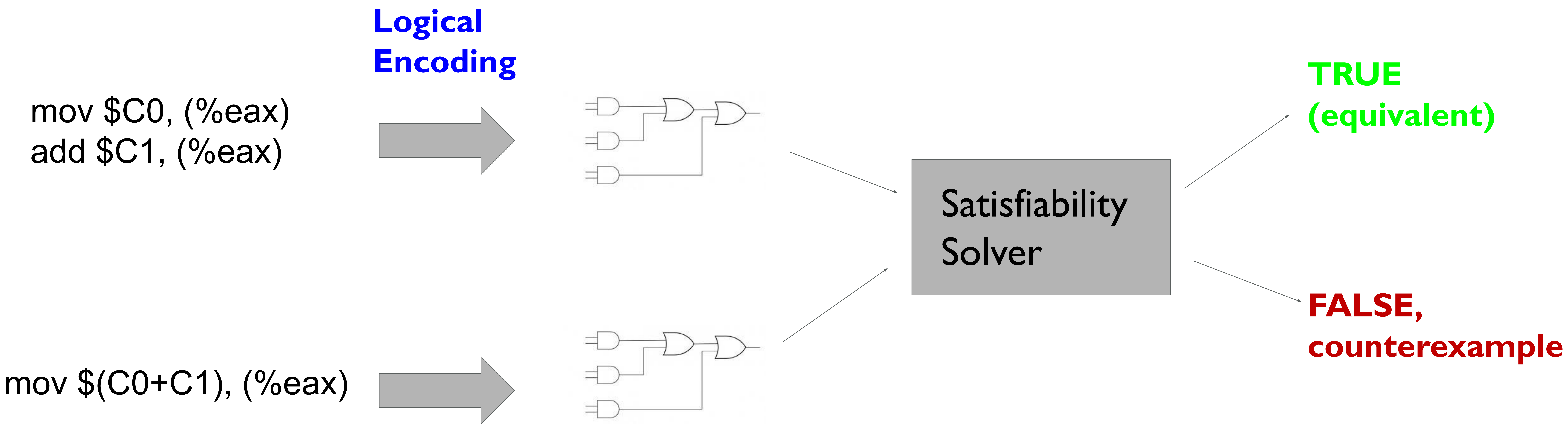
Pattern-Replacement Examples

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

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

replacement

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

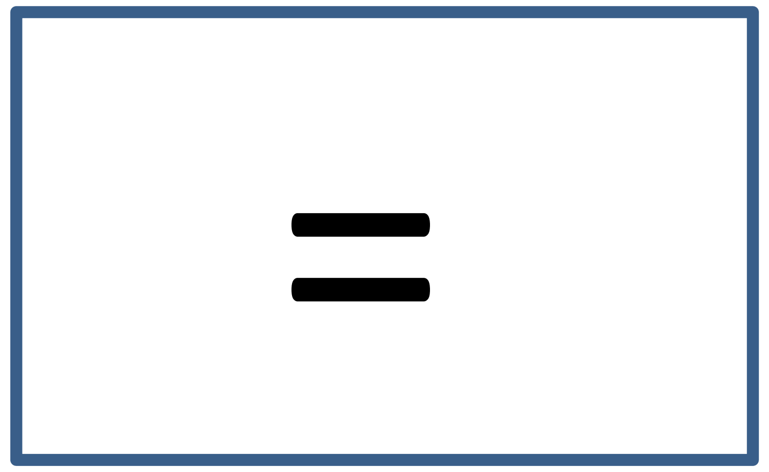
?

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



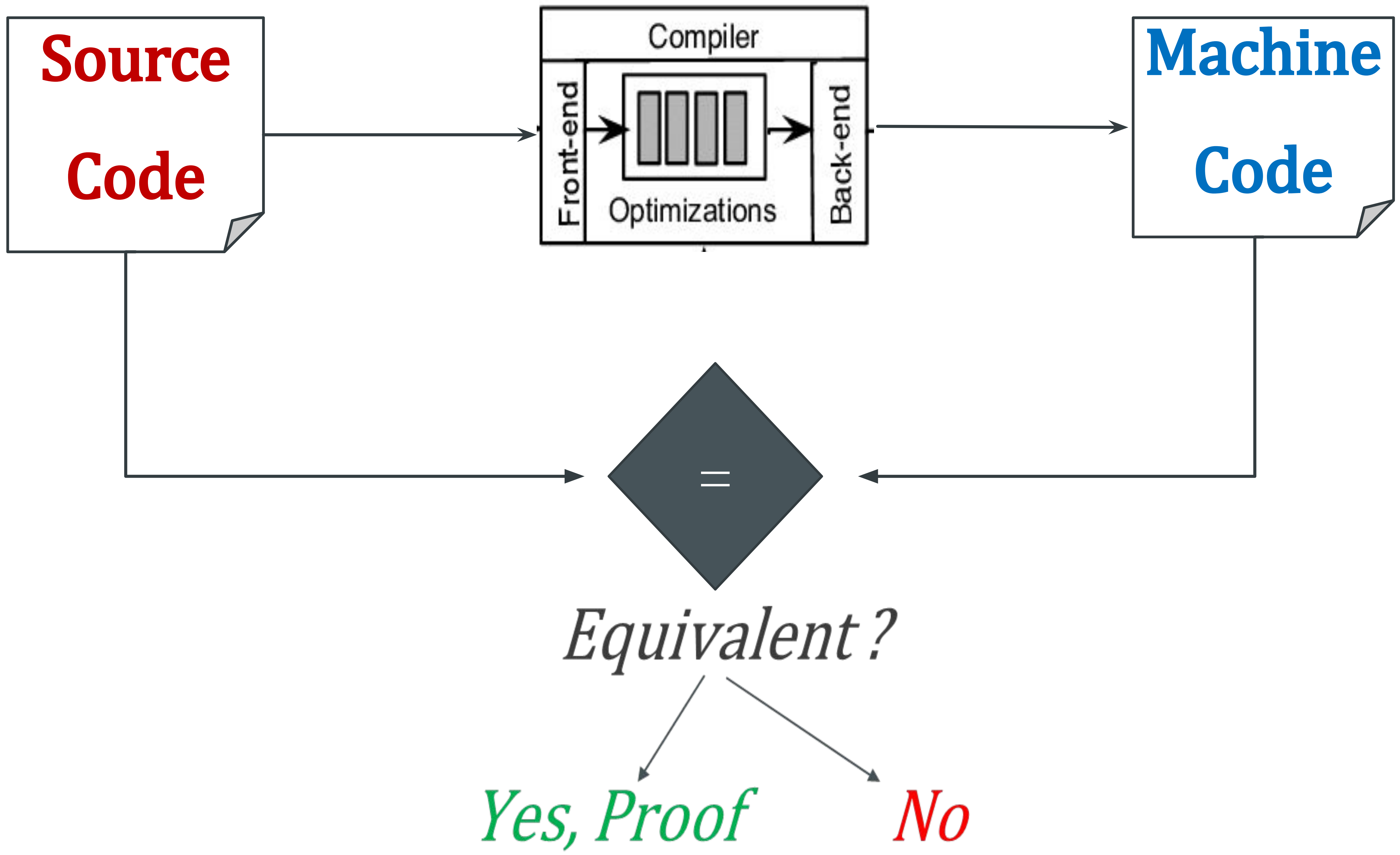
Equivalence Checker

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;
  }
}
```

C Program

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

(abstracted) Assembly

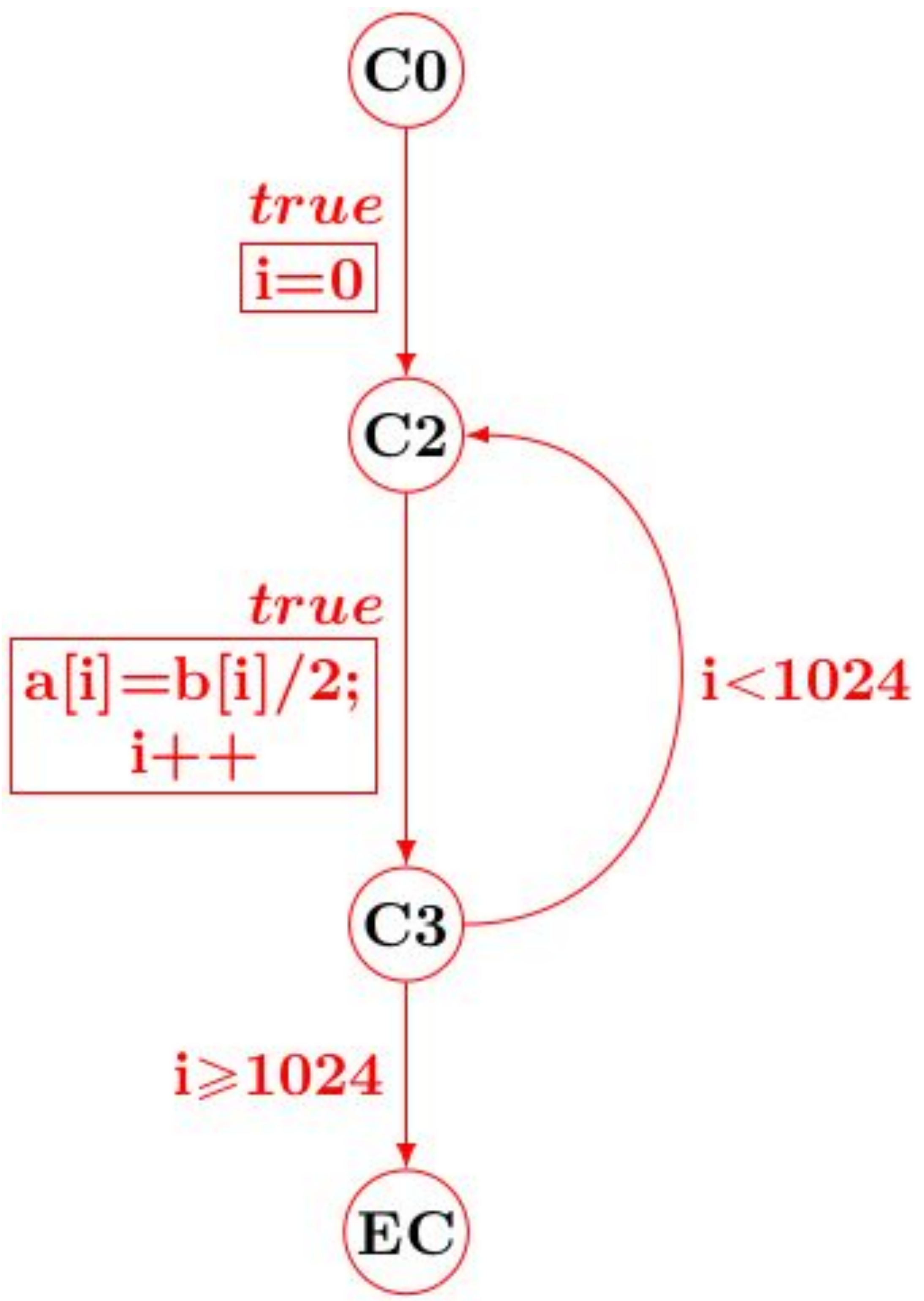
$Return_C == Return_A$

$Memory_C == Memory_A$?

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



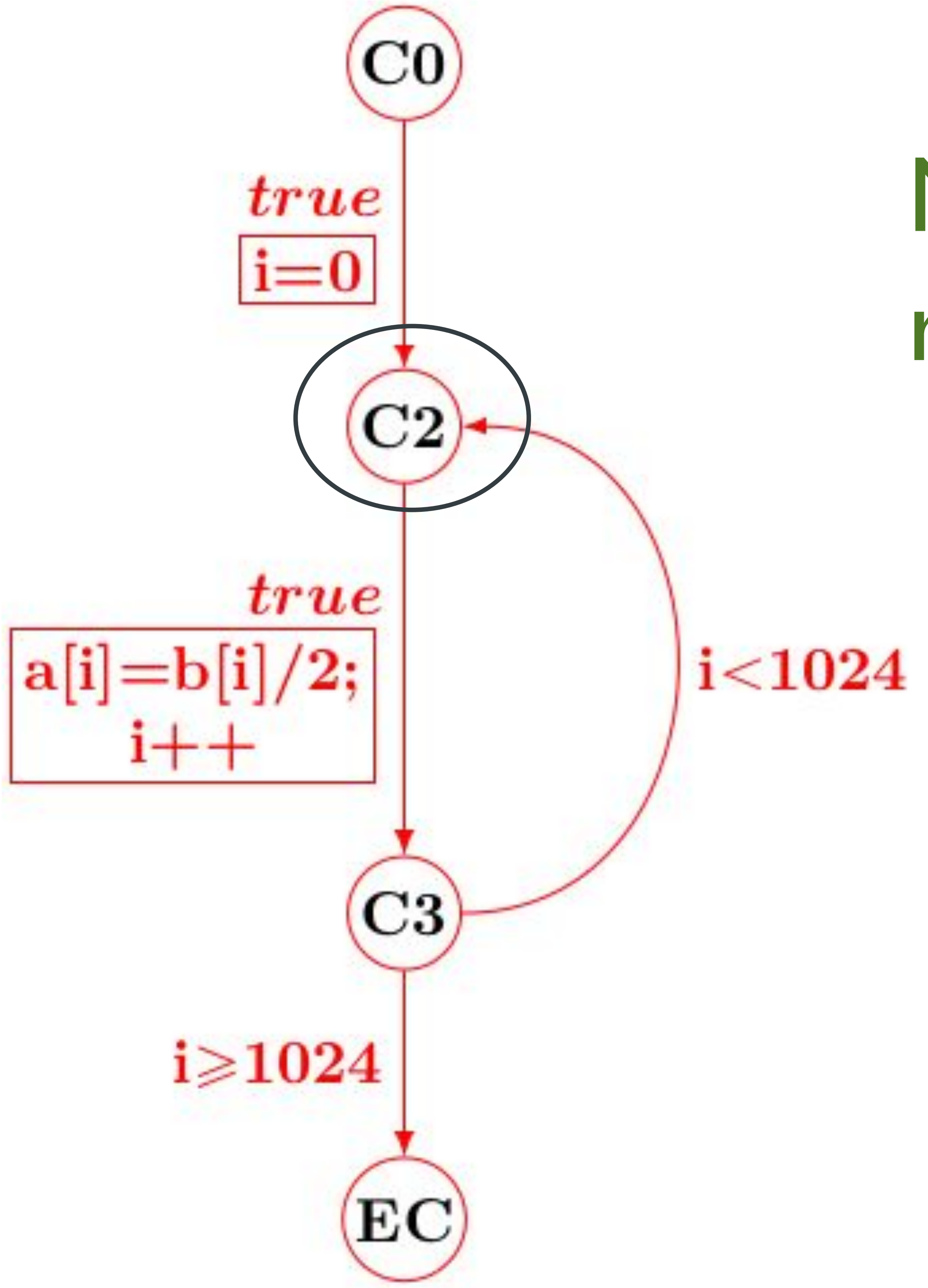
Nodes are PCs

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;) {

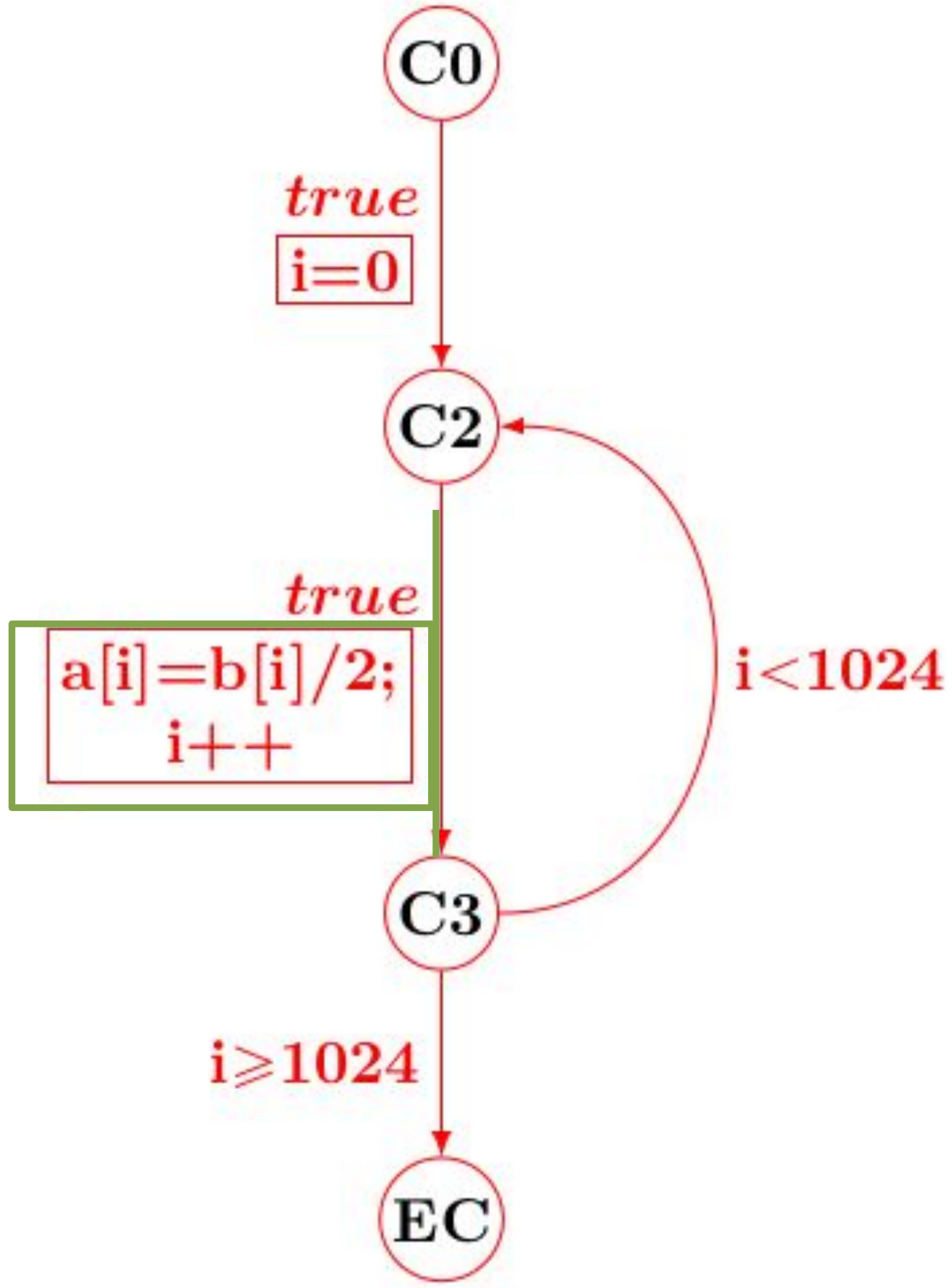
C2: a[i] = b[i]/2; i++;

C3: }

EC: }

C Program

Transfer function of C2→C3



Control Flow Graph (CFG)

C0: void divP ( ) {

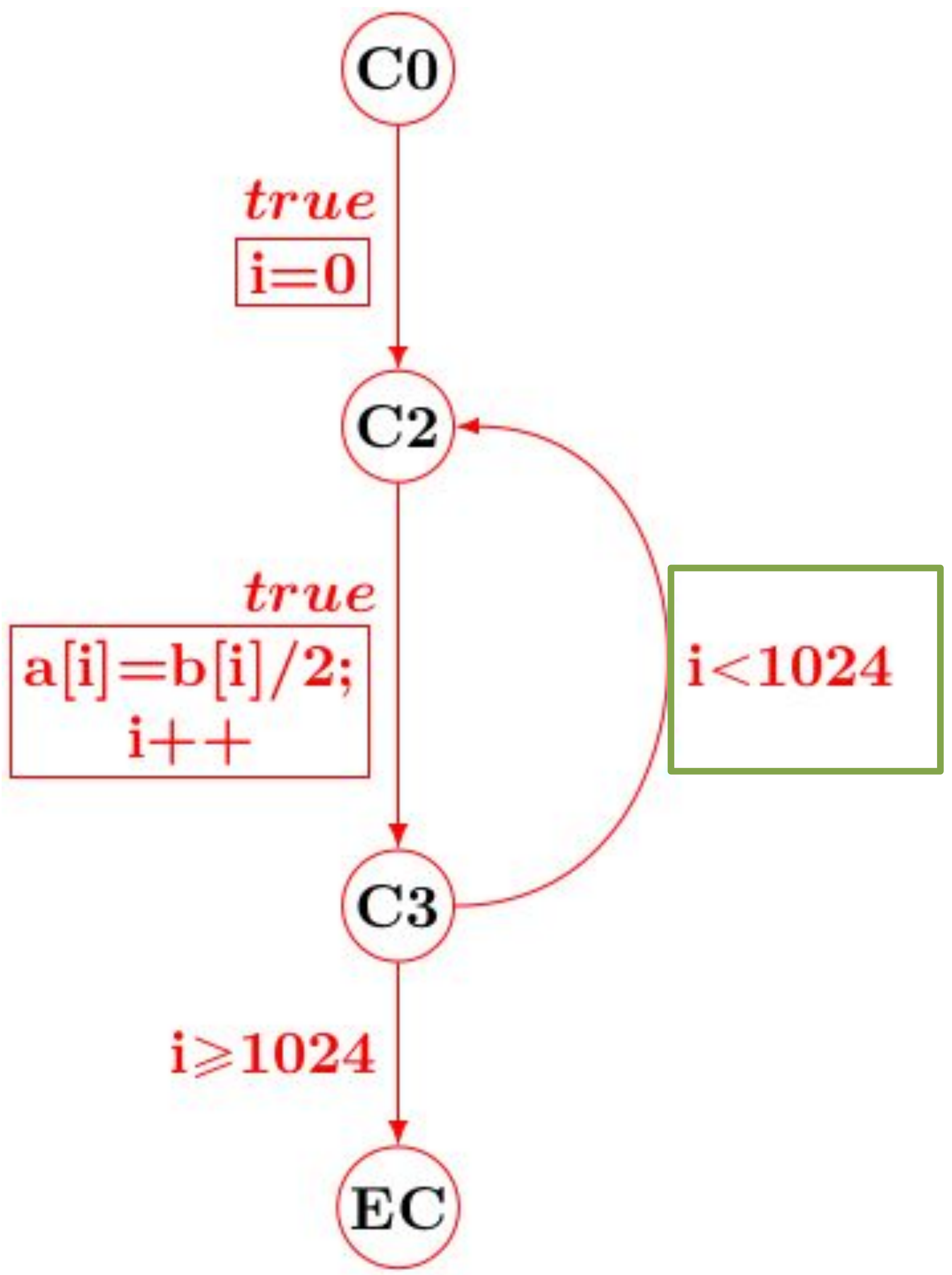
C1: for(int i=0; i < 1024;) {

C2: a[i] = b[i]/2; i++;

C3: }

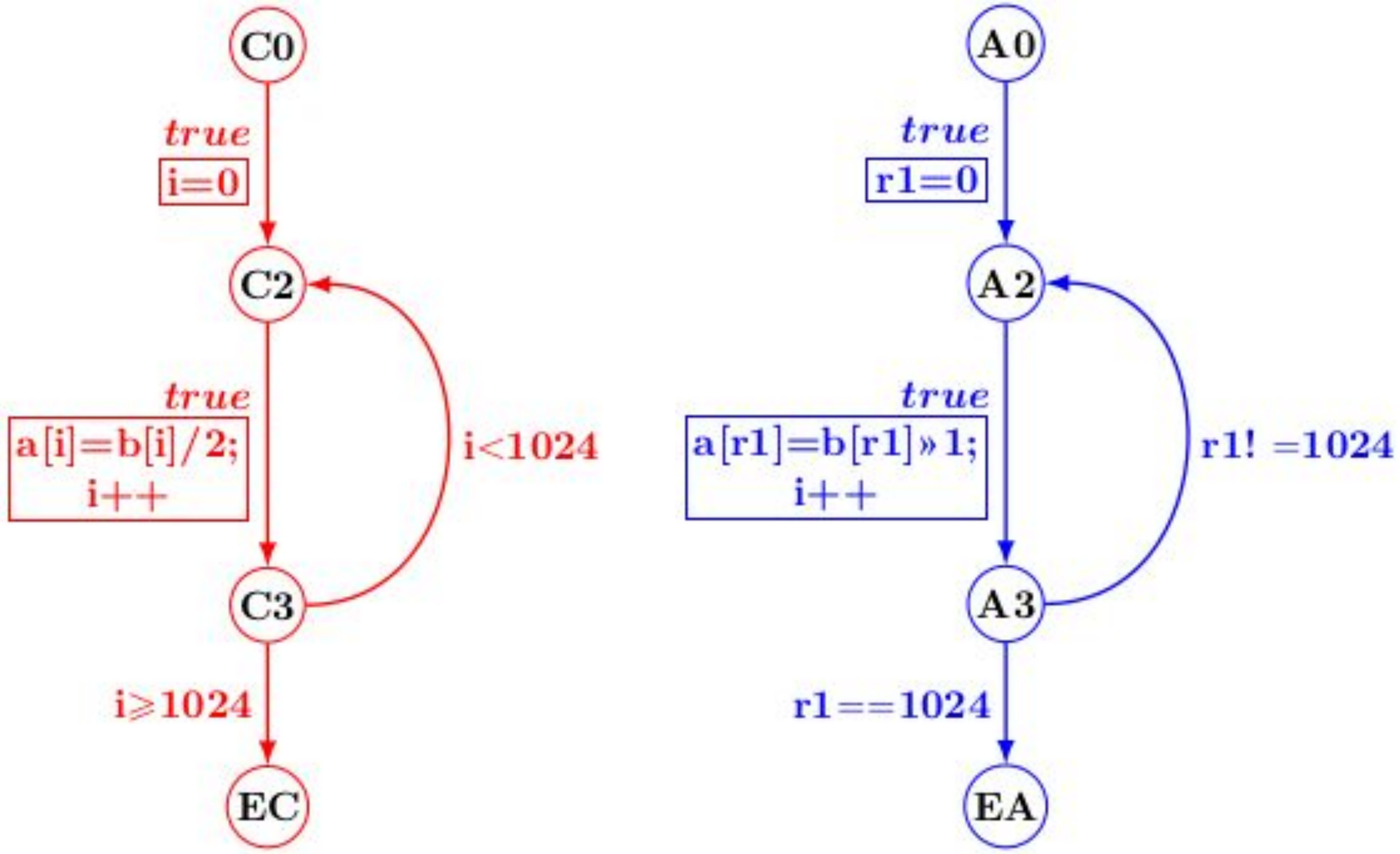
EC: }

C Program



Edge Condition of C3→C2

Product Program Construction

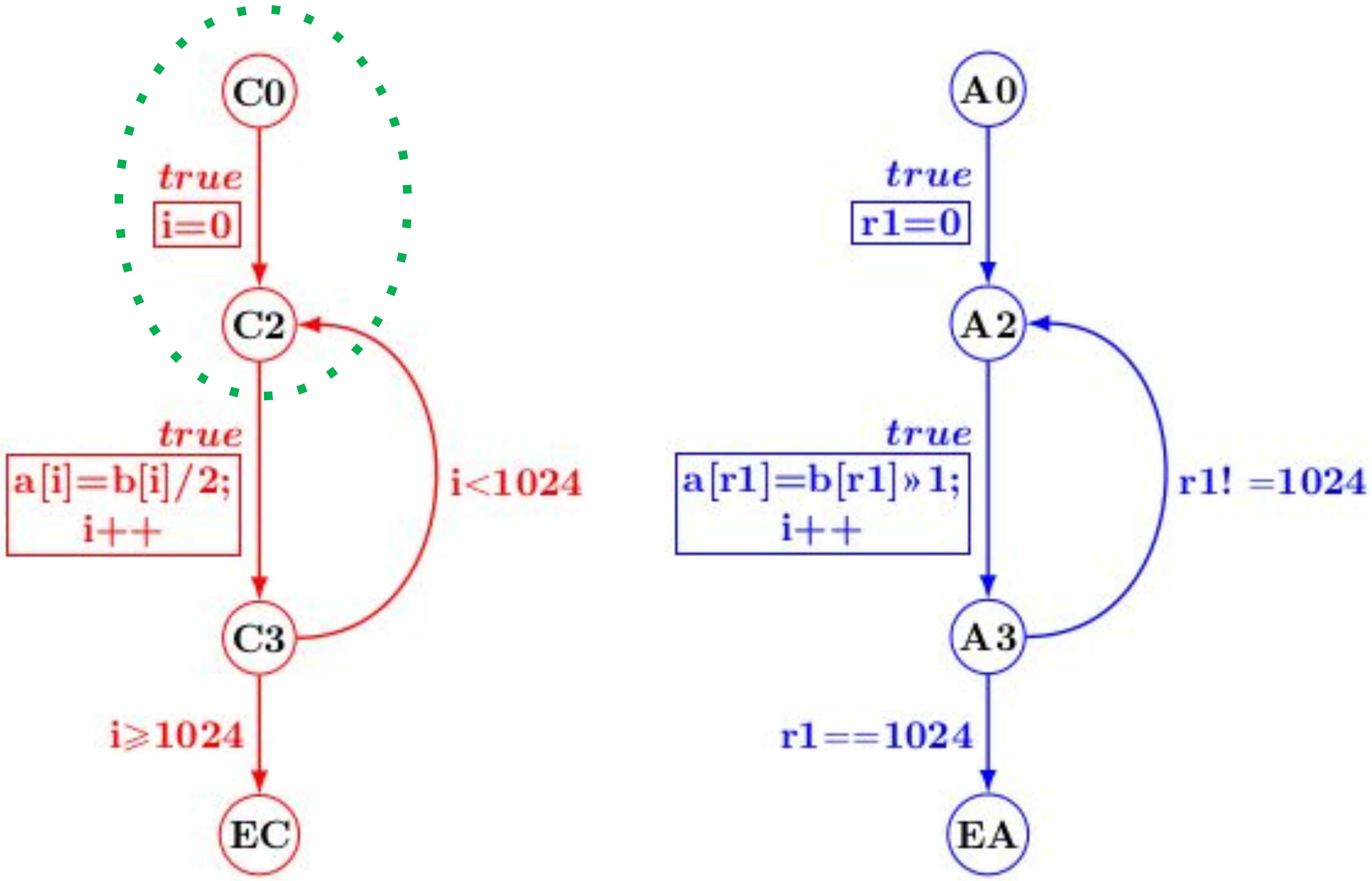


Goal: Construct a **Product Program** that executes both programs in lockstep

such that the two programs' states always remain related.

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

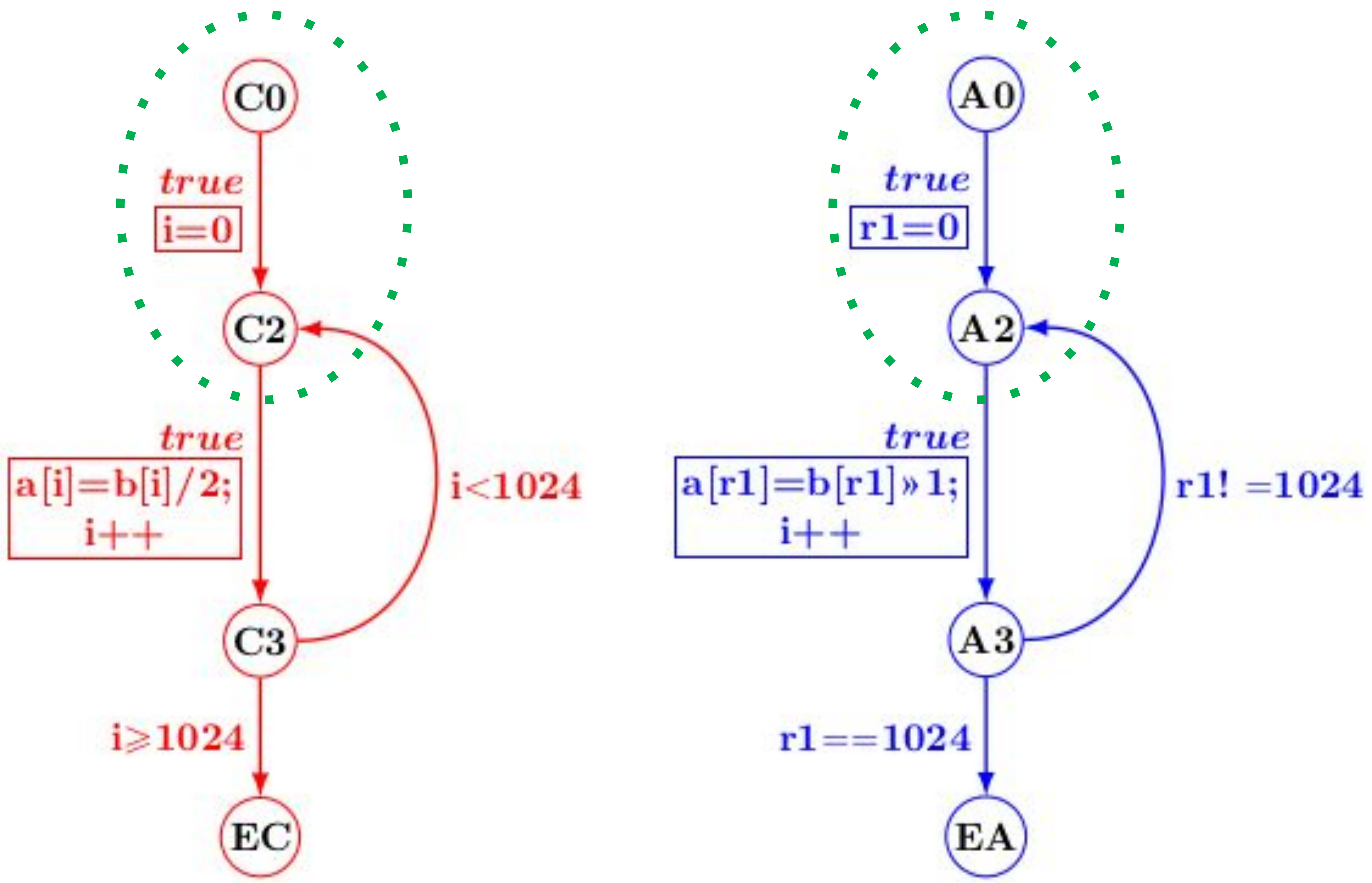
Product Program Construction



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

such that the two programs' states always remain related.

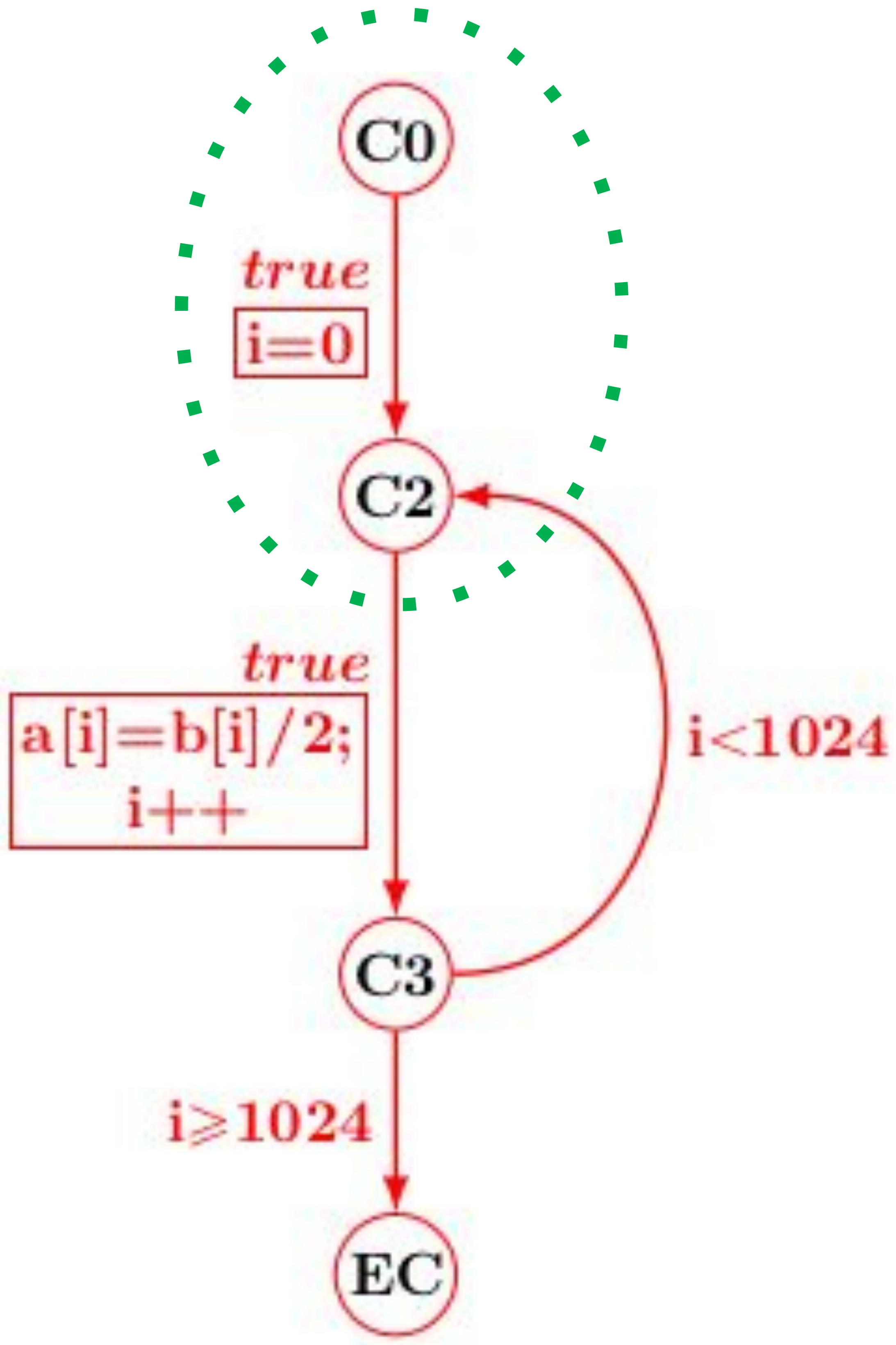
Product Program Construction

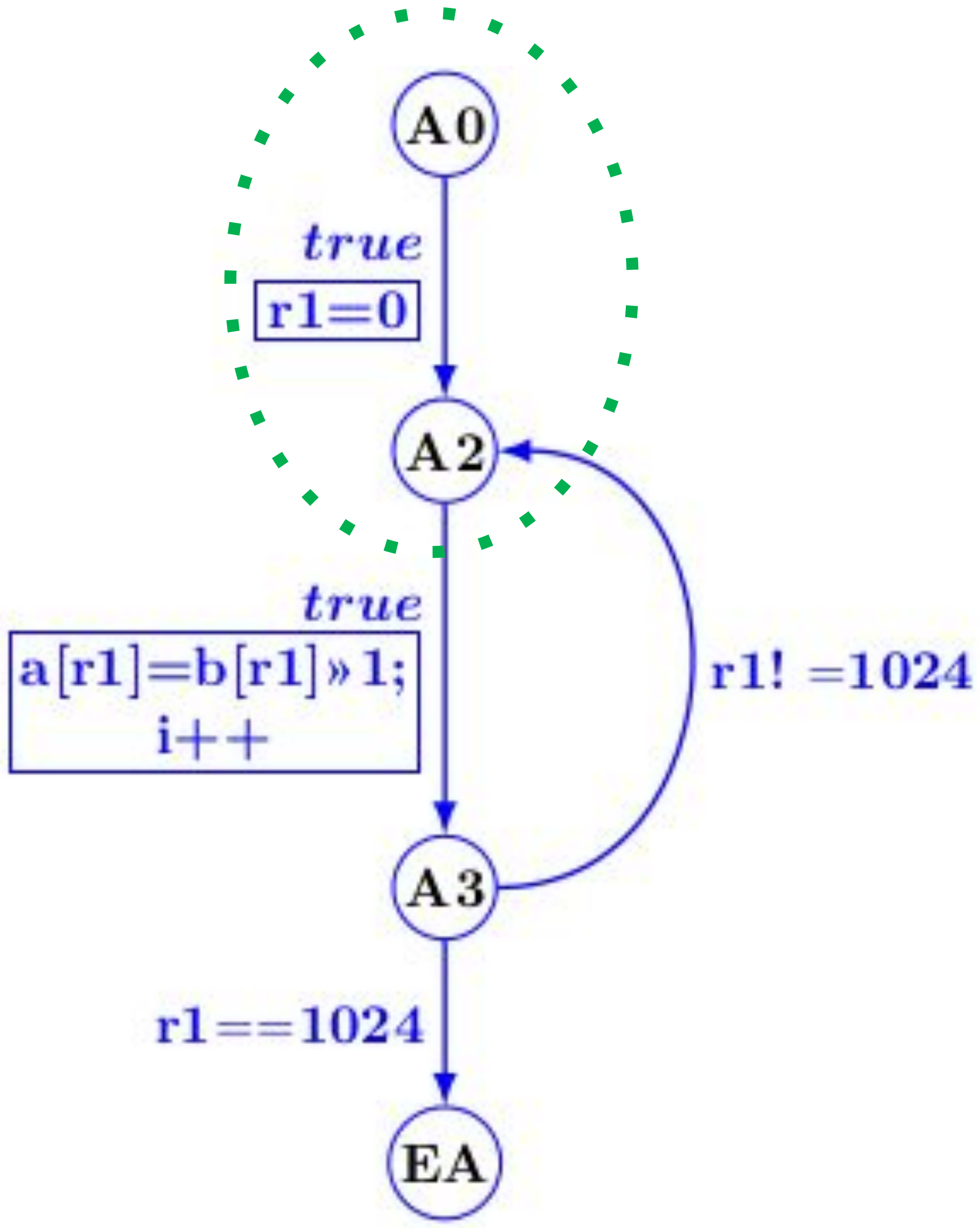


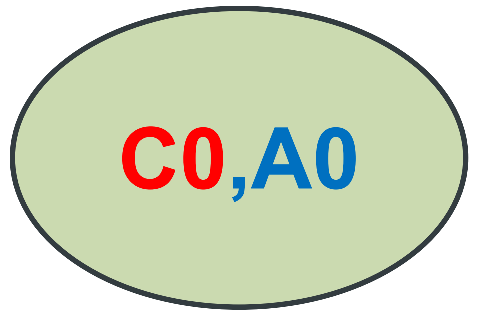
Goal: Construct a **Product Program** that executes both programs in lockstep

such that the two programs' states always remain related.

Product Program Construction

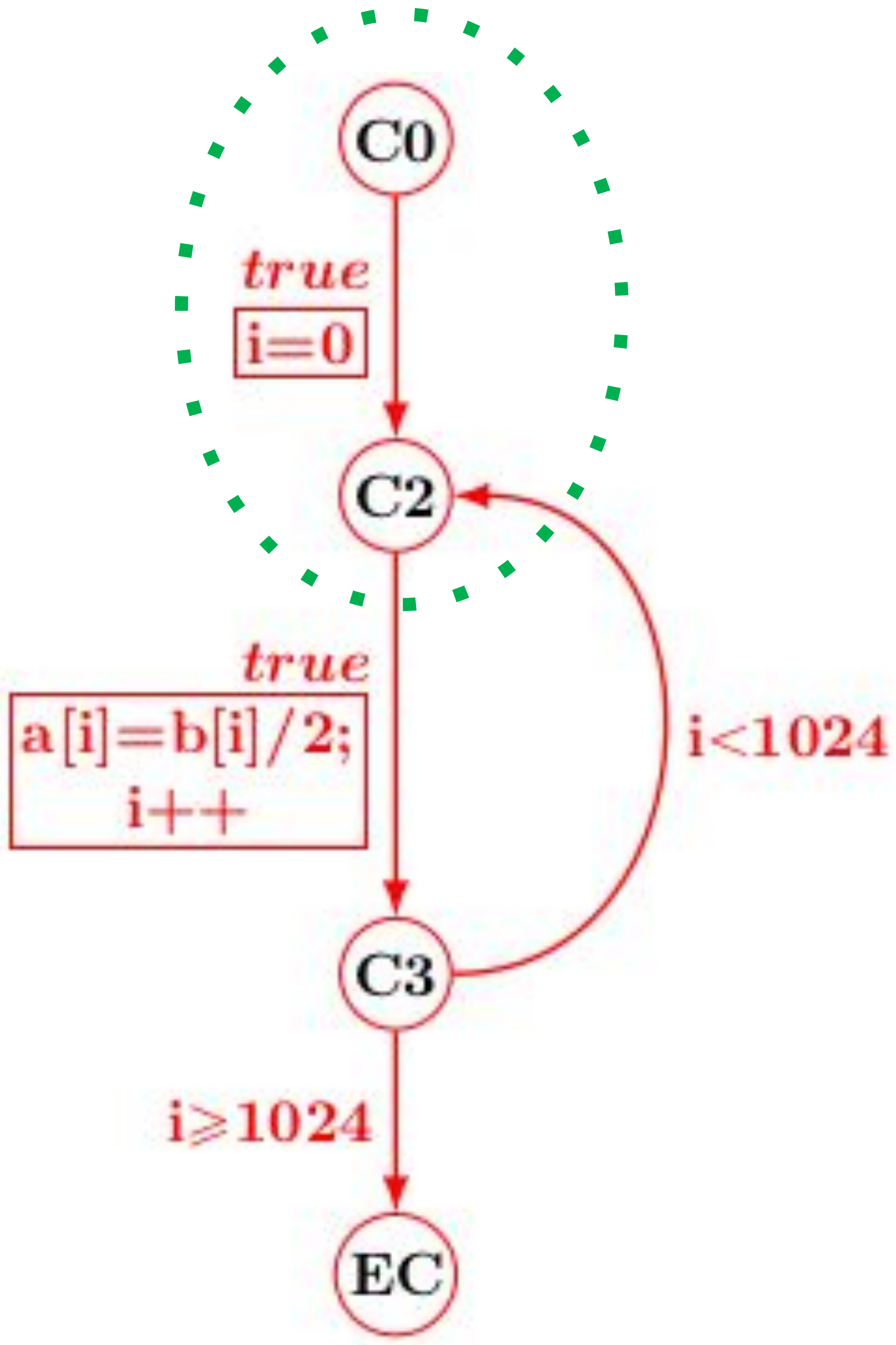


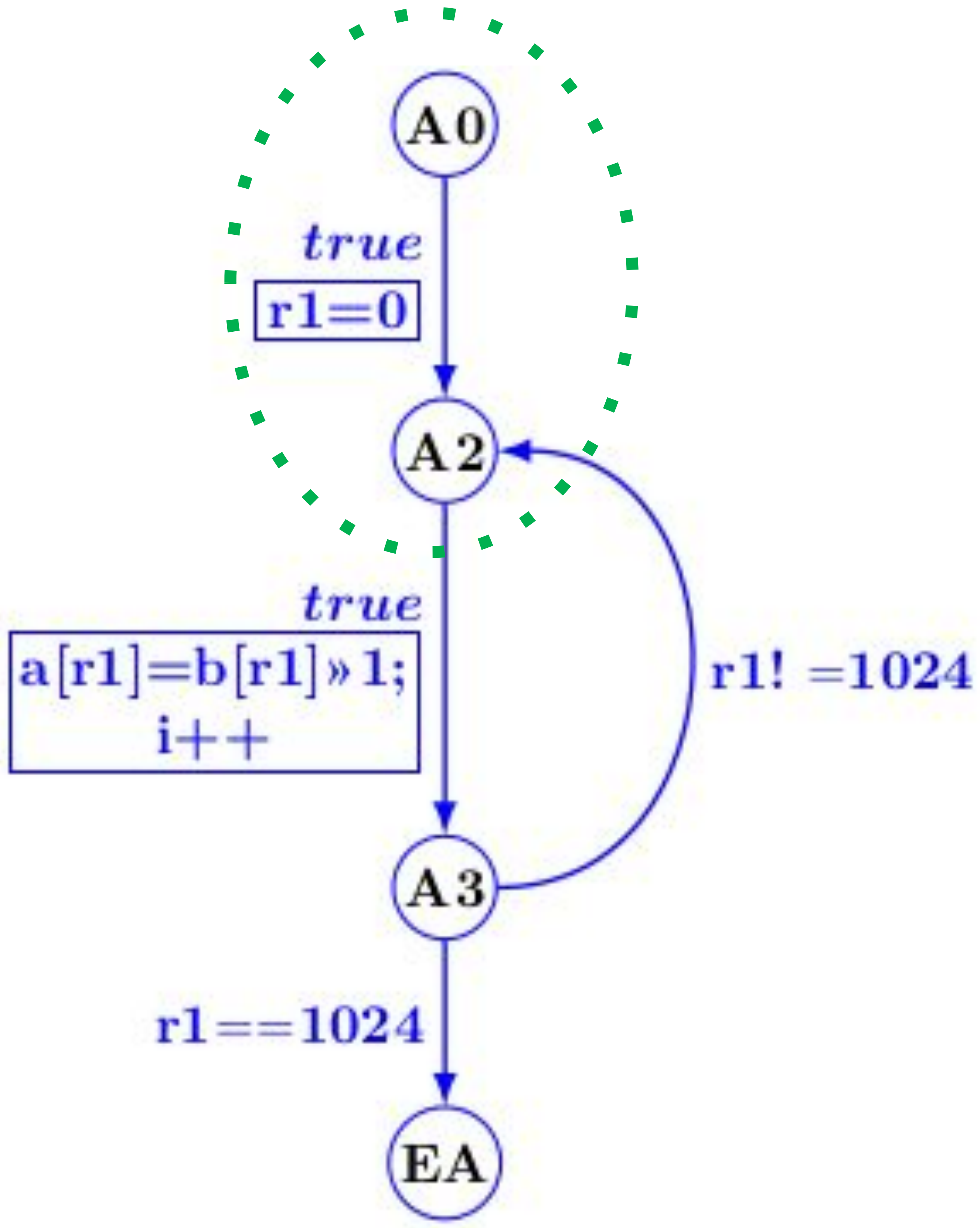


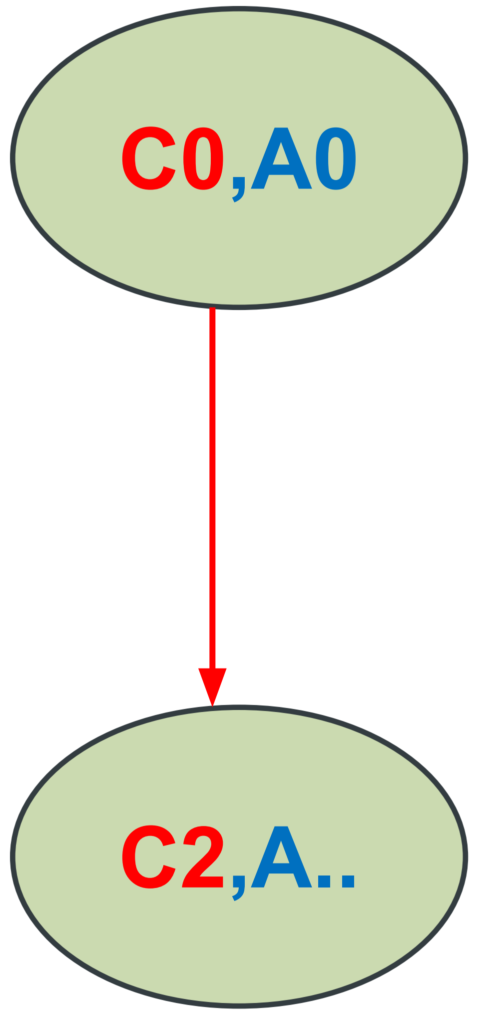


Product CFG

Product Program Construction

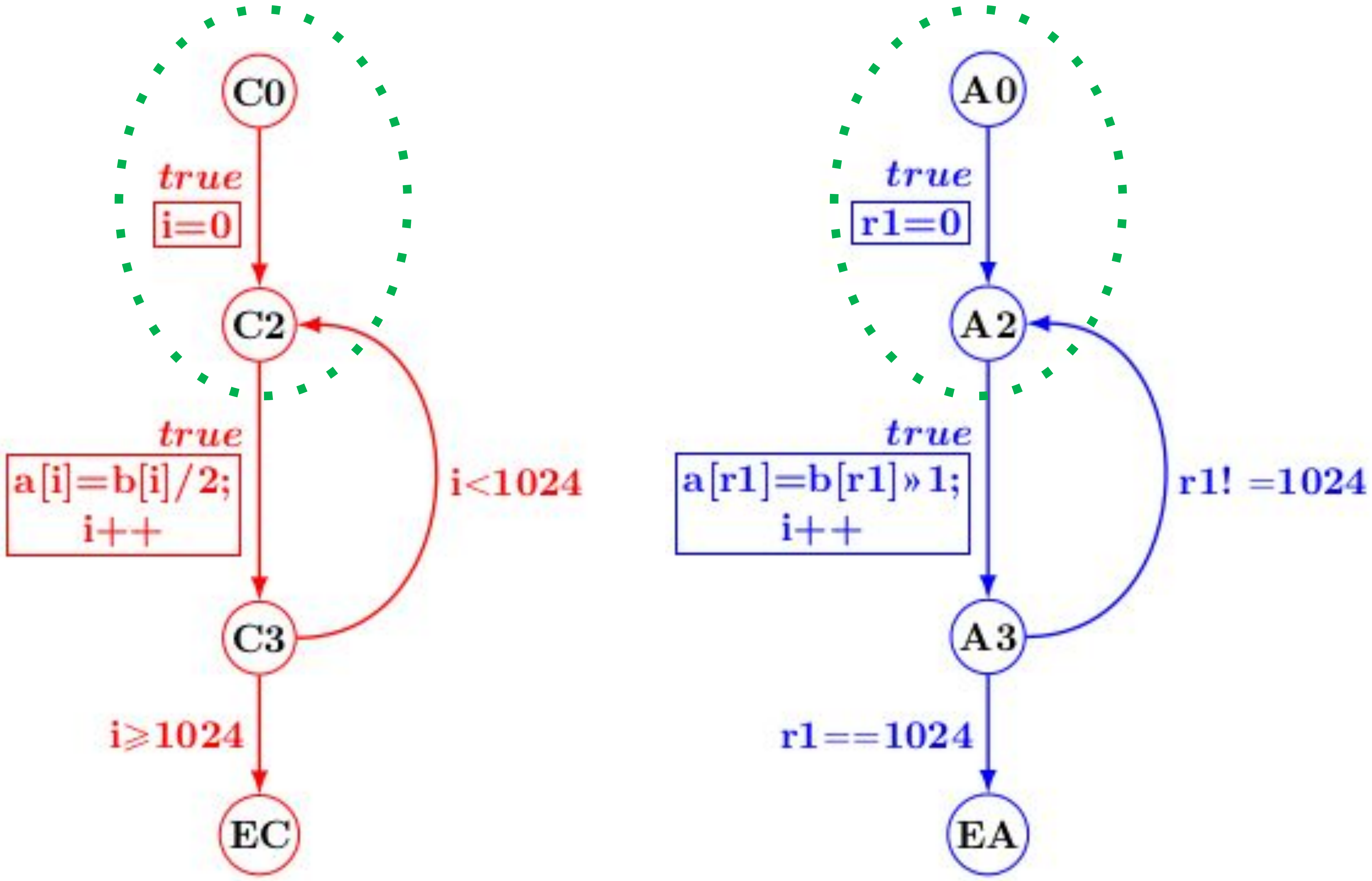


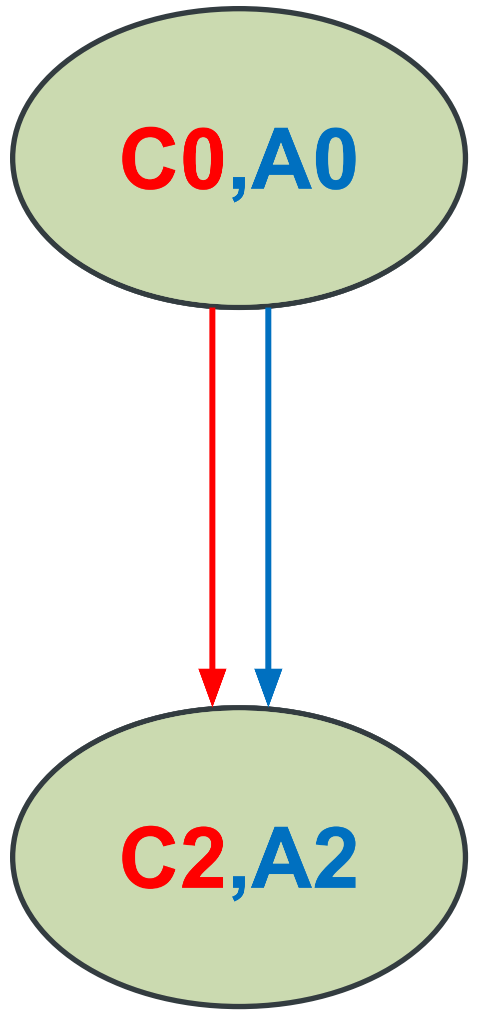




Product CFG

Product Program Construction

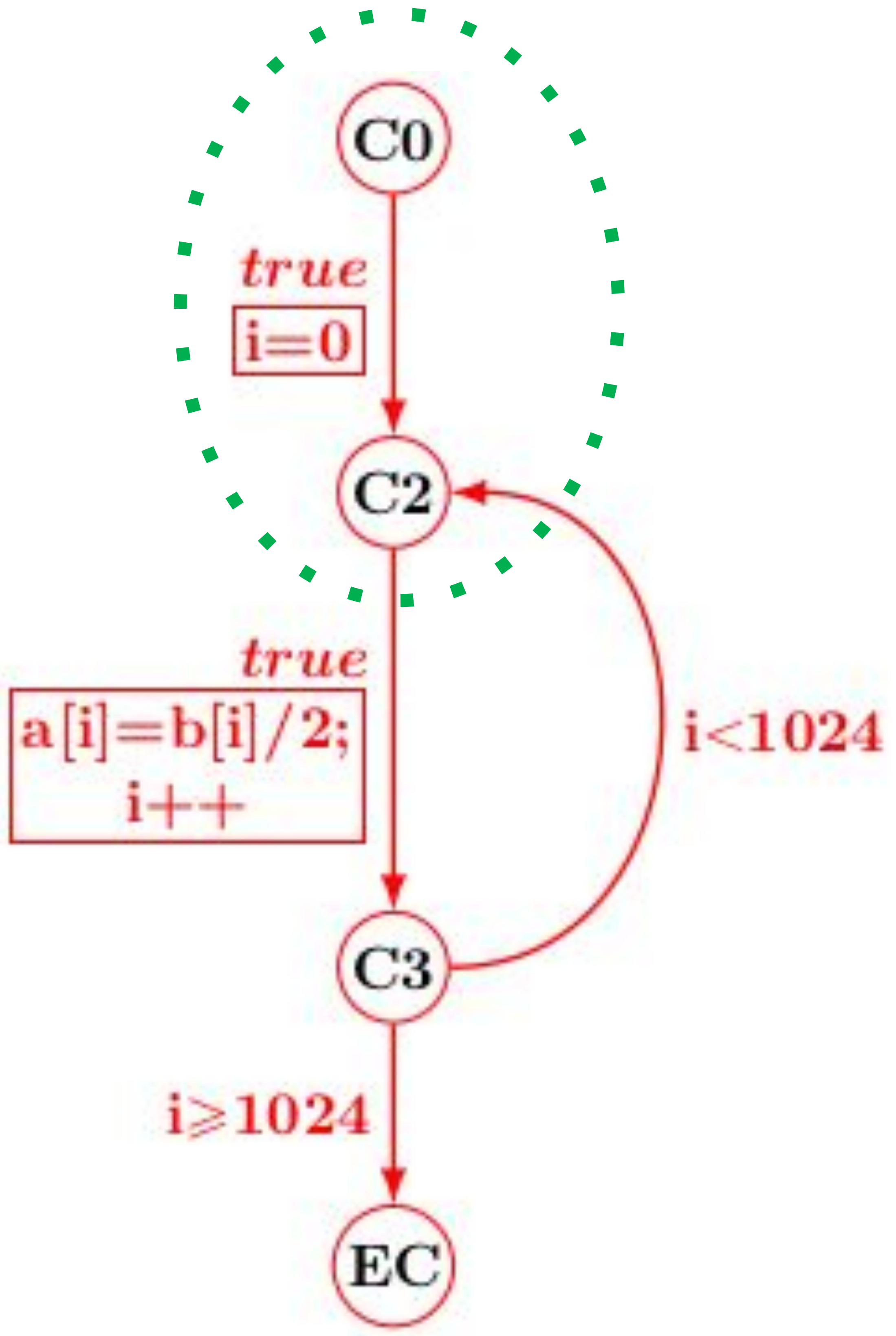


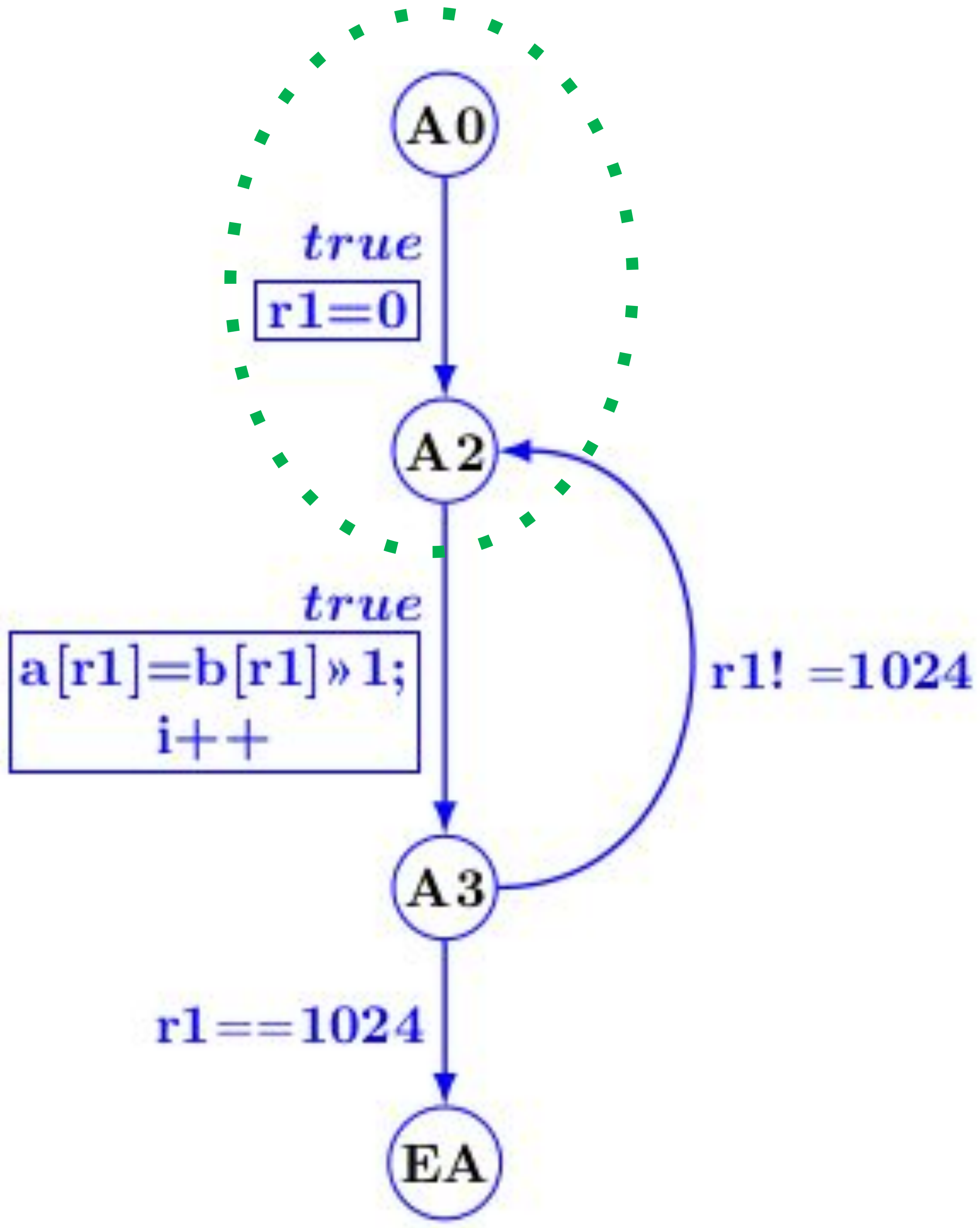


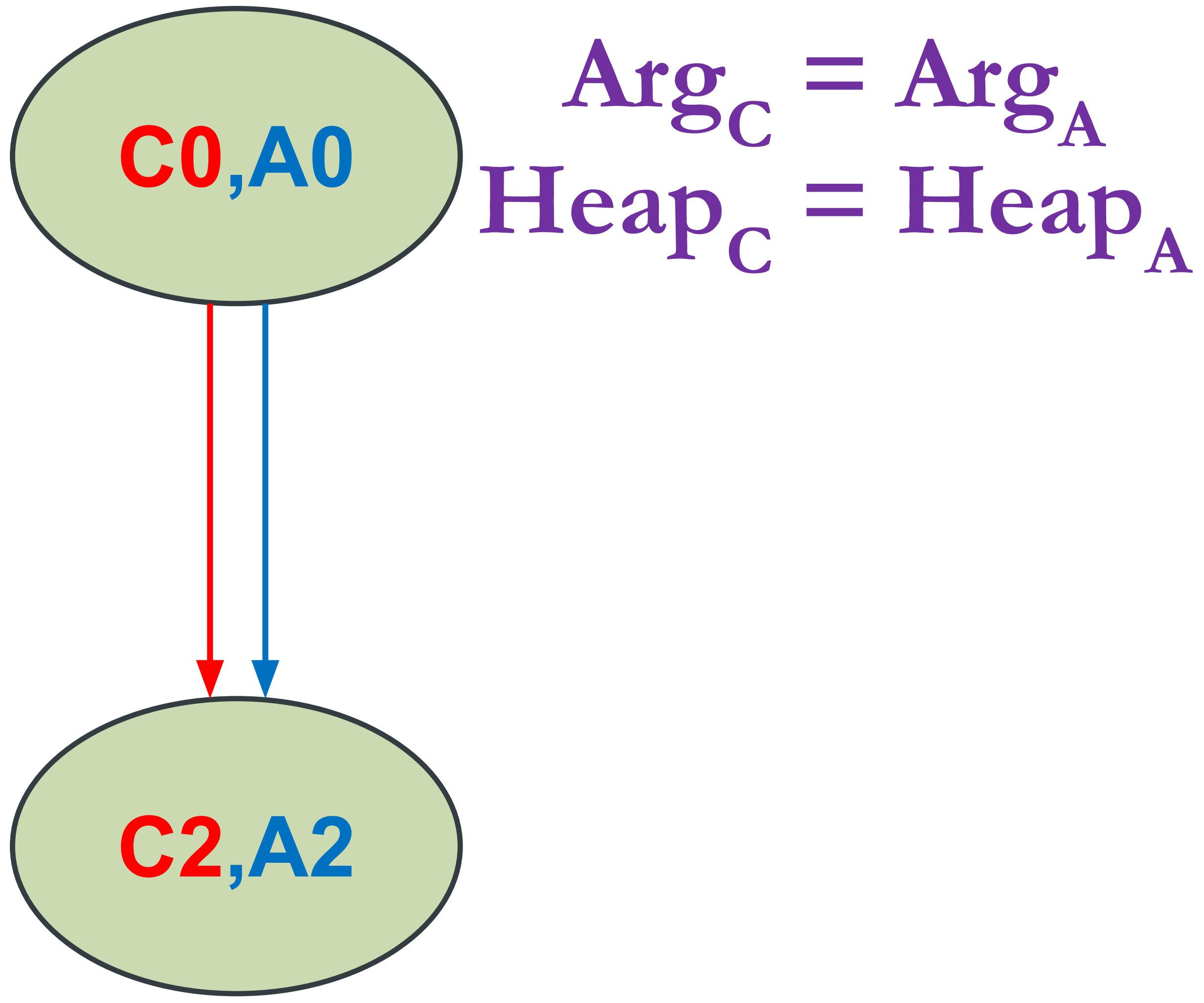
Product CFG

Product Program Construction

Product CFG

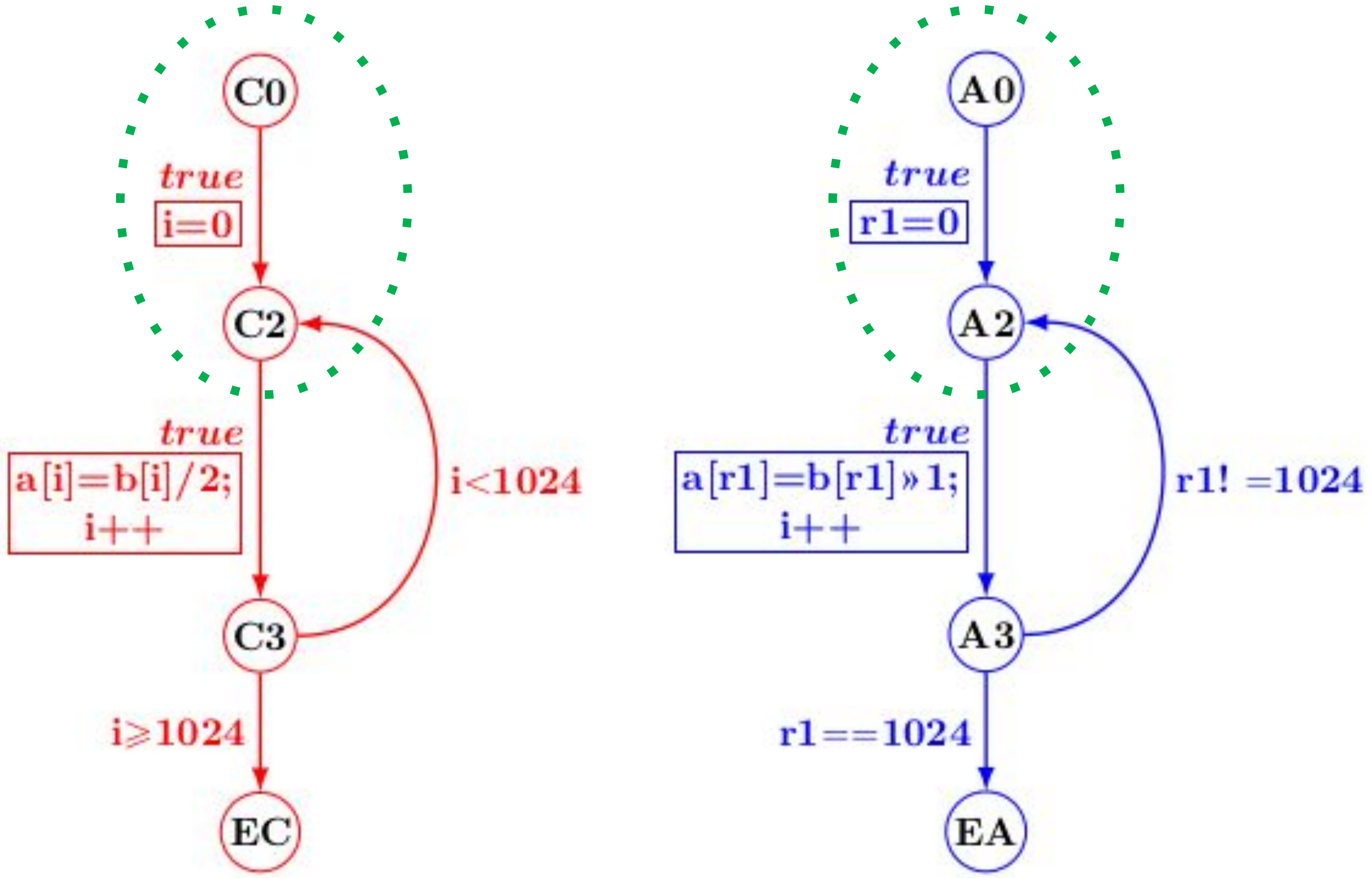


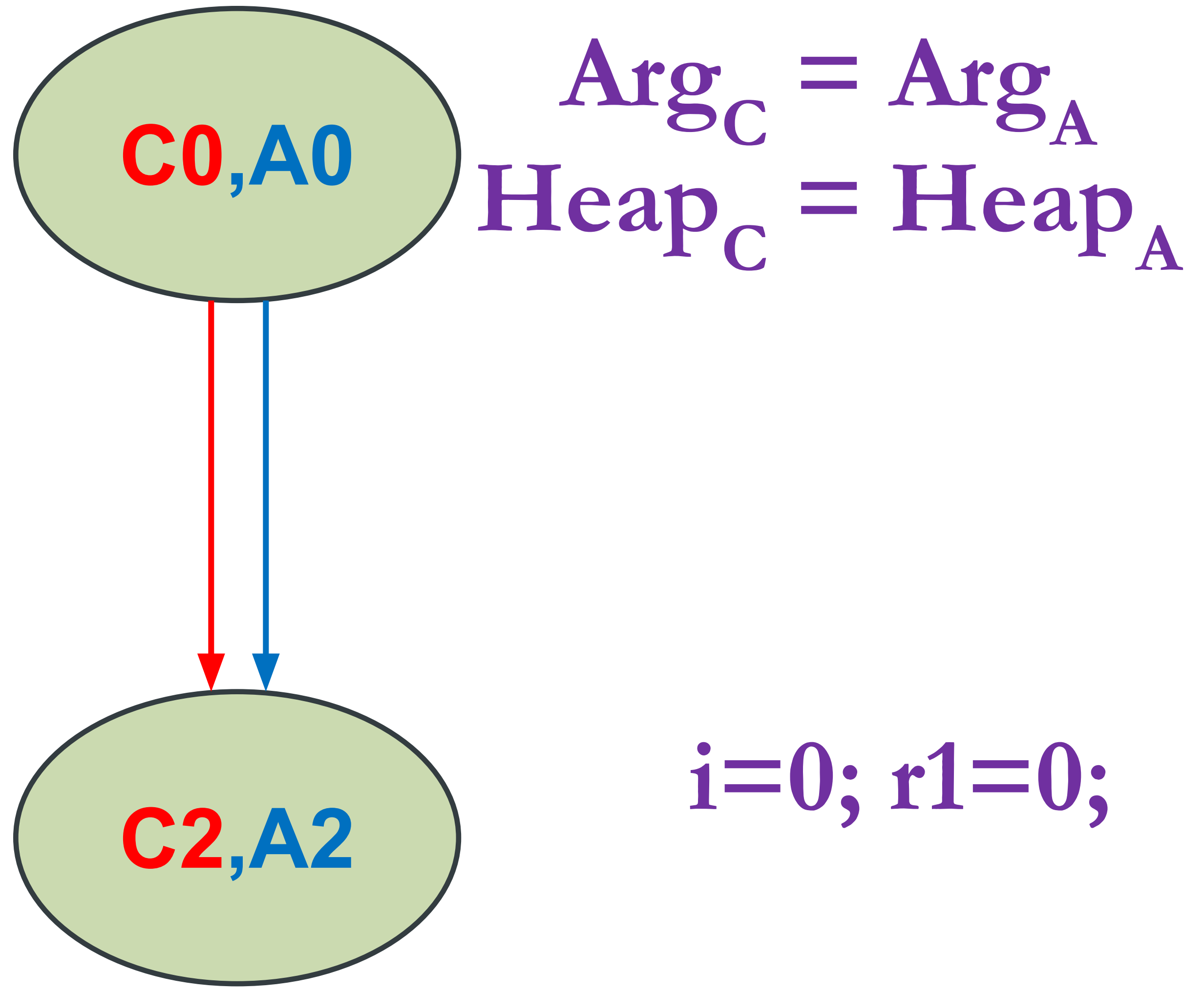




Product Program Construction

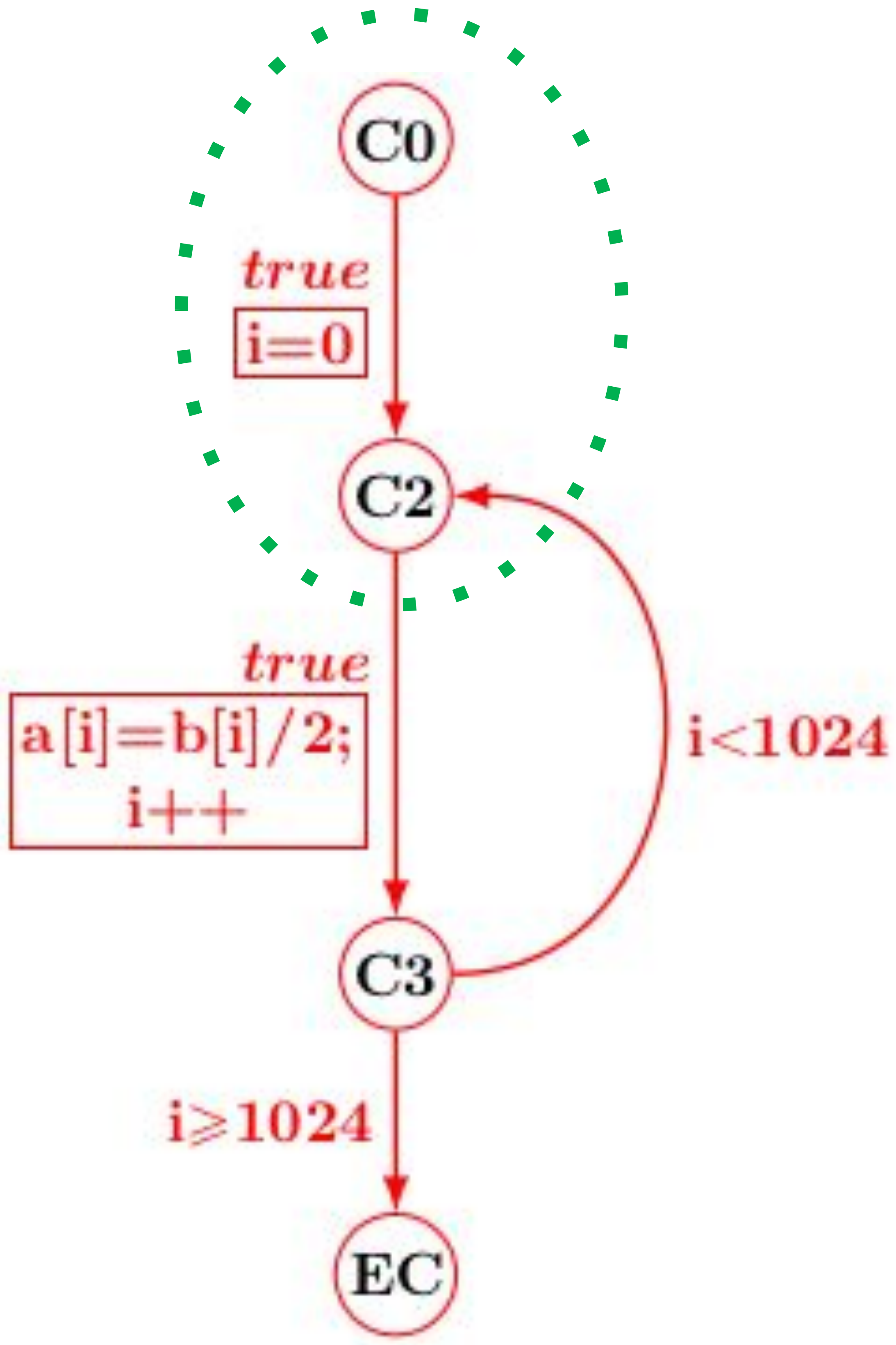
Product CFG

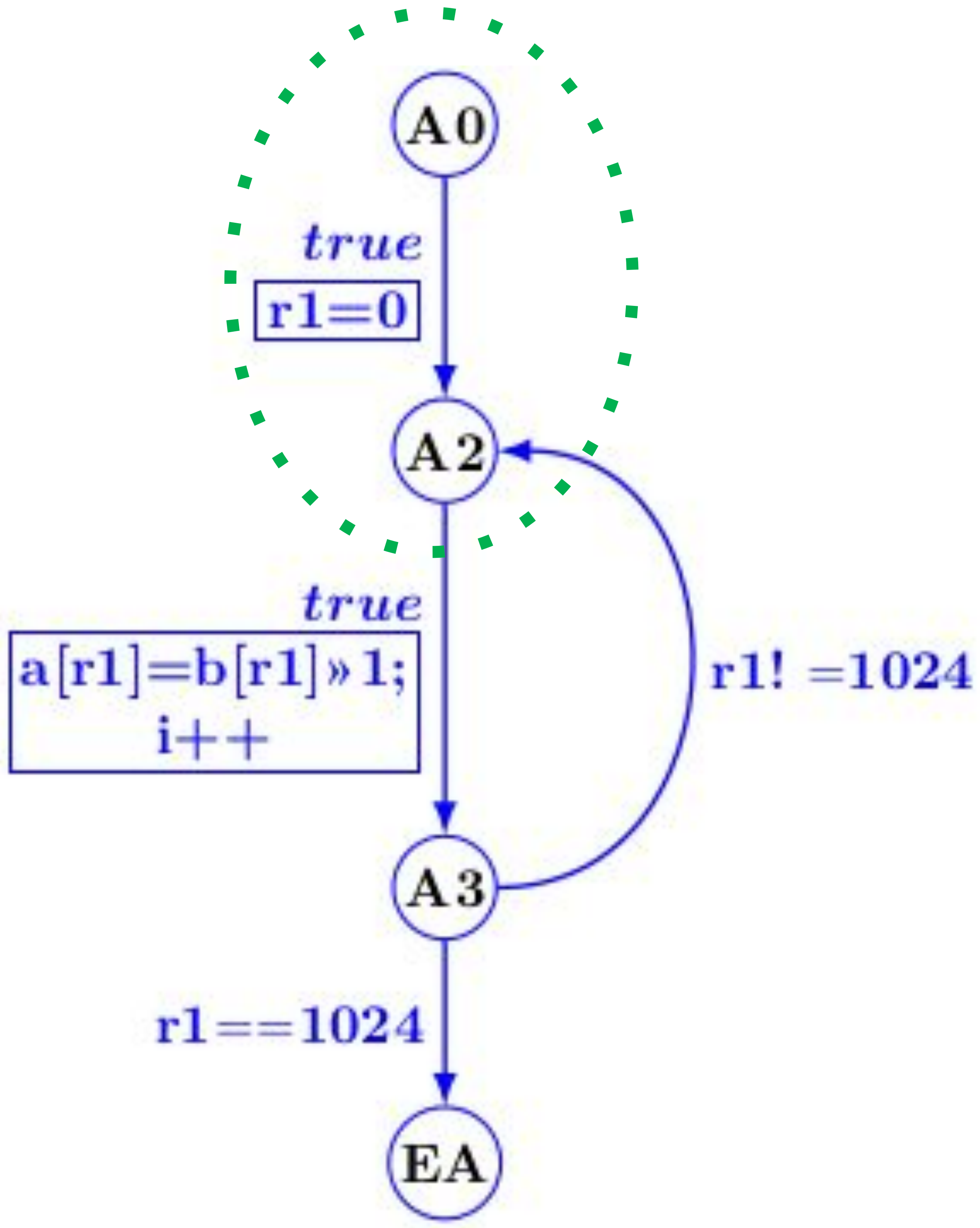


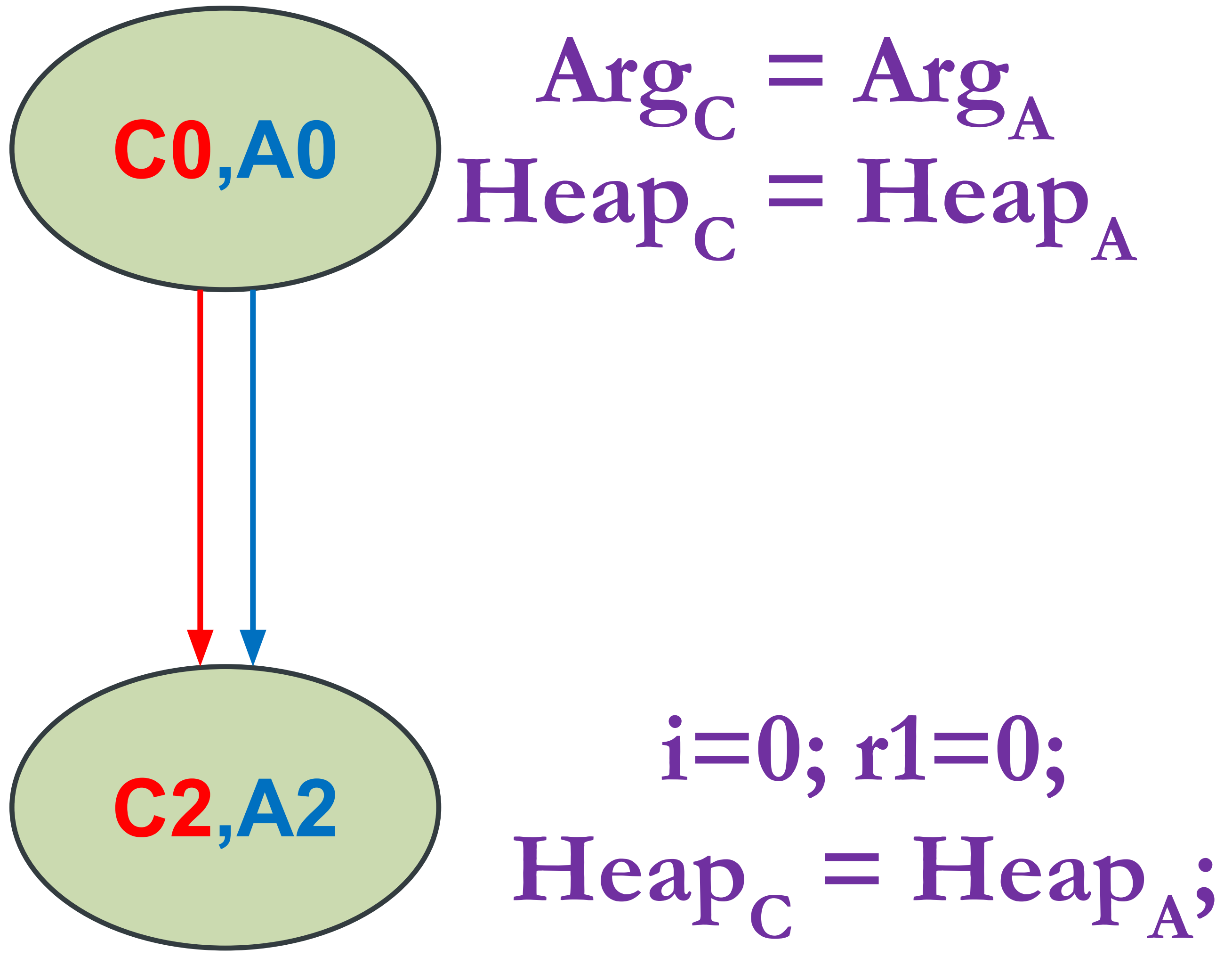


Product Program Construction

Product CFG

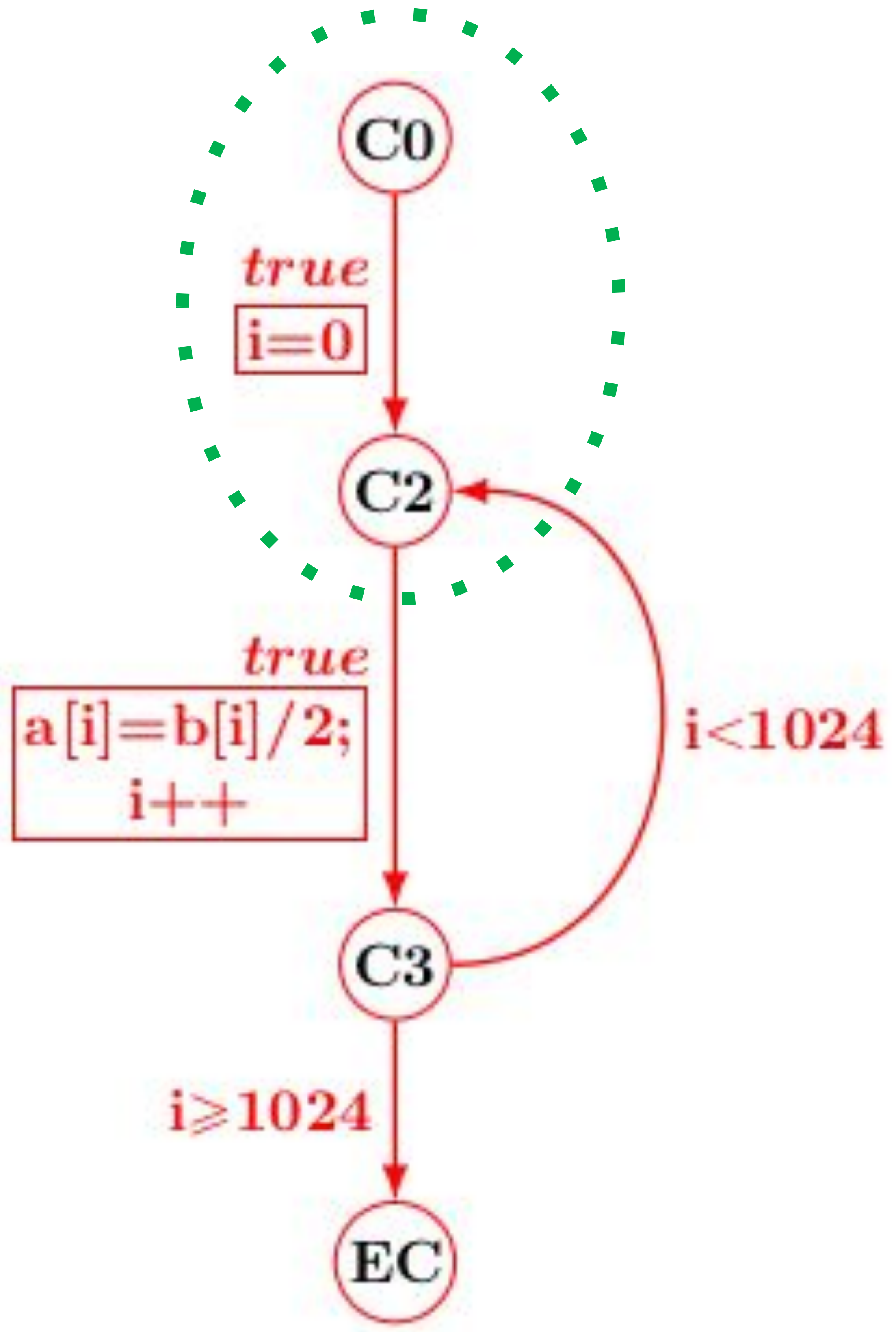


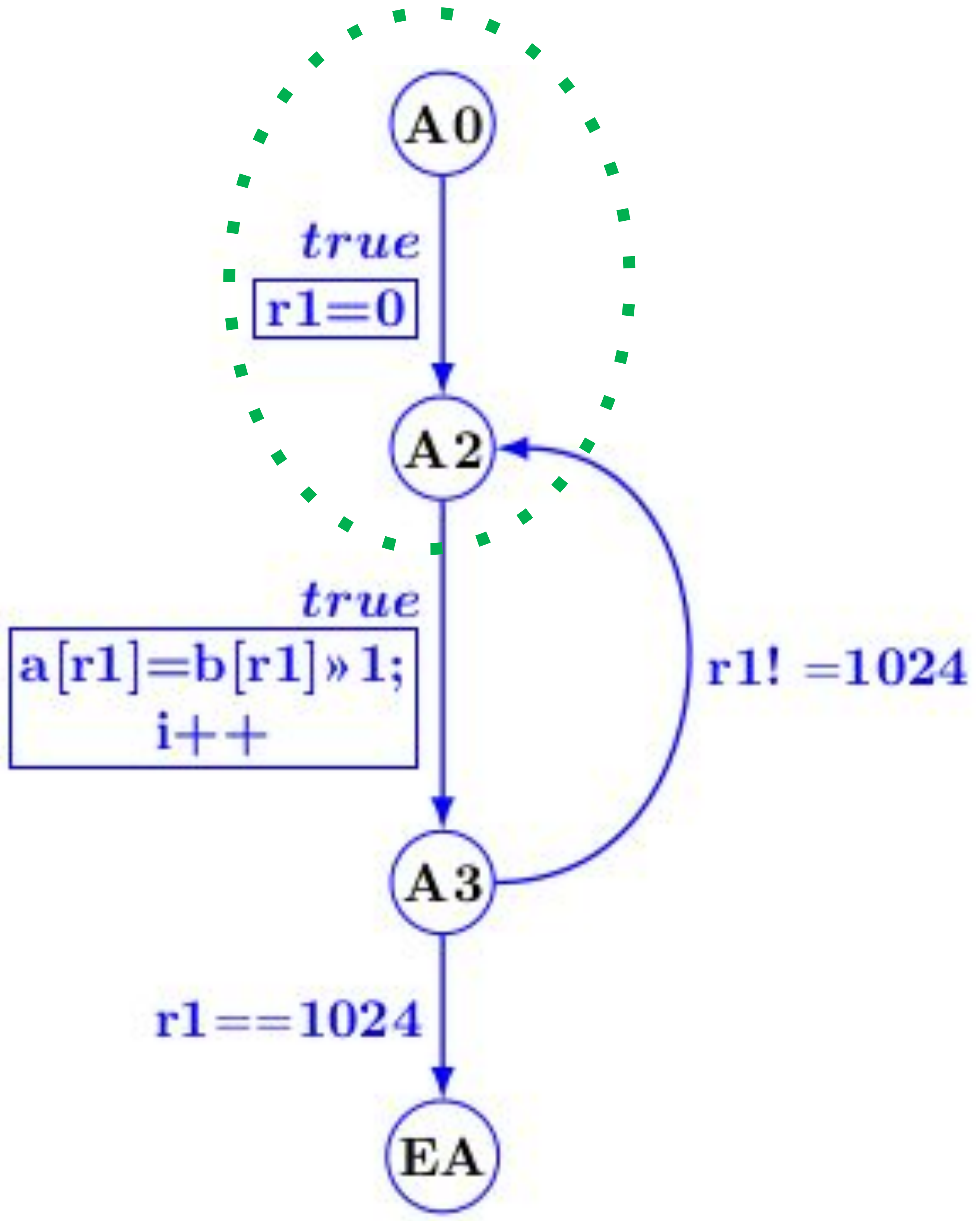


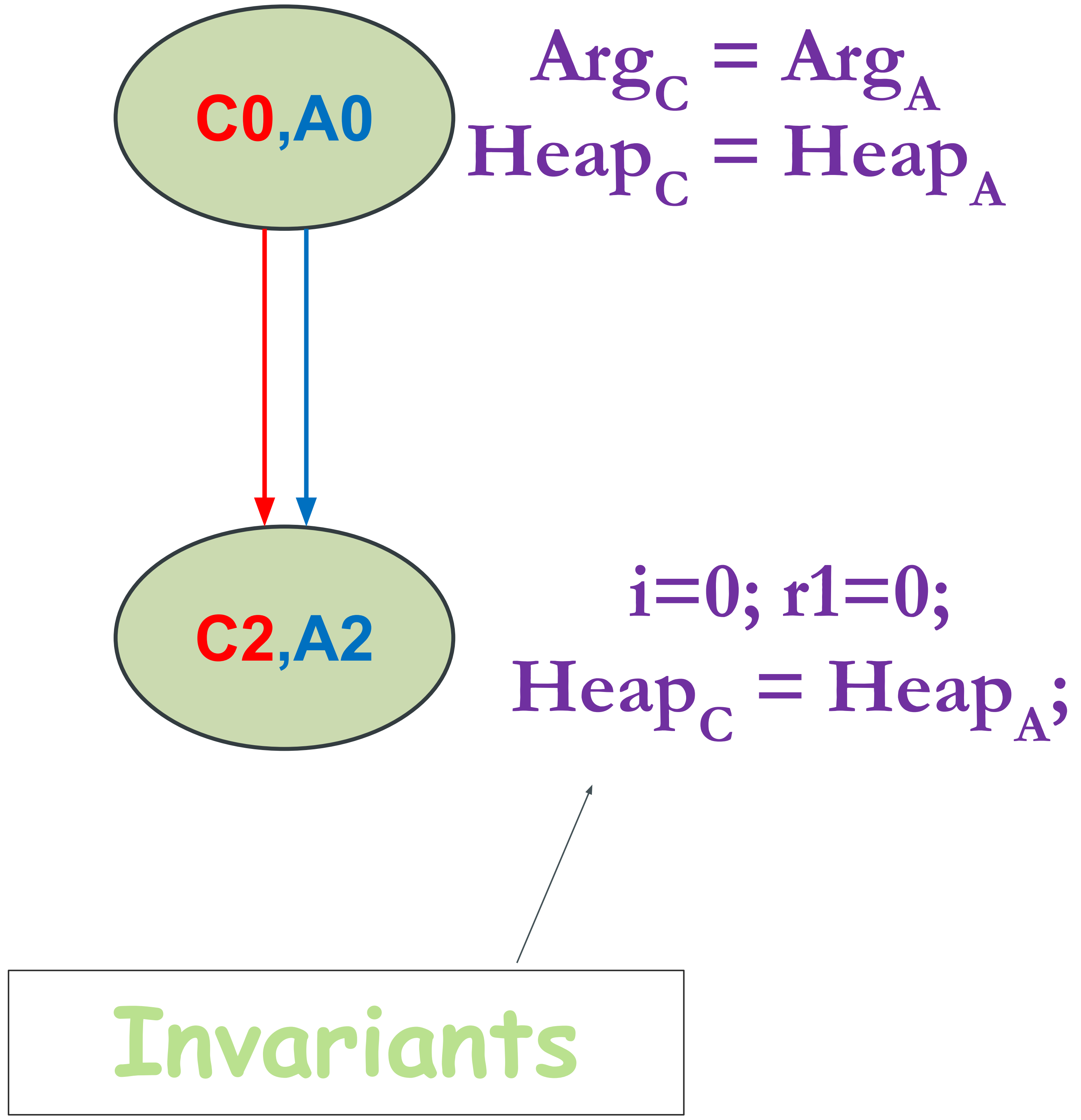


Product Program Construction

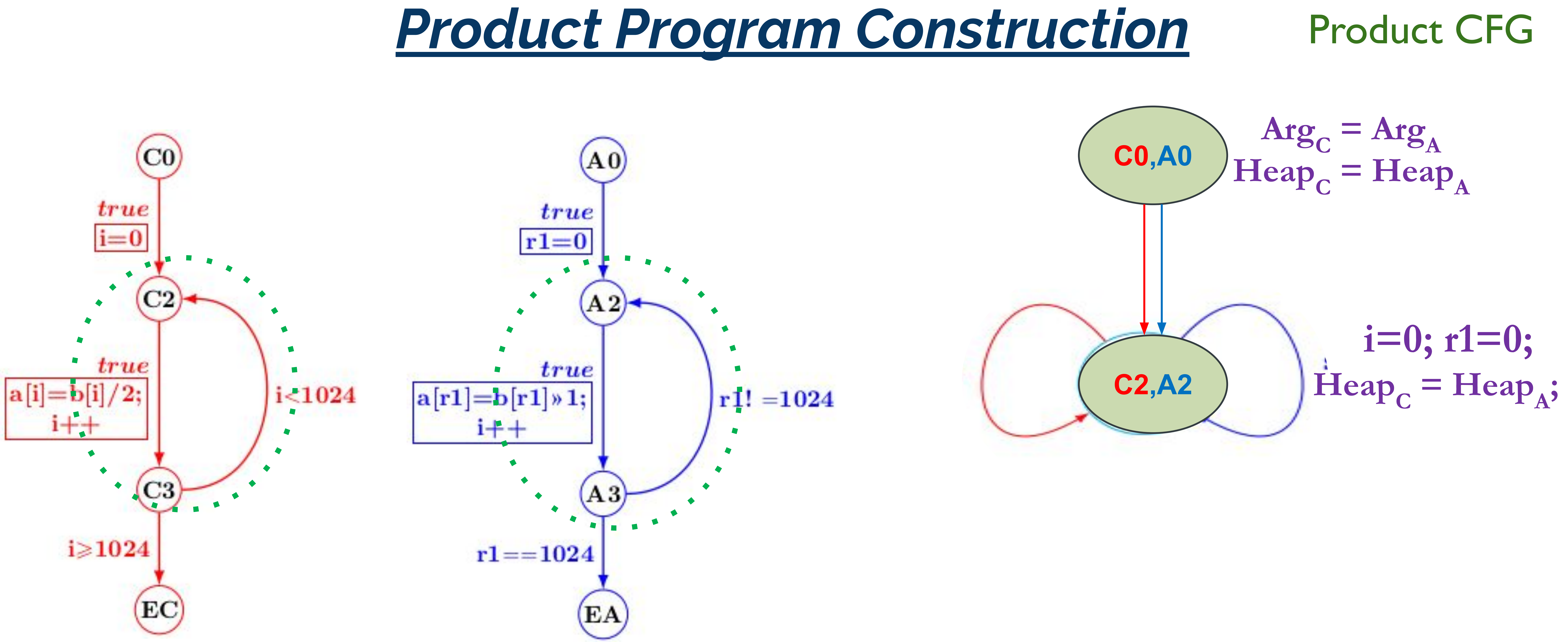
Product CFG





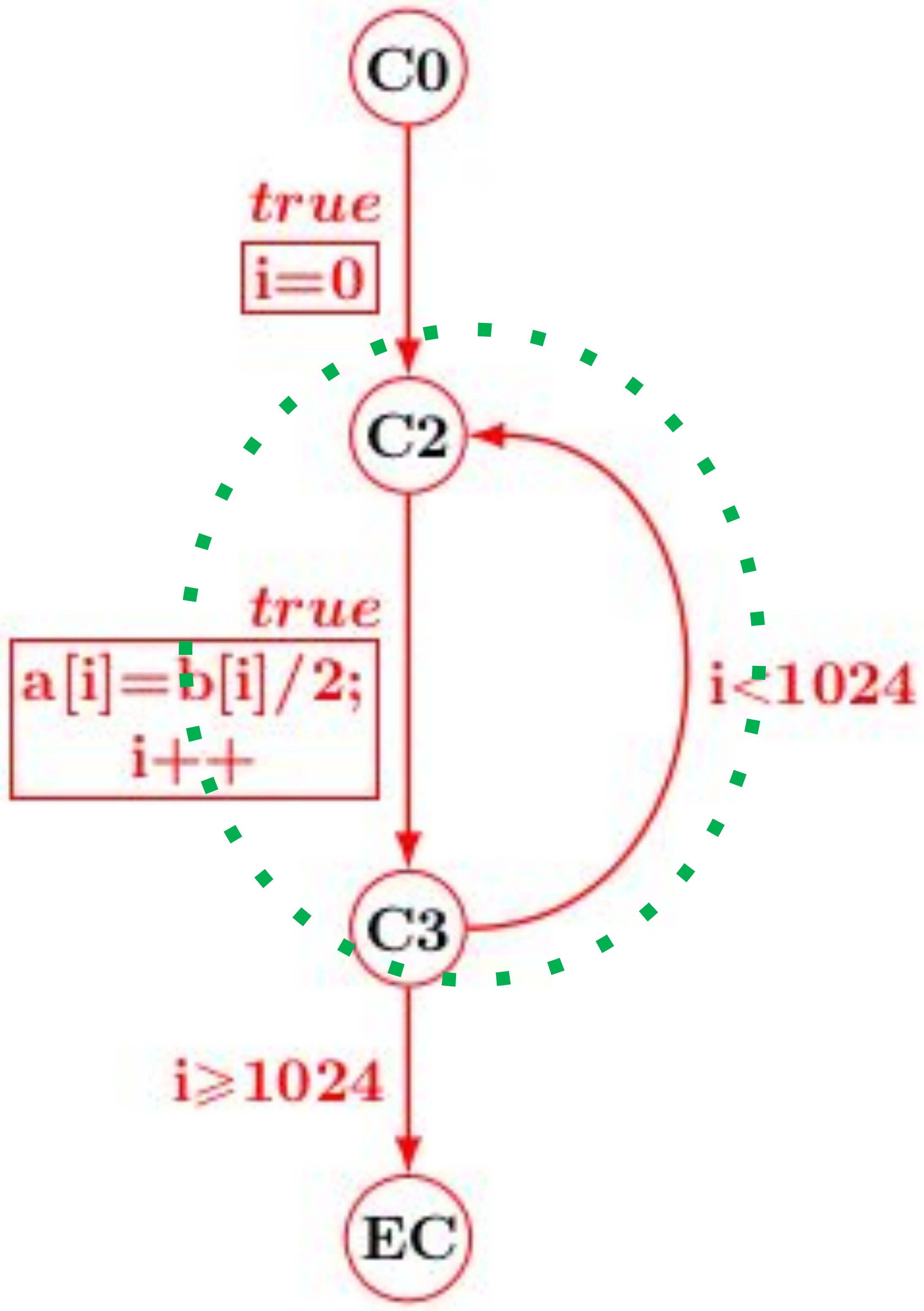


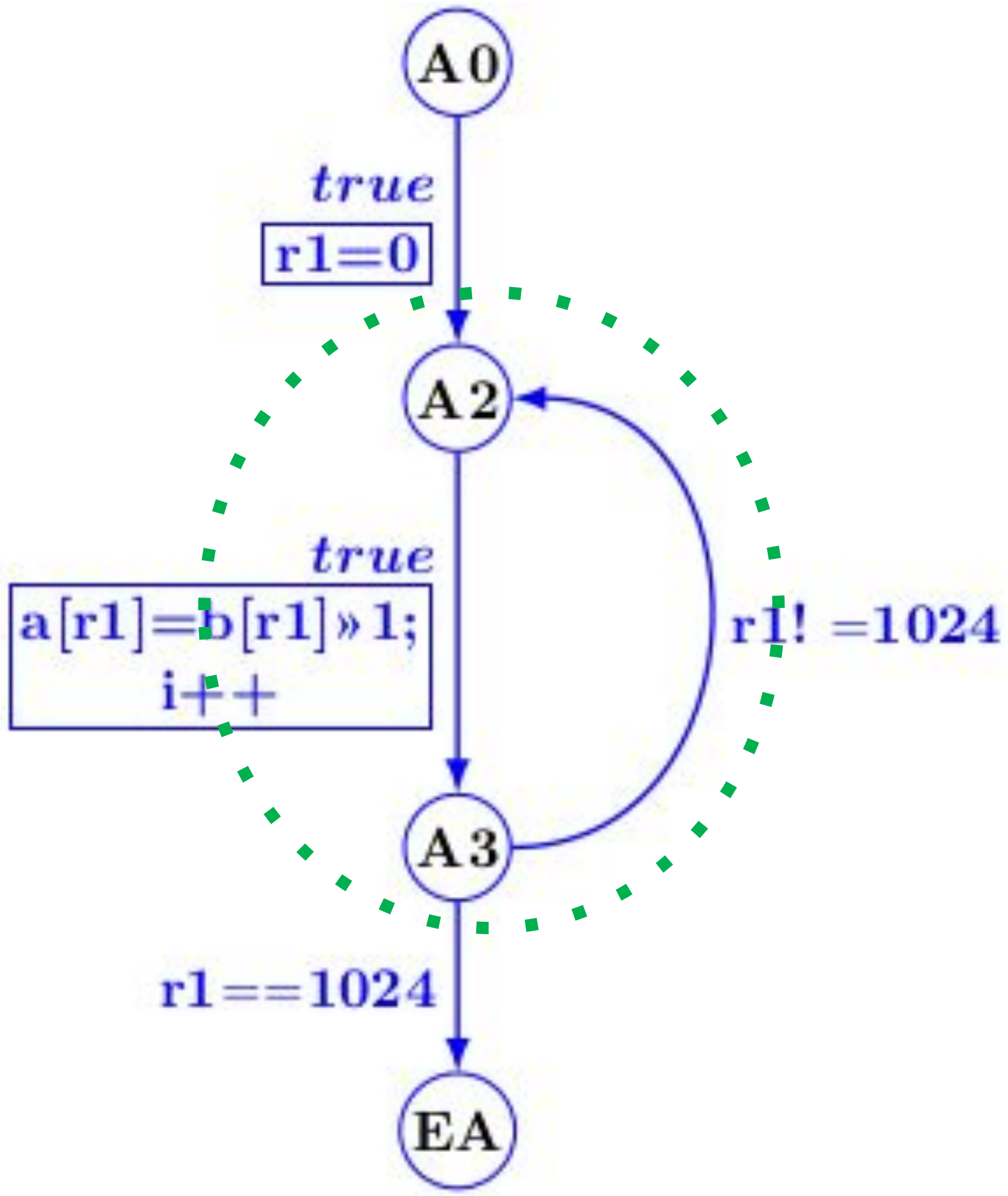
Product Program Construction

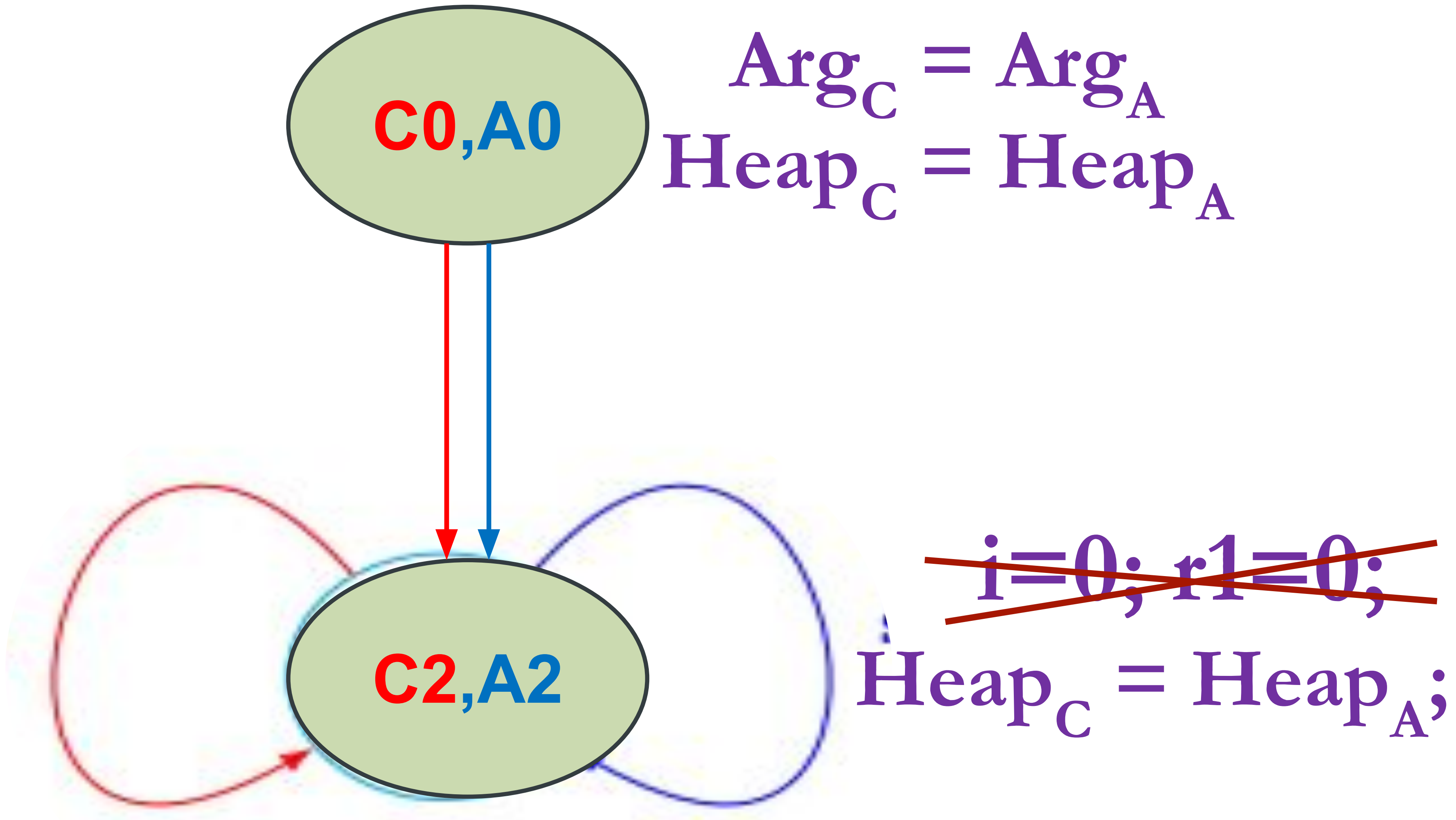


Product Program Construction

Product CFG

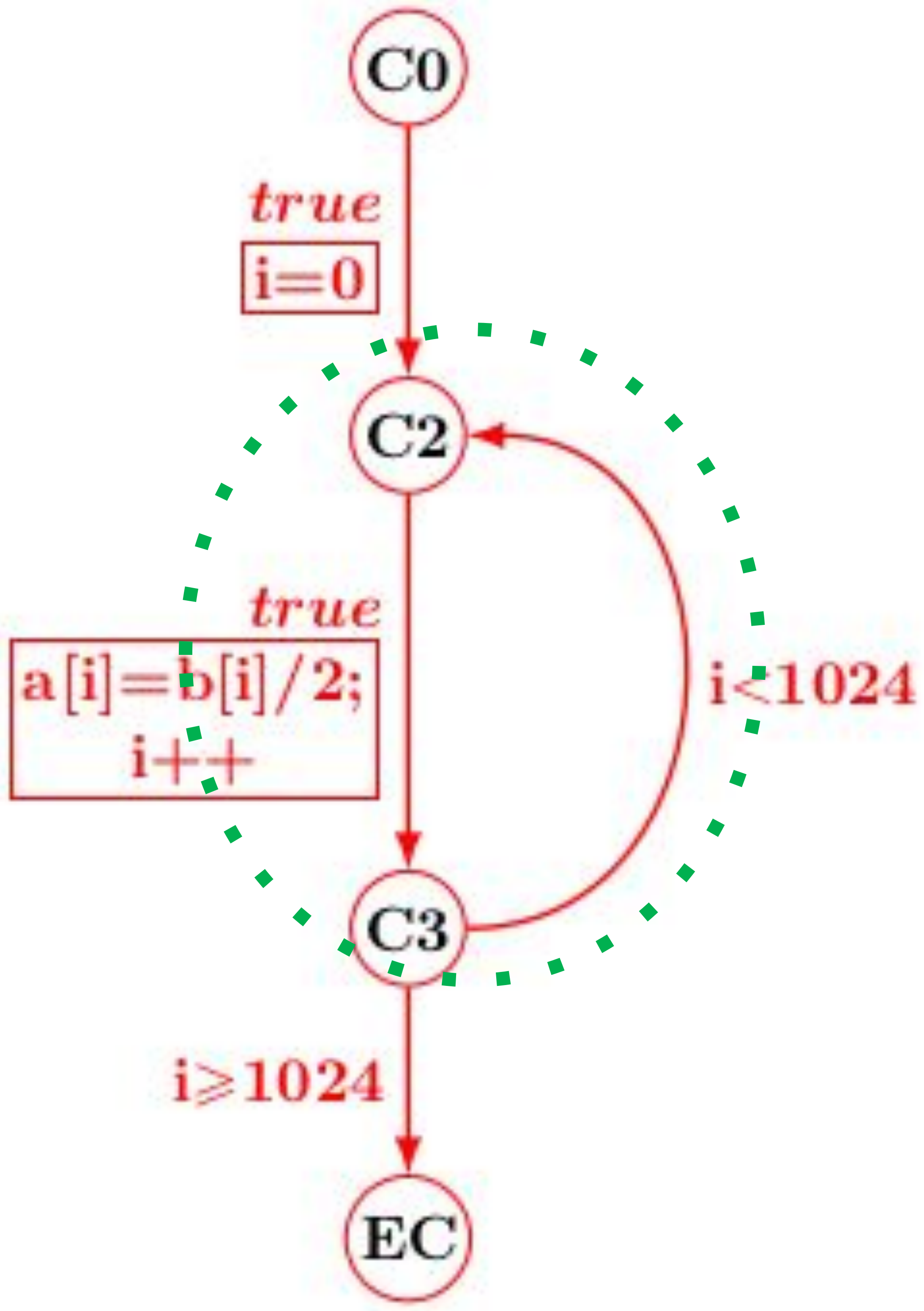


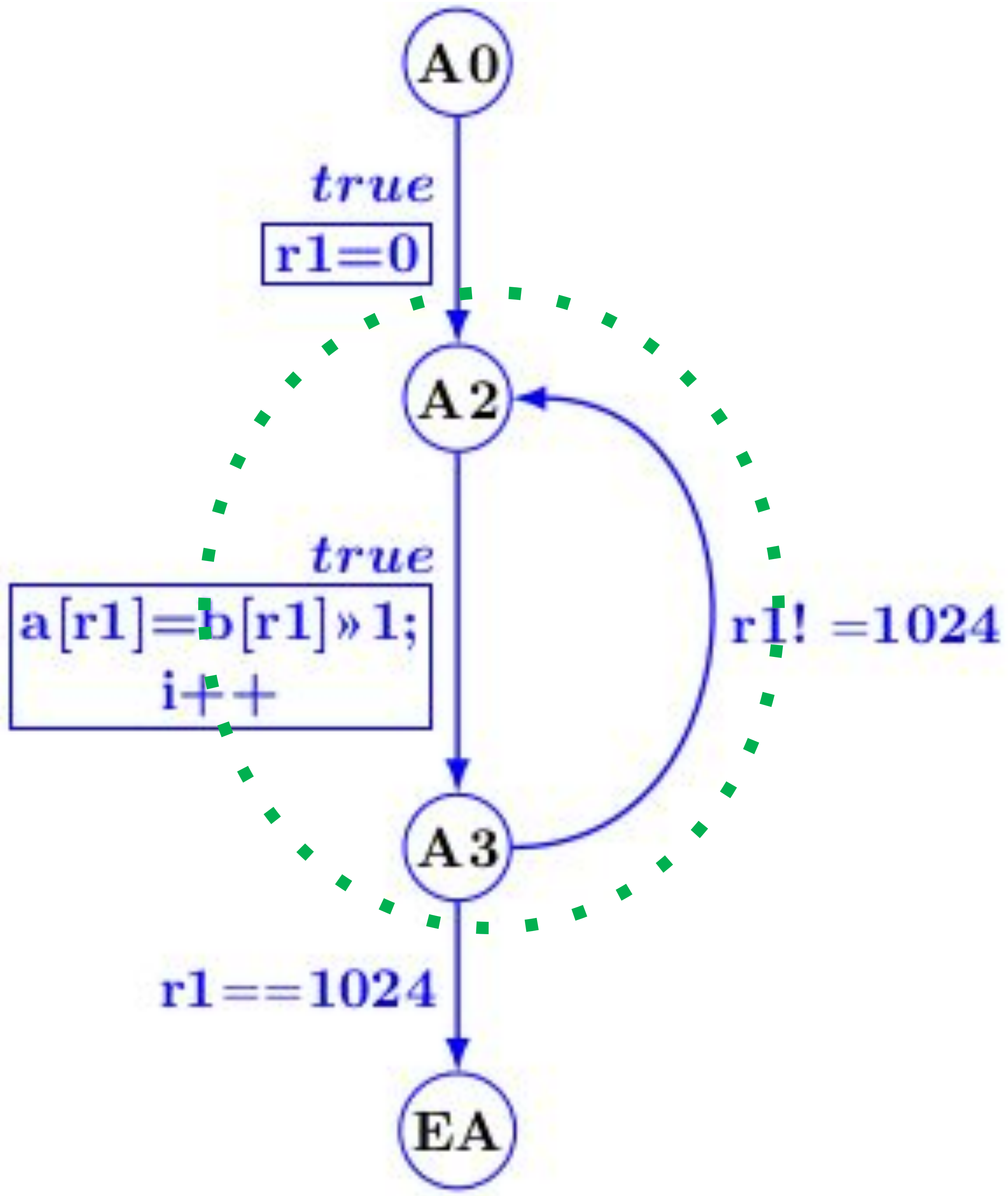


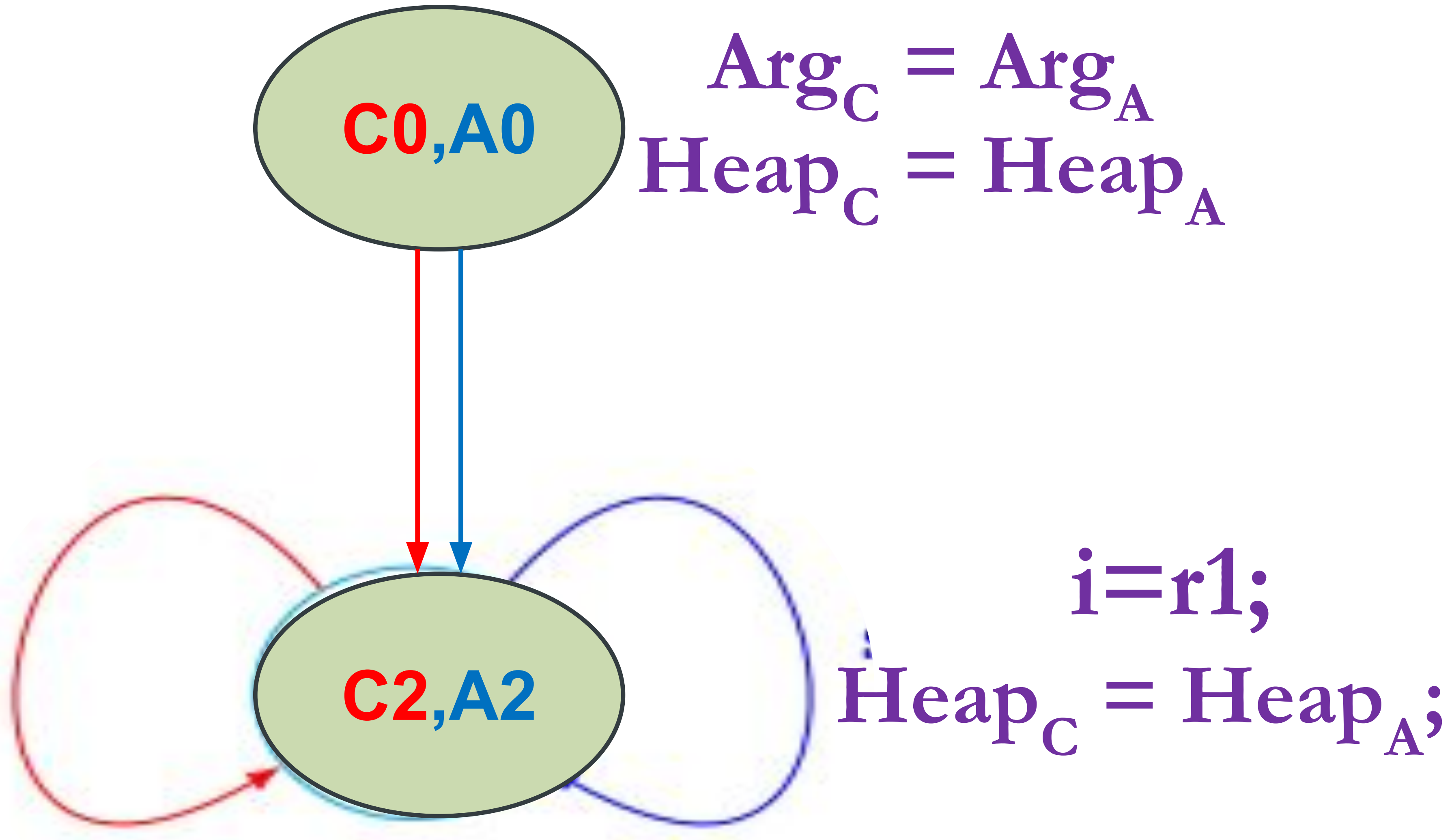


Product Program Construction

Product CFG

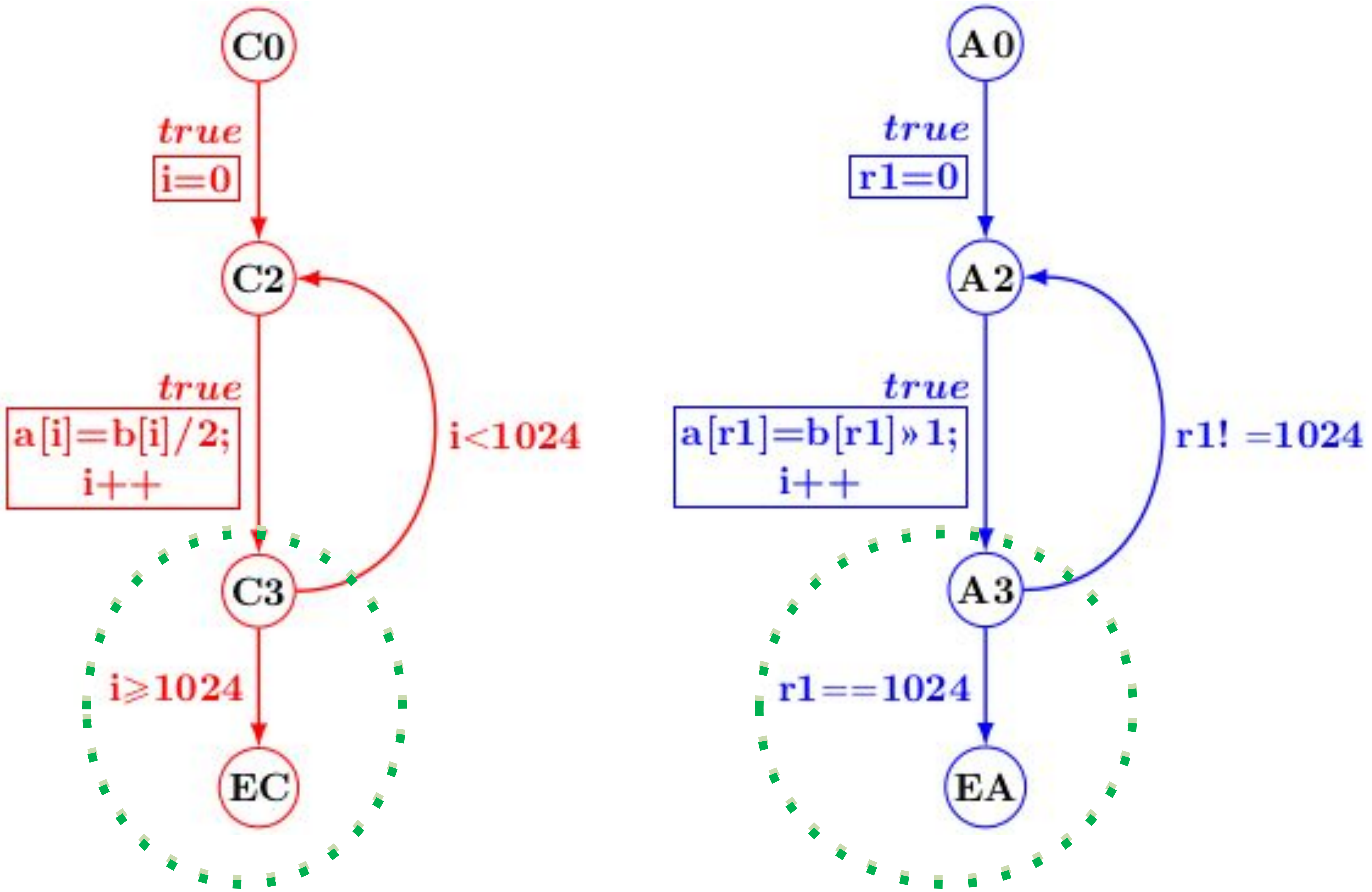


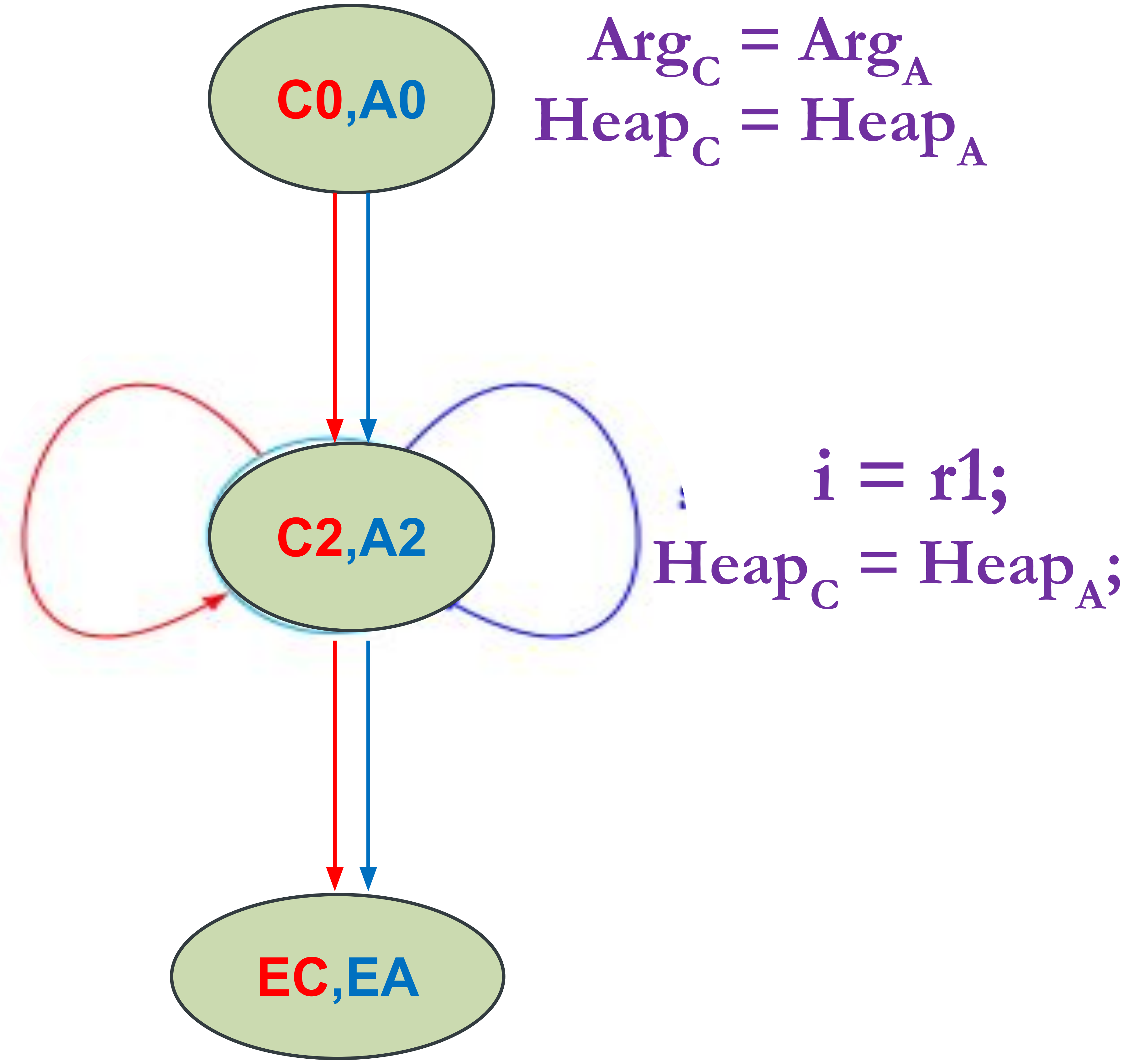




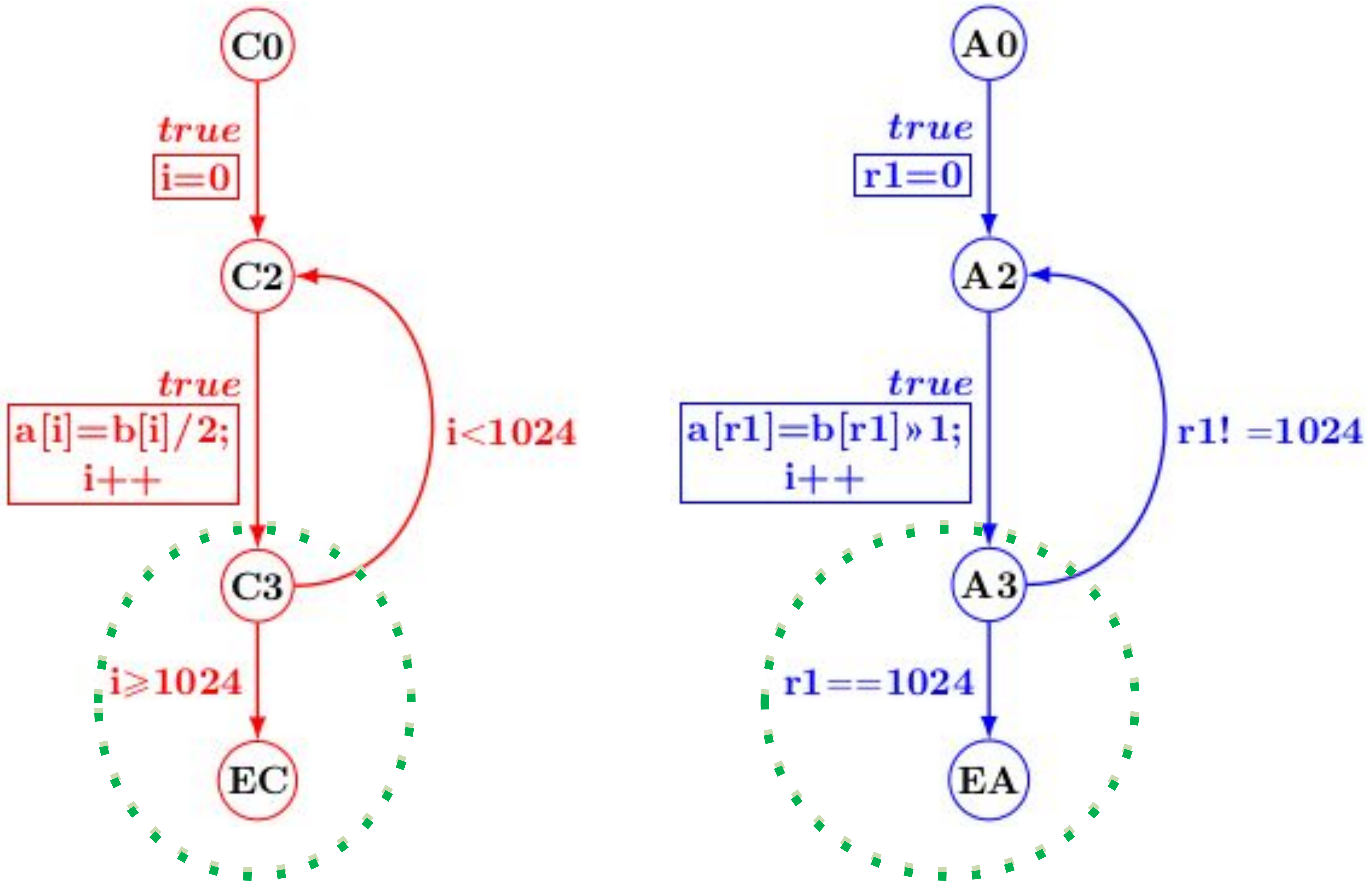
Product Program Construction

Product CFG

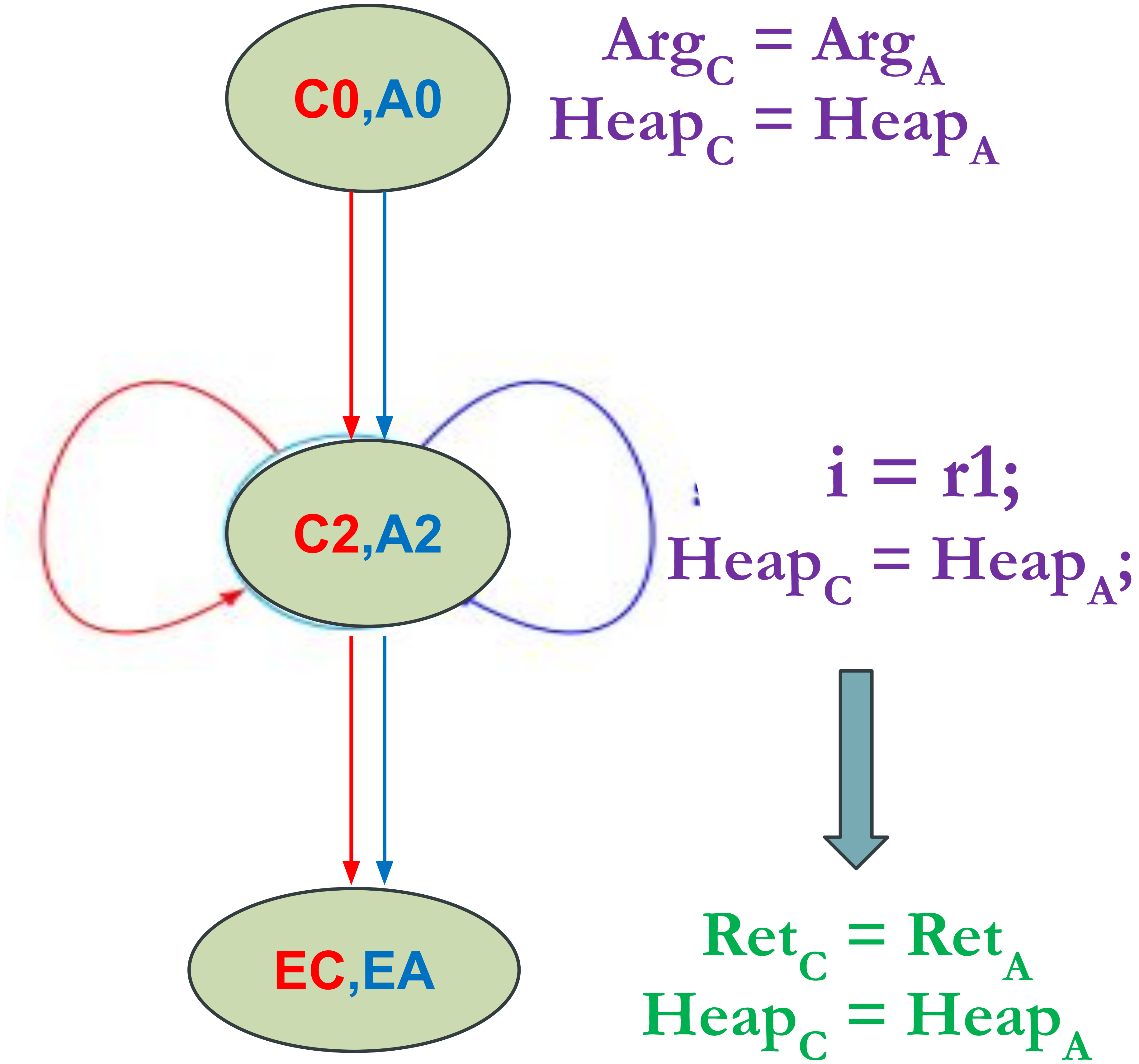




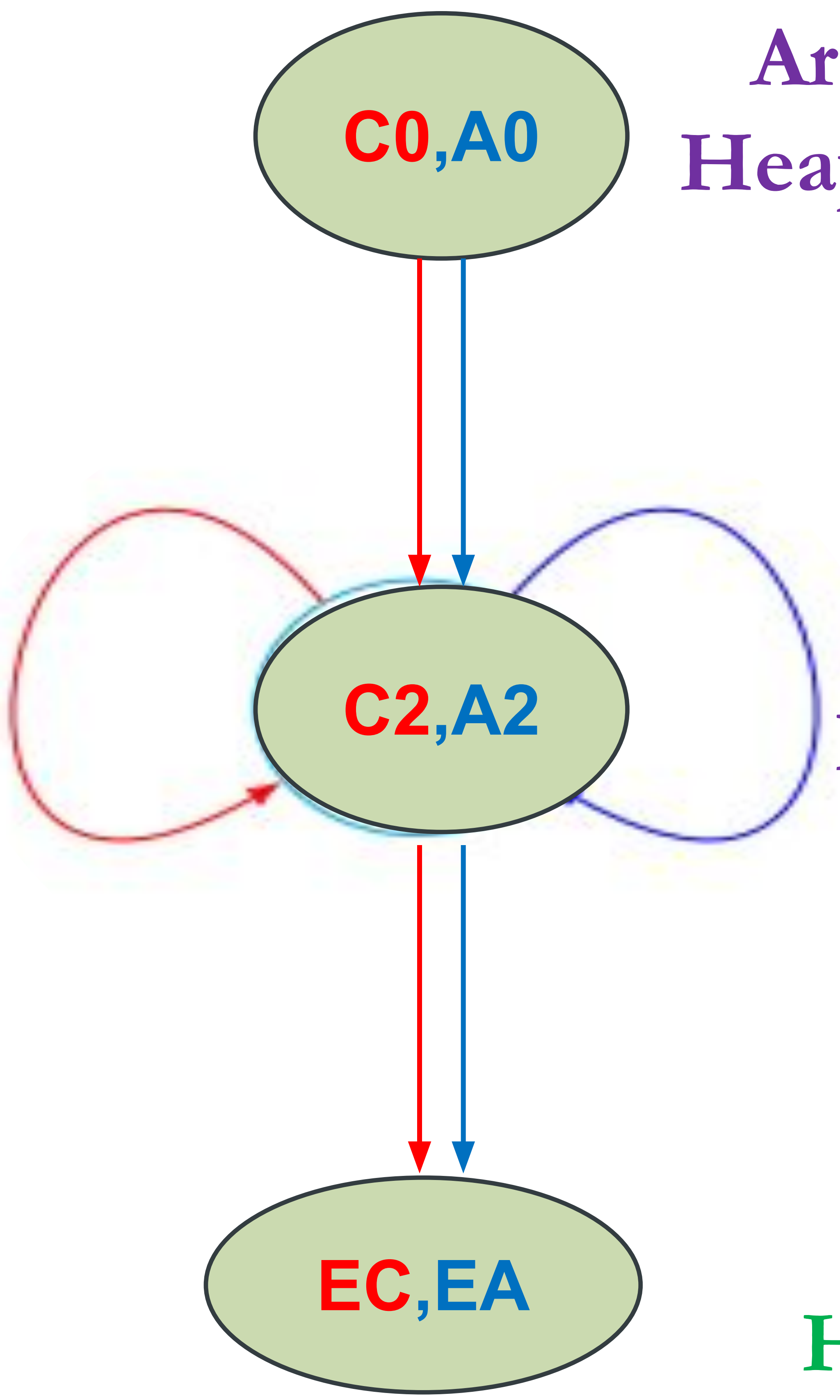
Prove at exit points



Product CFG



Equivalence Checking



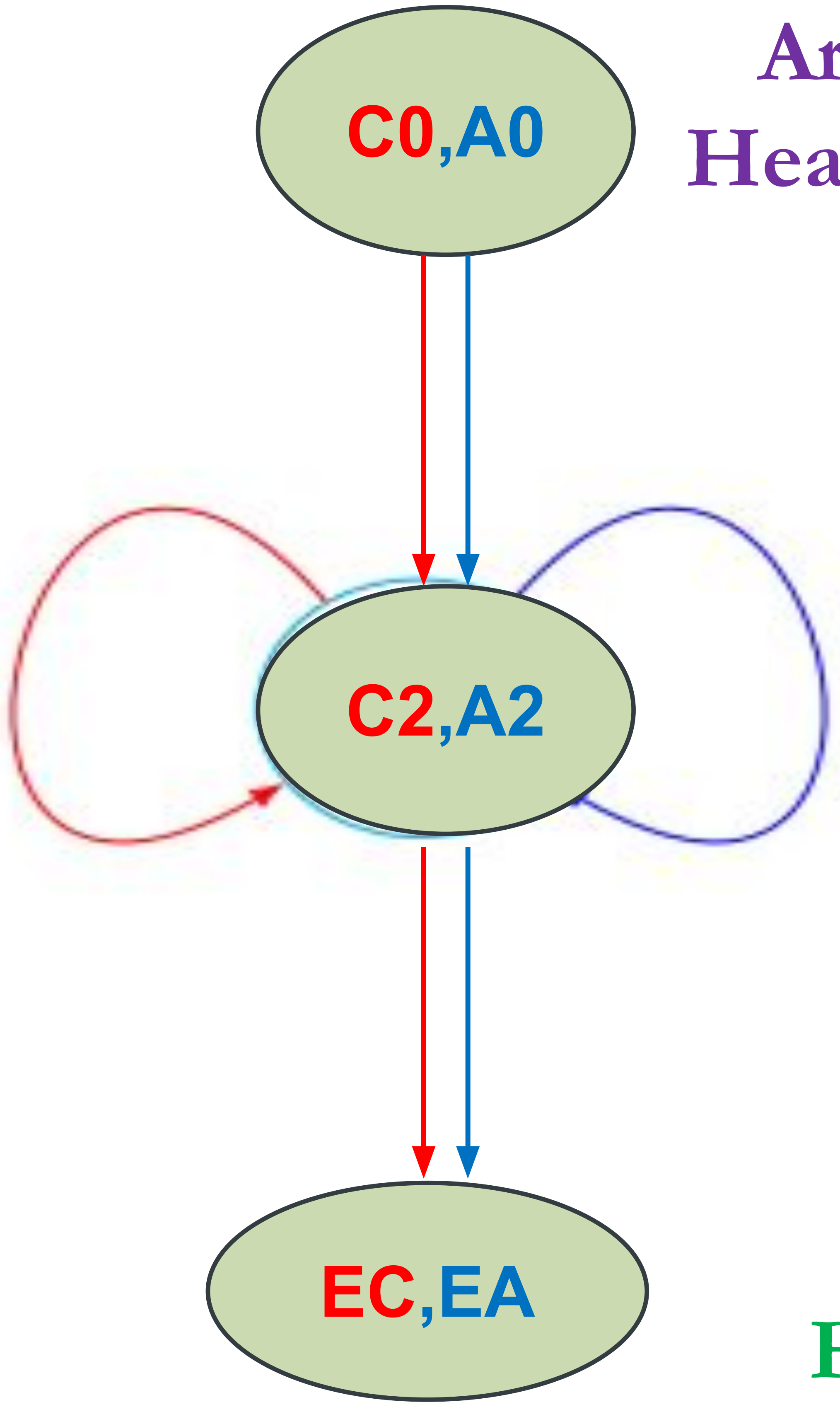
$Arg_C = Arg_A$
$Heap_C = Heap_A$

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

$Ret_C = Ret_A$
$Heap_C = Heap_A$

Sound

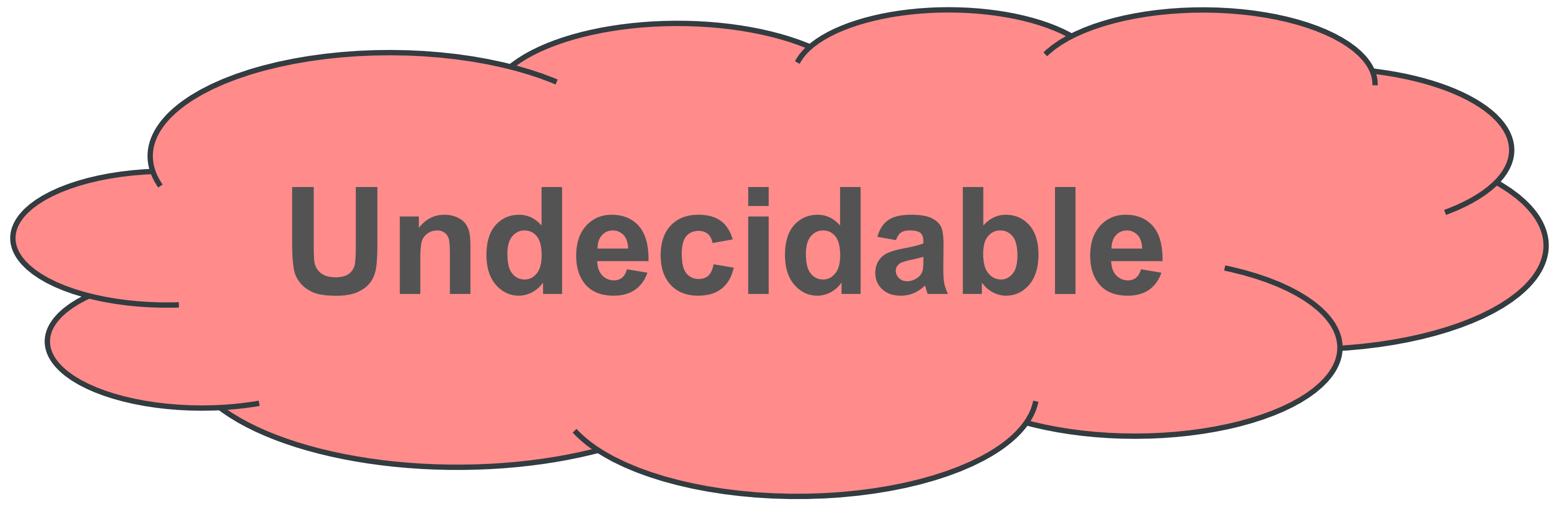
Equivalence Checking



$Arg_C = Arg_A$
$Heap_C = Heap_A$

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

$Ret_C = Ret_A$
$Heap_C = Heap_A$



Identifying the correlated transitions

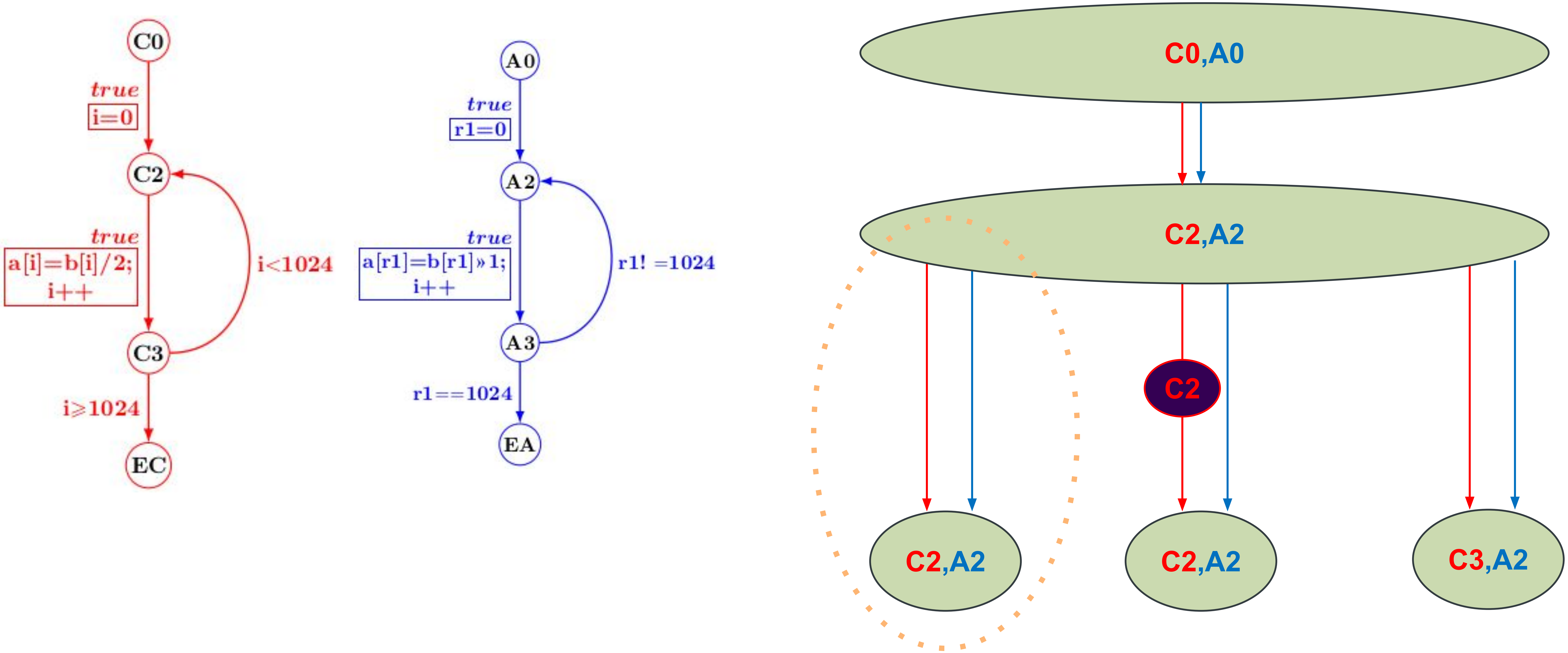
Identifying the Invariants

Equivalence Checking

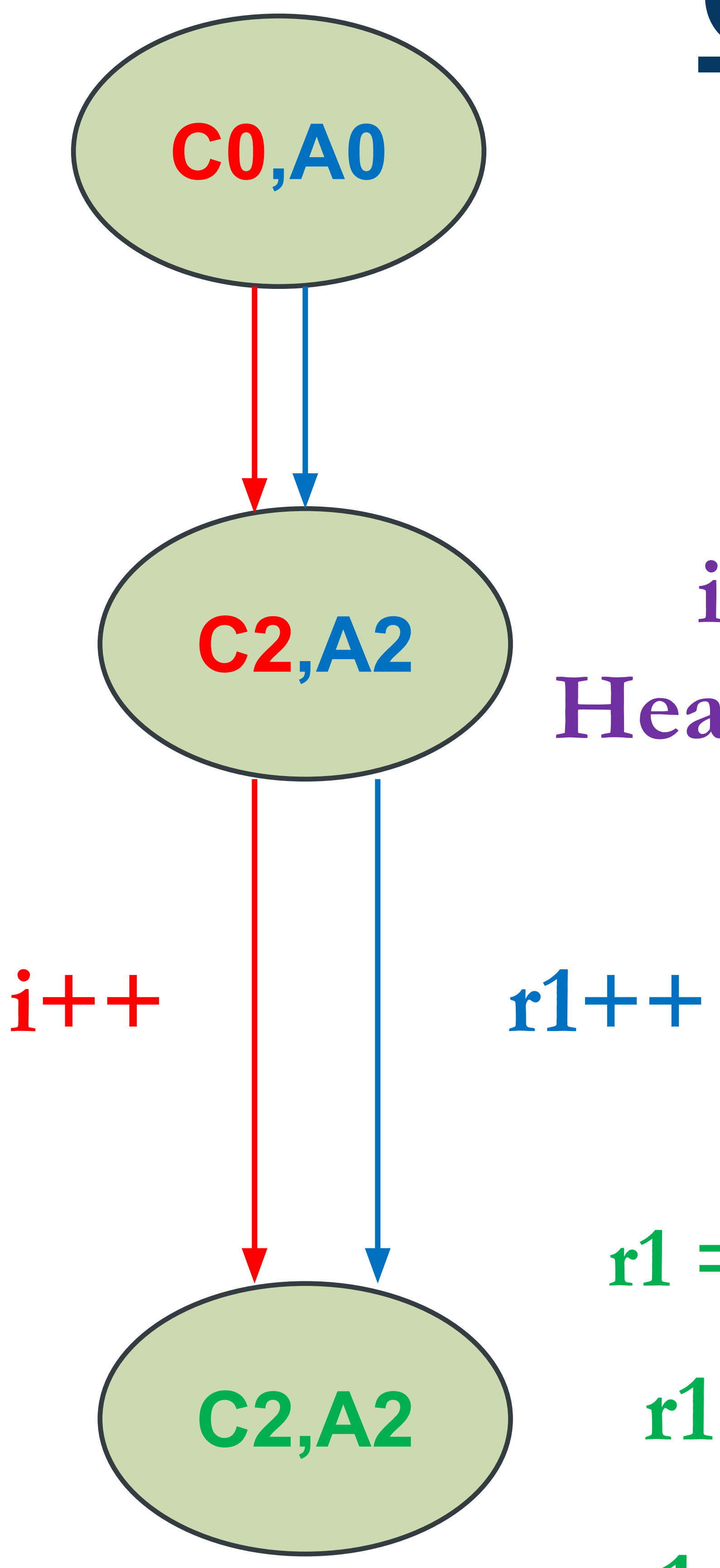
Robustness

GOAL

Enumerate multiple possible paths



Guessing the relations



i=0; r1=0;

$Heap_C = Heap_A$;

Affine relations

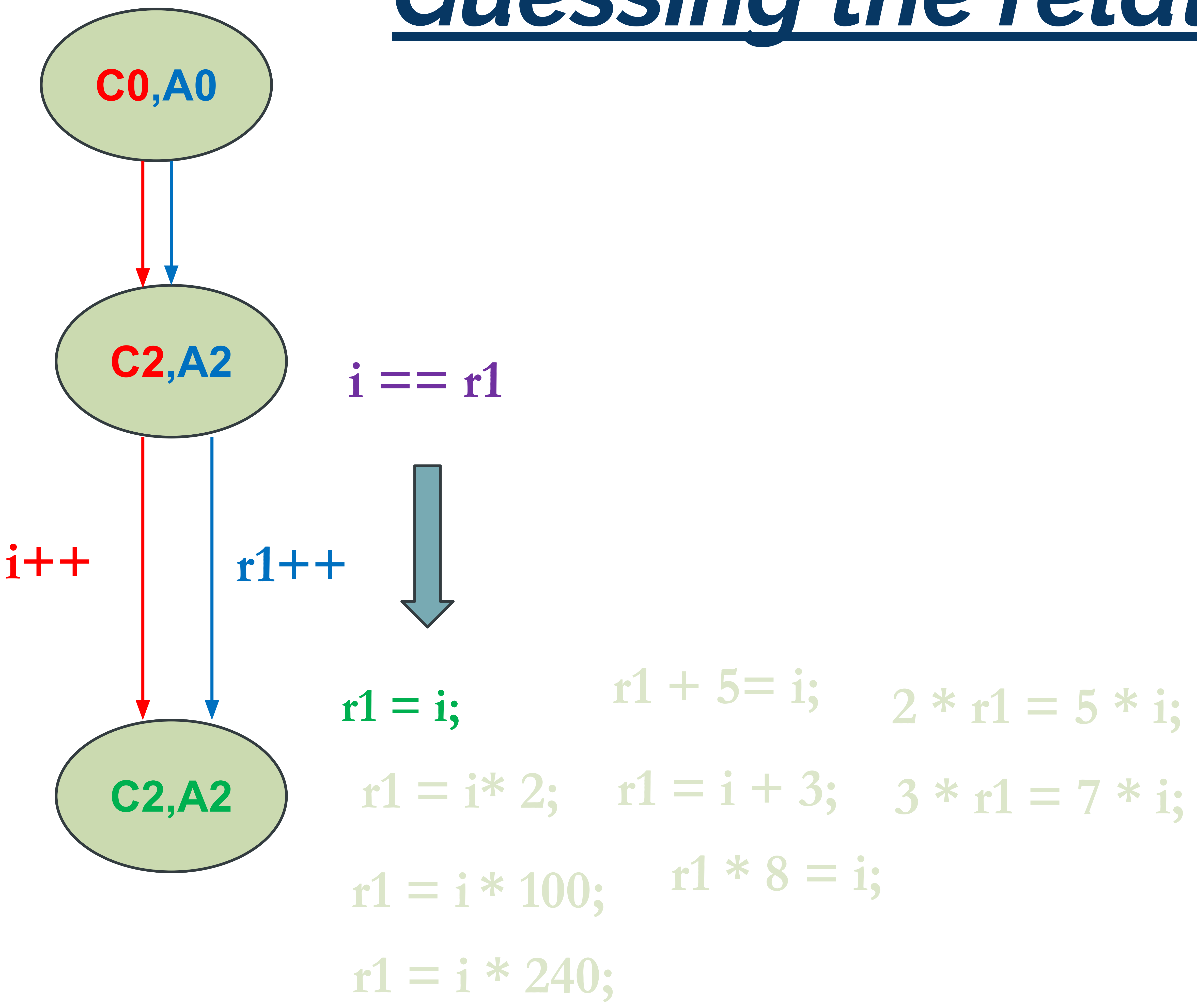
r1 = i* 1; r1 + 5= i; 2 * r1 = 5 * i;

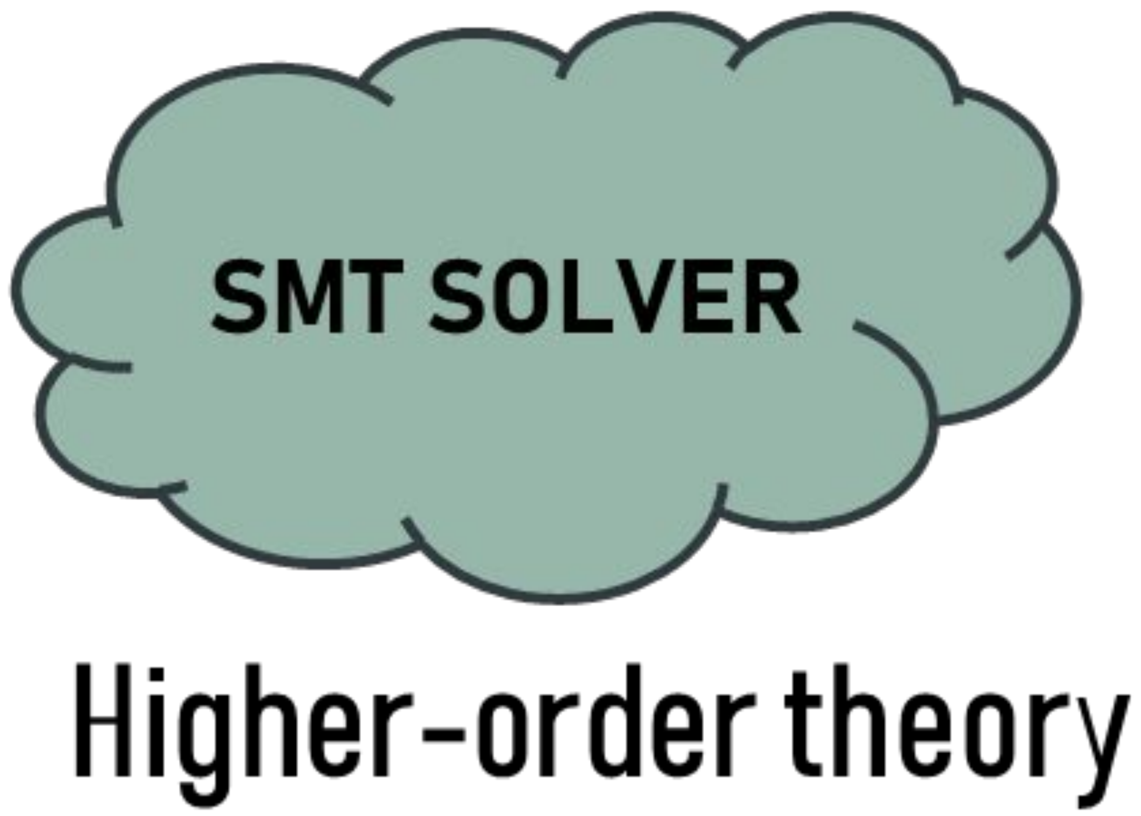
r1 = i* 2; r1 = i + 3; 3 * r1 = 7 * i;

r1 = i * 100; r1 * 8 = i;

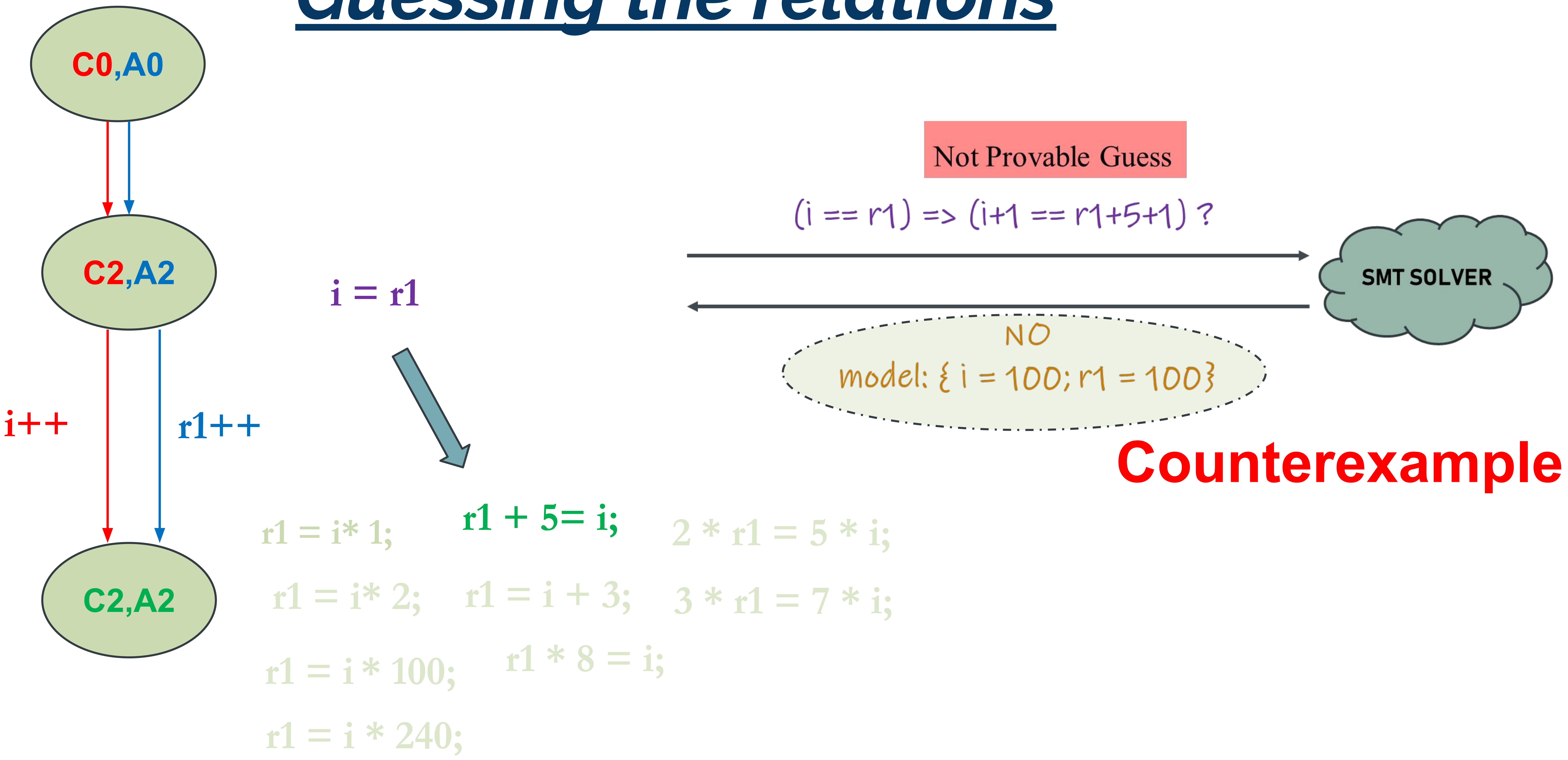
r1 = i * 240;

Guessing the relations





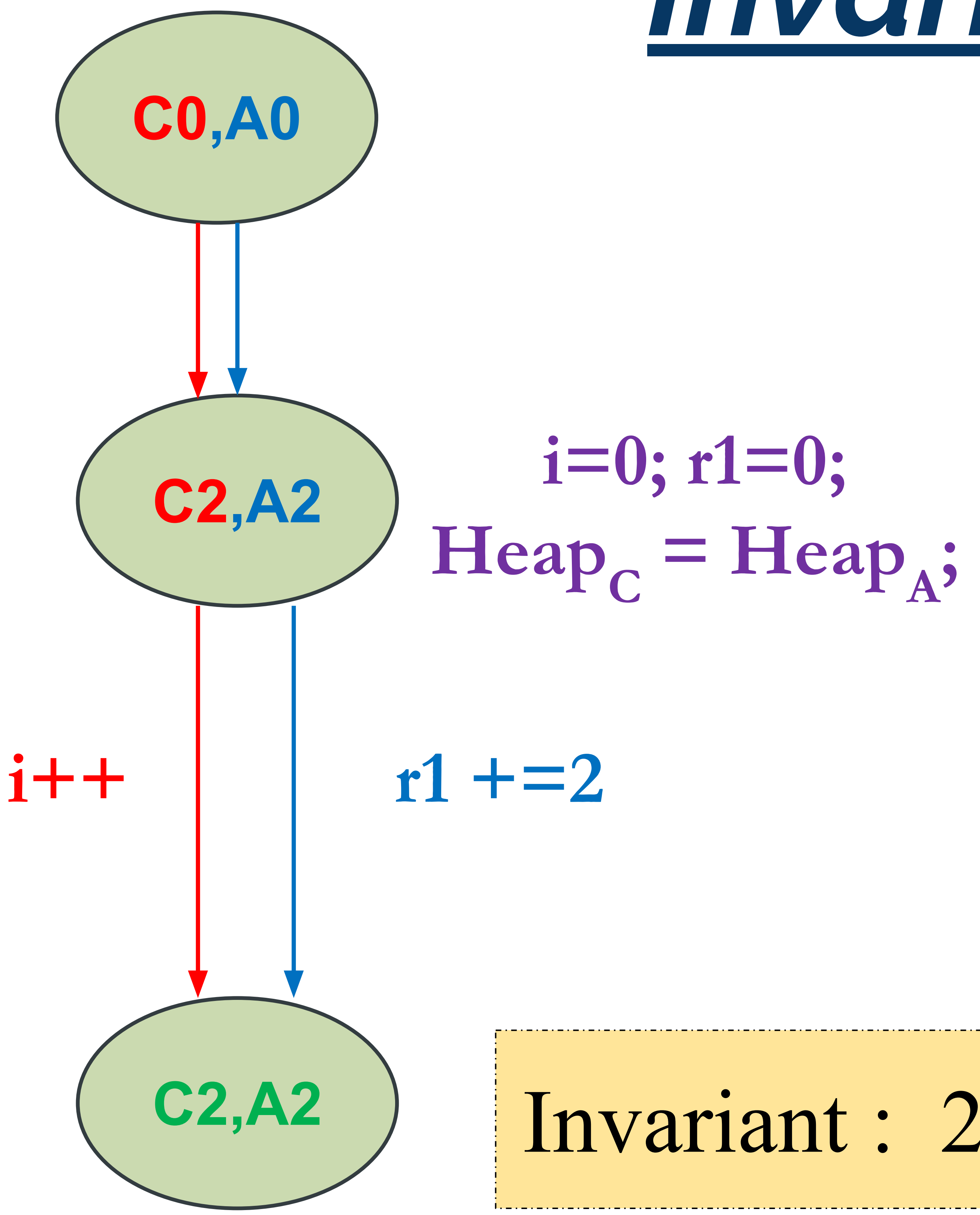
Guessing the relations



Research Contributions

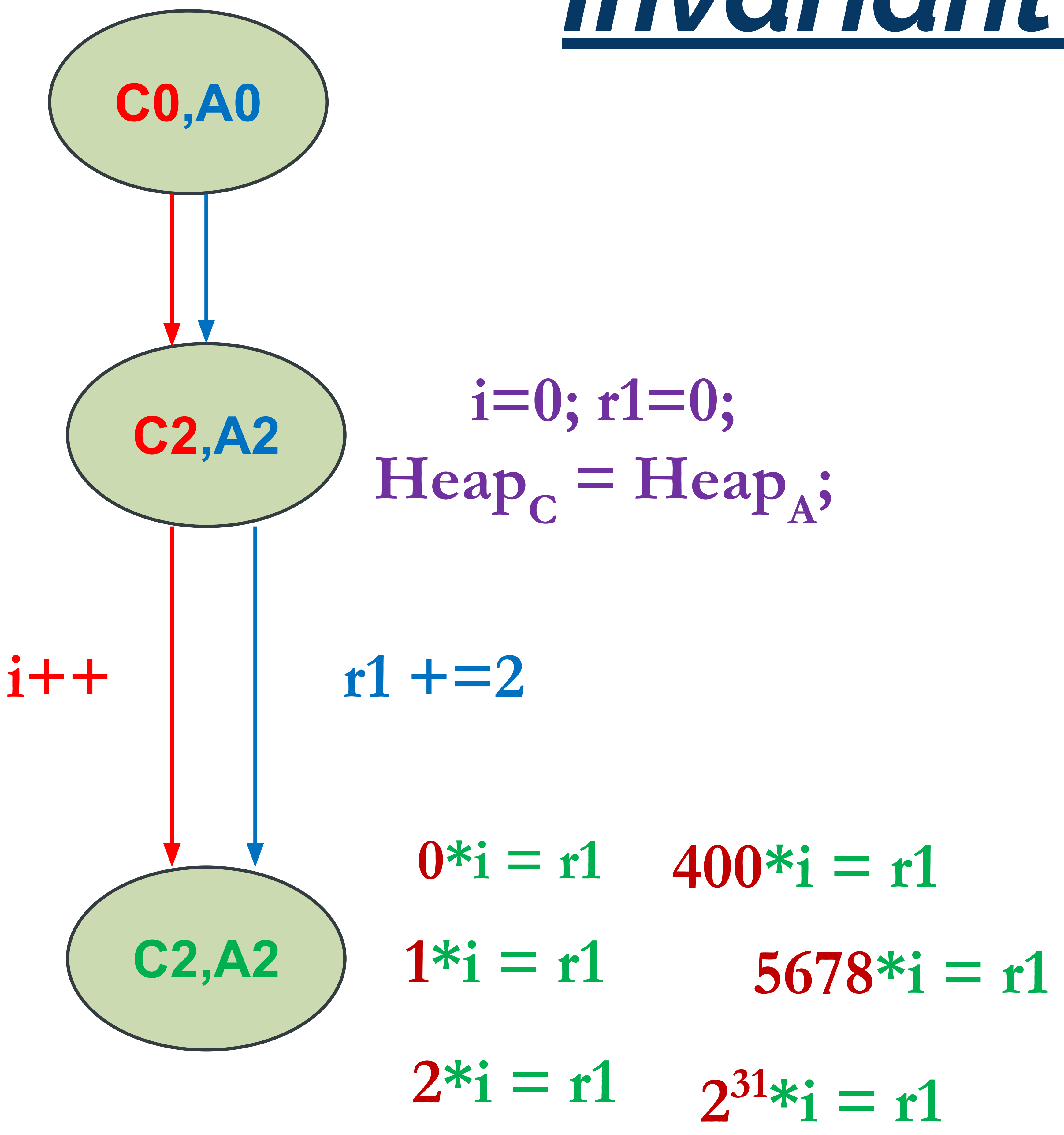
Counterexample Guided
Invariant Inference

Invariant Inference



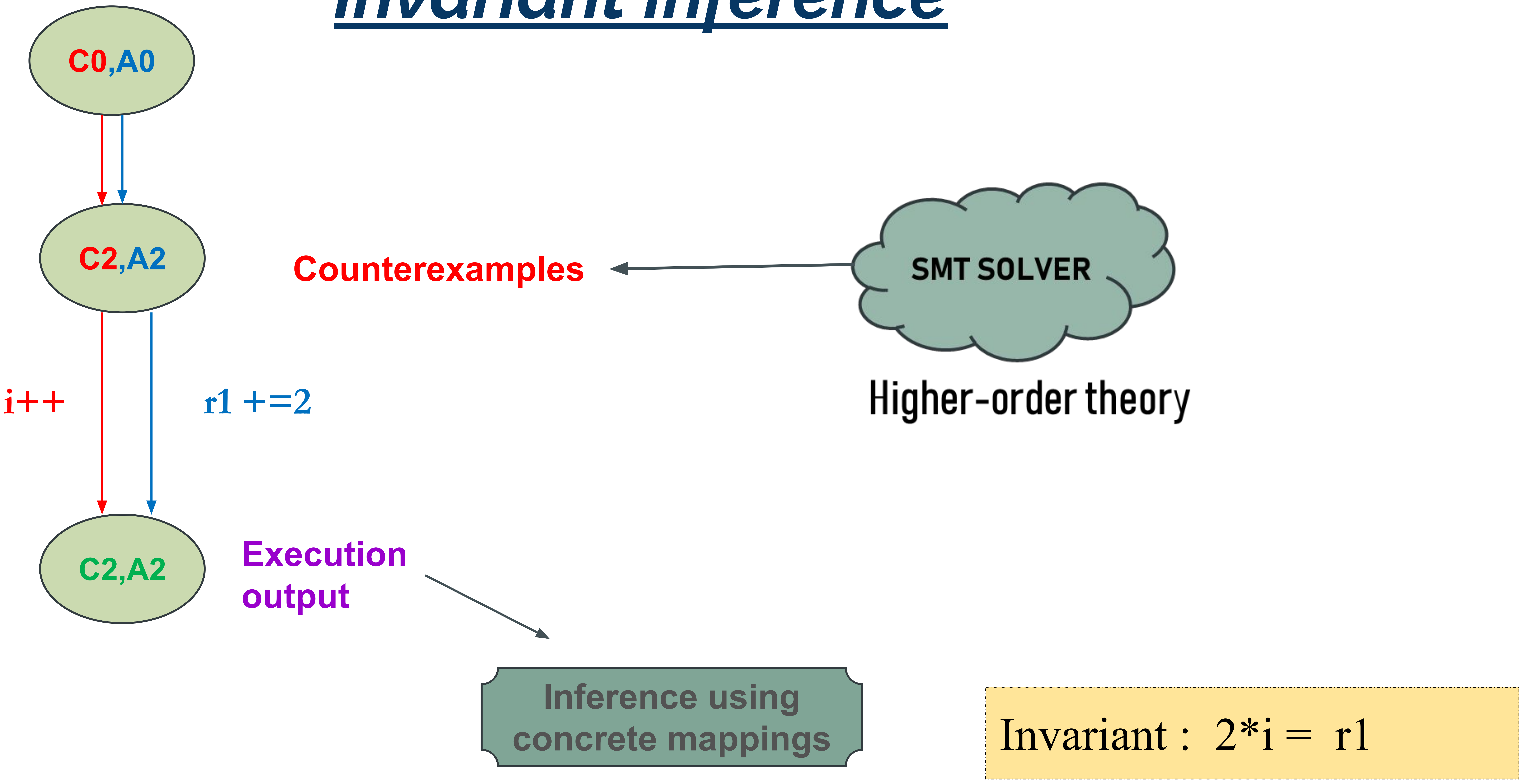
Invariant : $2*i = r1$

Invariant Inference

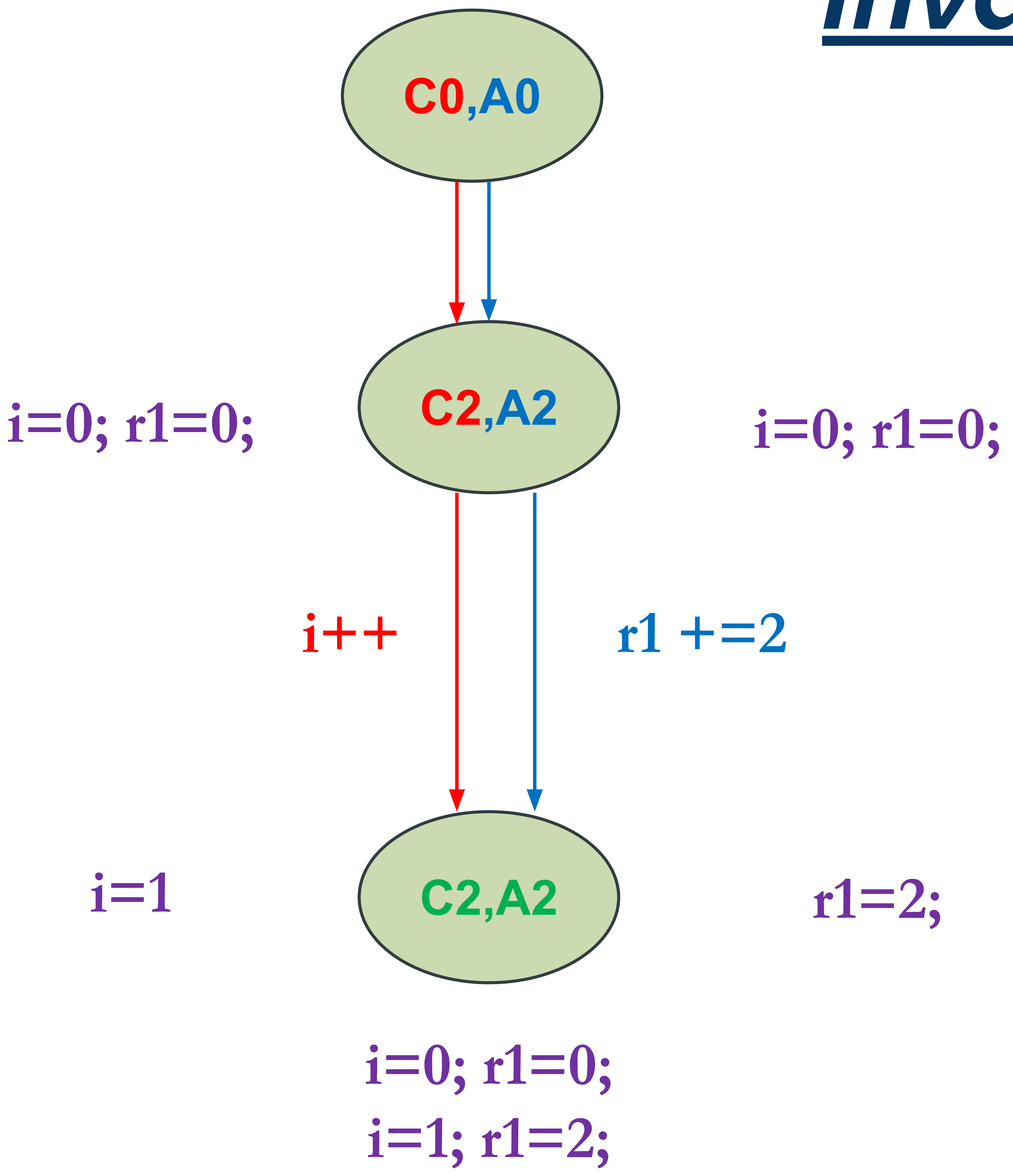


Invariant : 2*i = r1

Invariant Inference

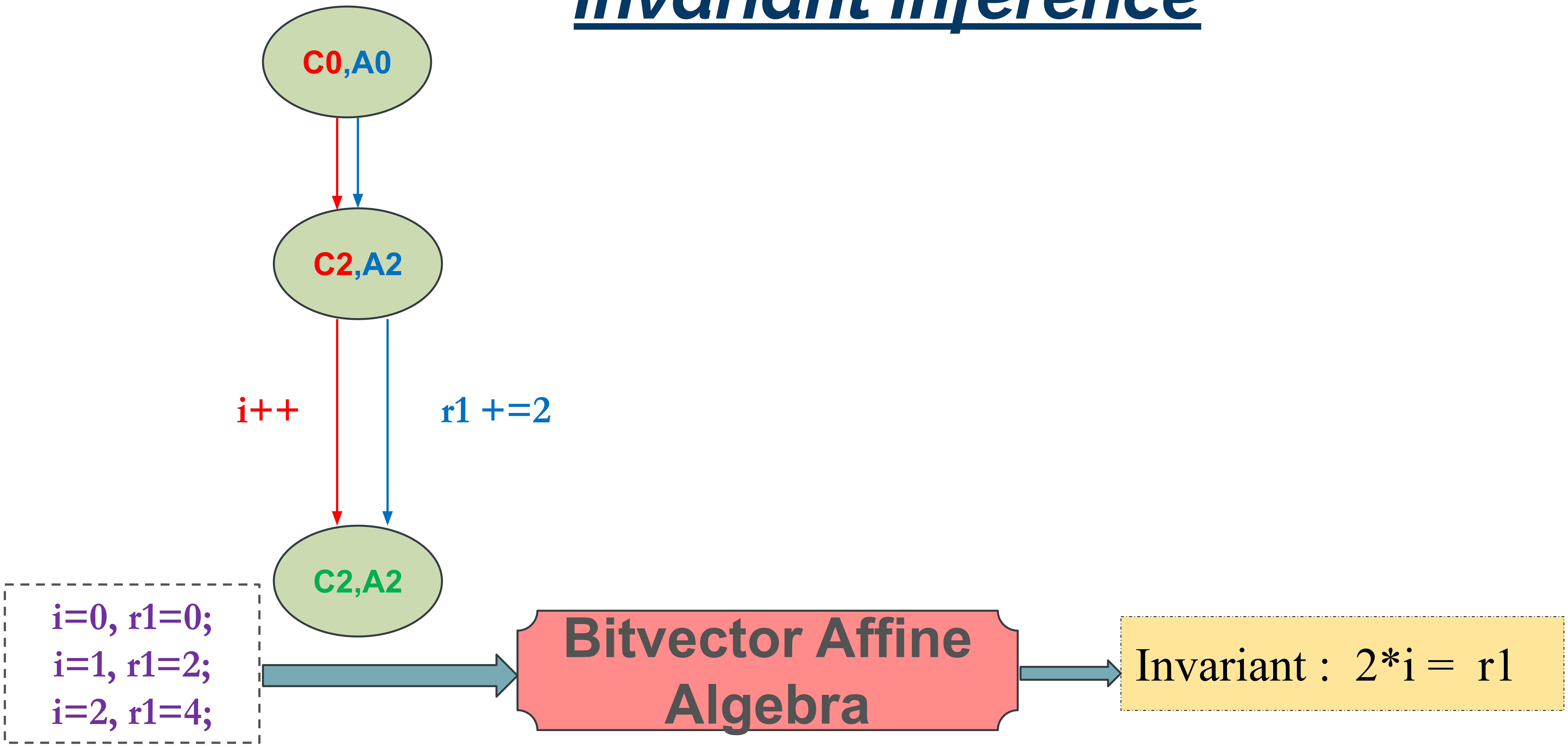


Invariant Inference



Invariant : 2*i = r1

Invariant Inference

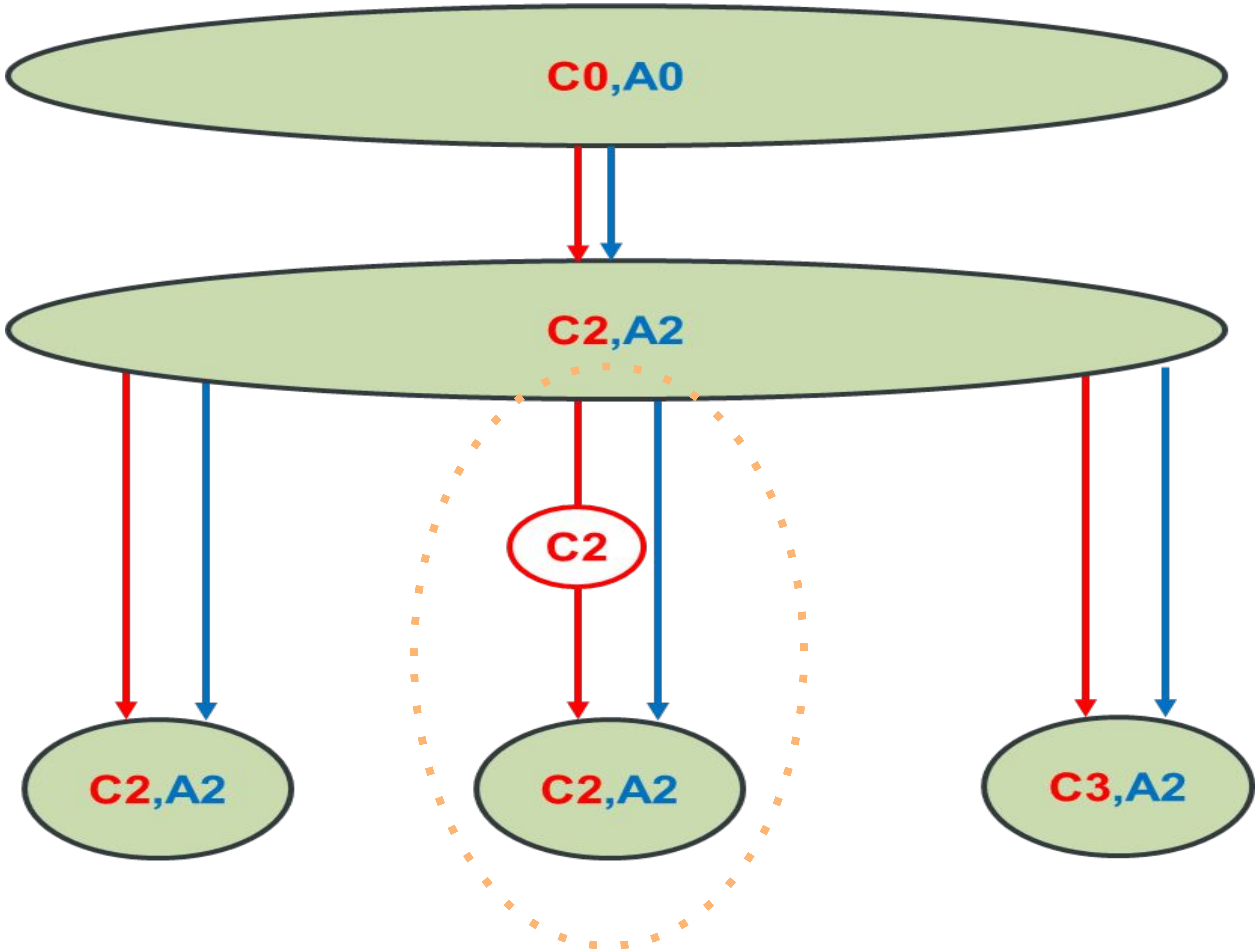


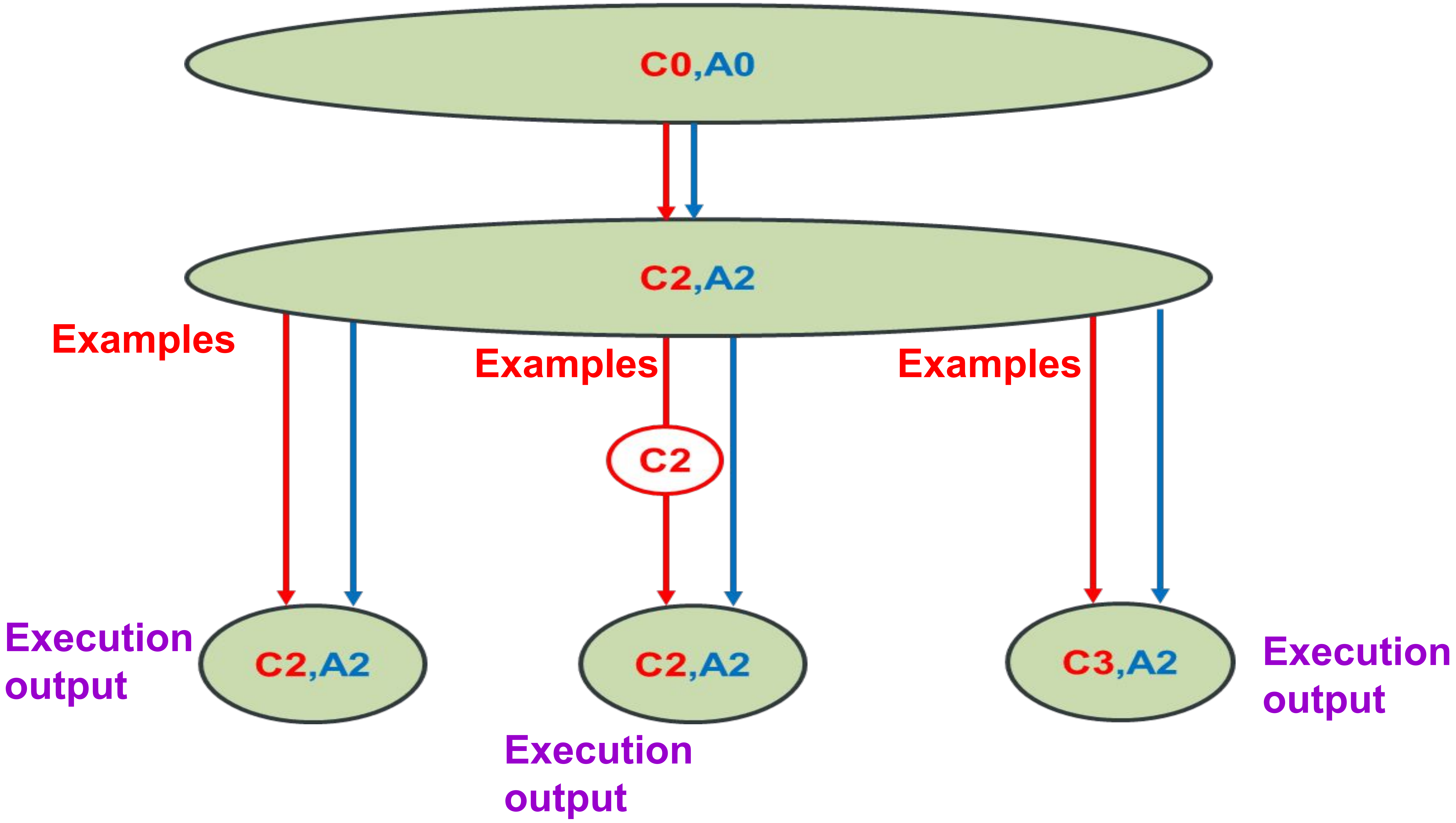
Effective use of SMT solvers for Program Equivalence Checking through Invariant-Sketching and Query-Decomposition (SAT2018)

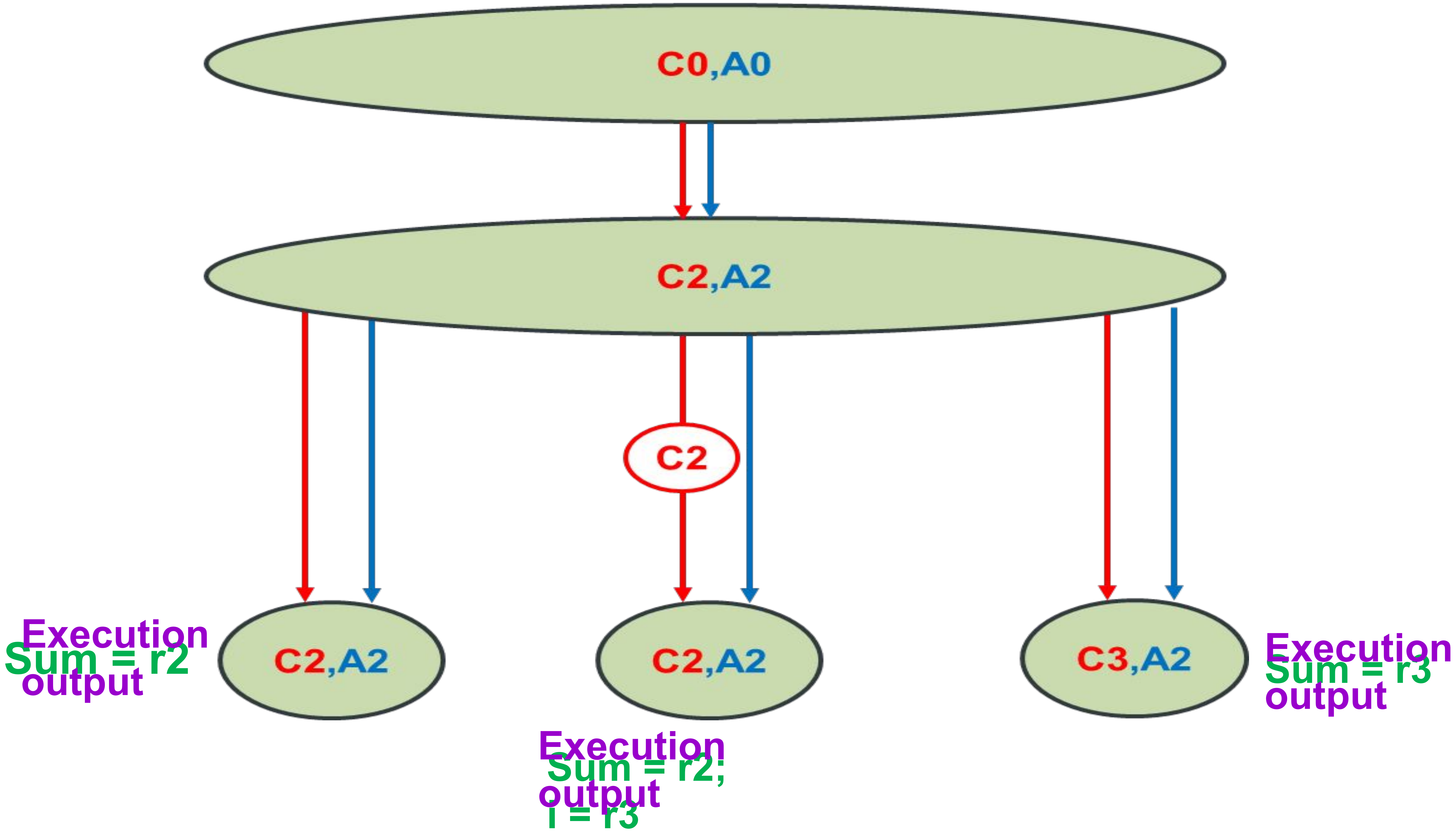
Shubhani Gupta, Aseem Saxena, Anmol Mahajan, Sorav Bansal
Indian Institute Of Technology Delhi

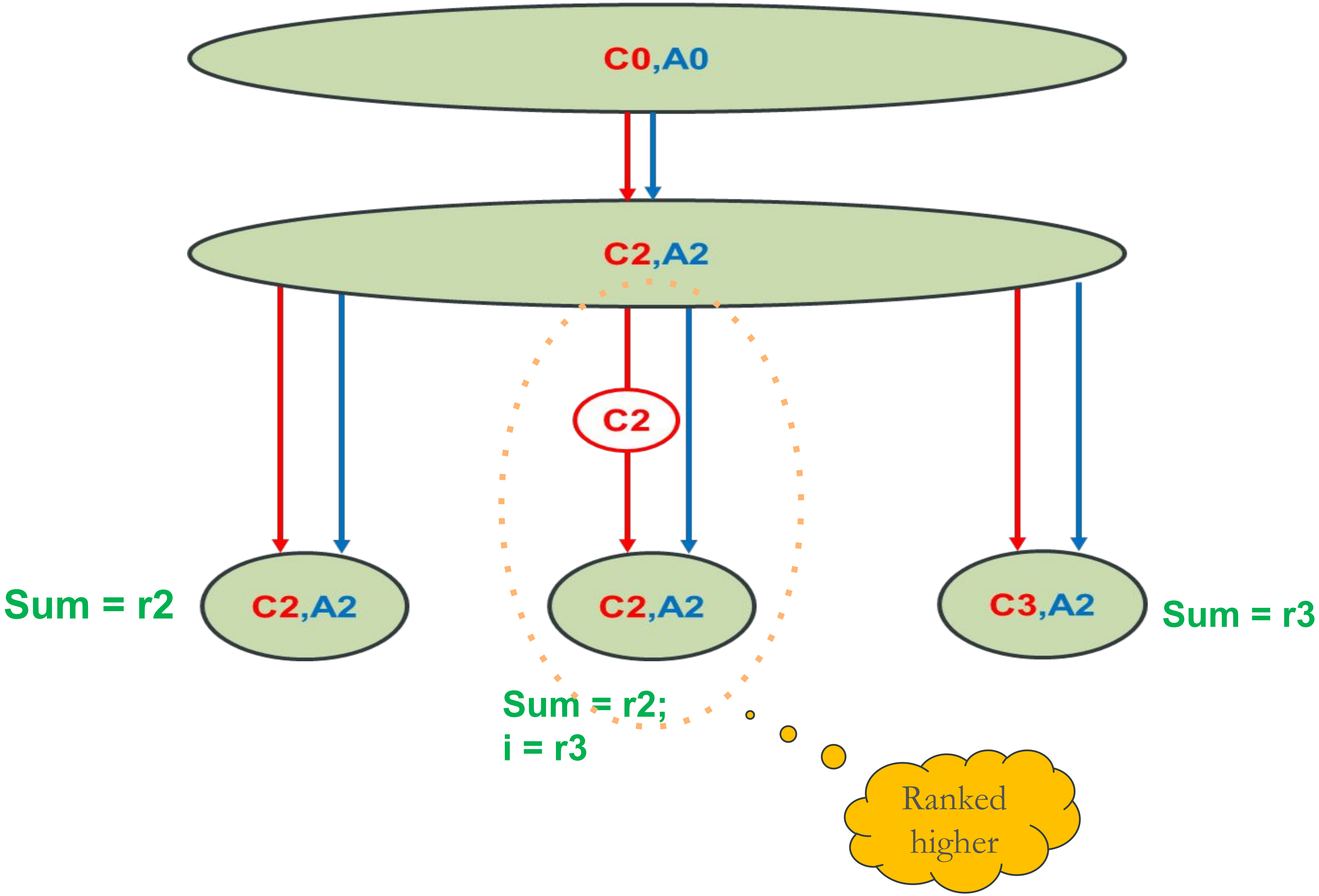
https://doi.org/10.1007/978-3-319-94144-8_22

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

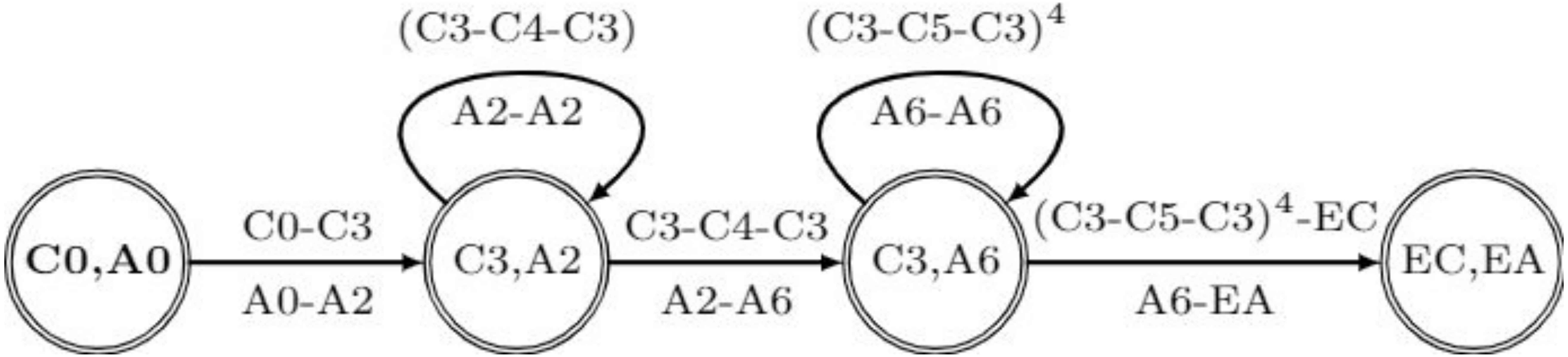
```
int loopUnrolling() {
   int sum = 0;
   int mid = LEN /2;
   for ( int i = 0; i < mid ; i ++) {
     sum += c[a[i]];
   }
   for ( int i = mid; i < LEN ; i +=4) {
     sum += b[ i ];
     sum += b[ i+1 ];
     sum += b[ i +2];
     sum += b[ i +3];
   }
   return sum ;
}
```

```
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
```

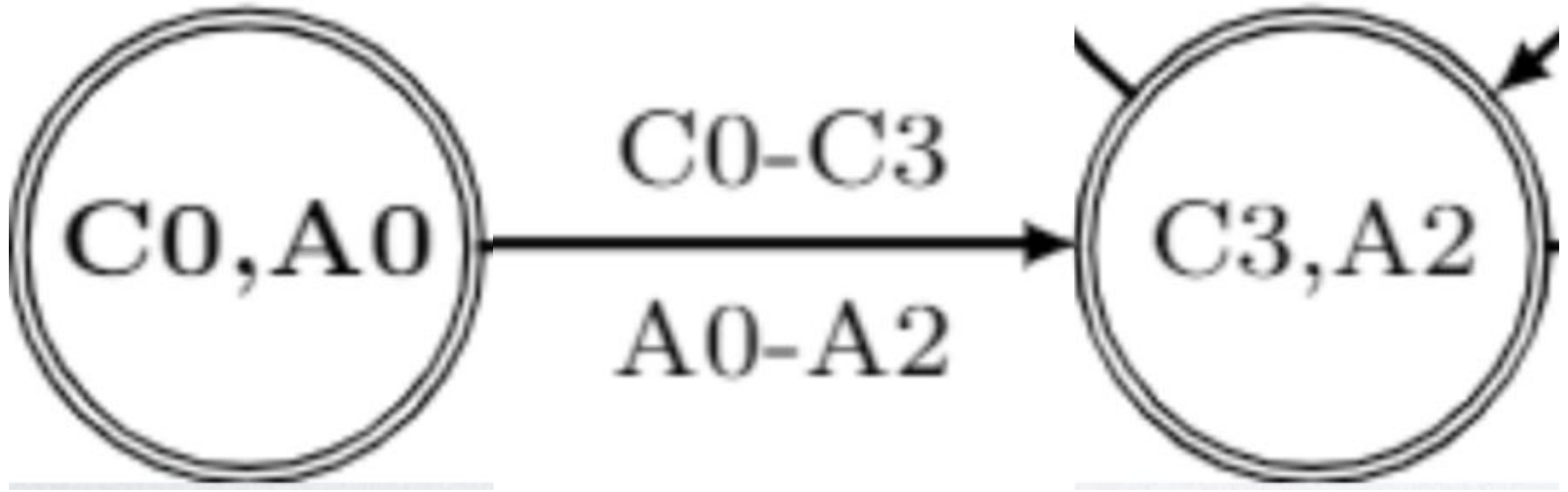


Incremental Construction of the Product CFG

Incremental Construction of the Product CFG

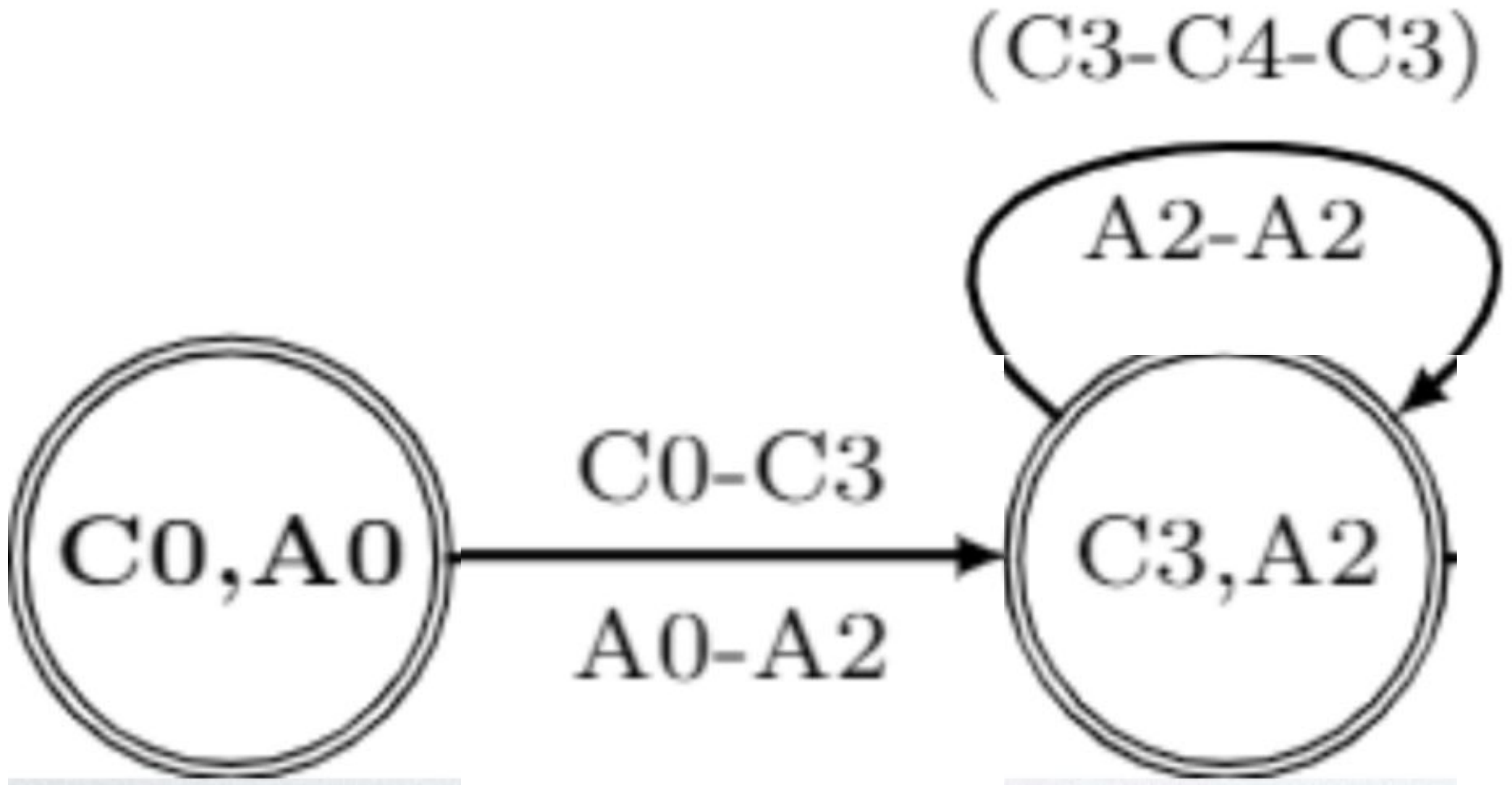
Incremental Construction of the Product CFG

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



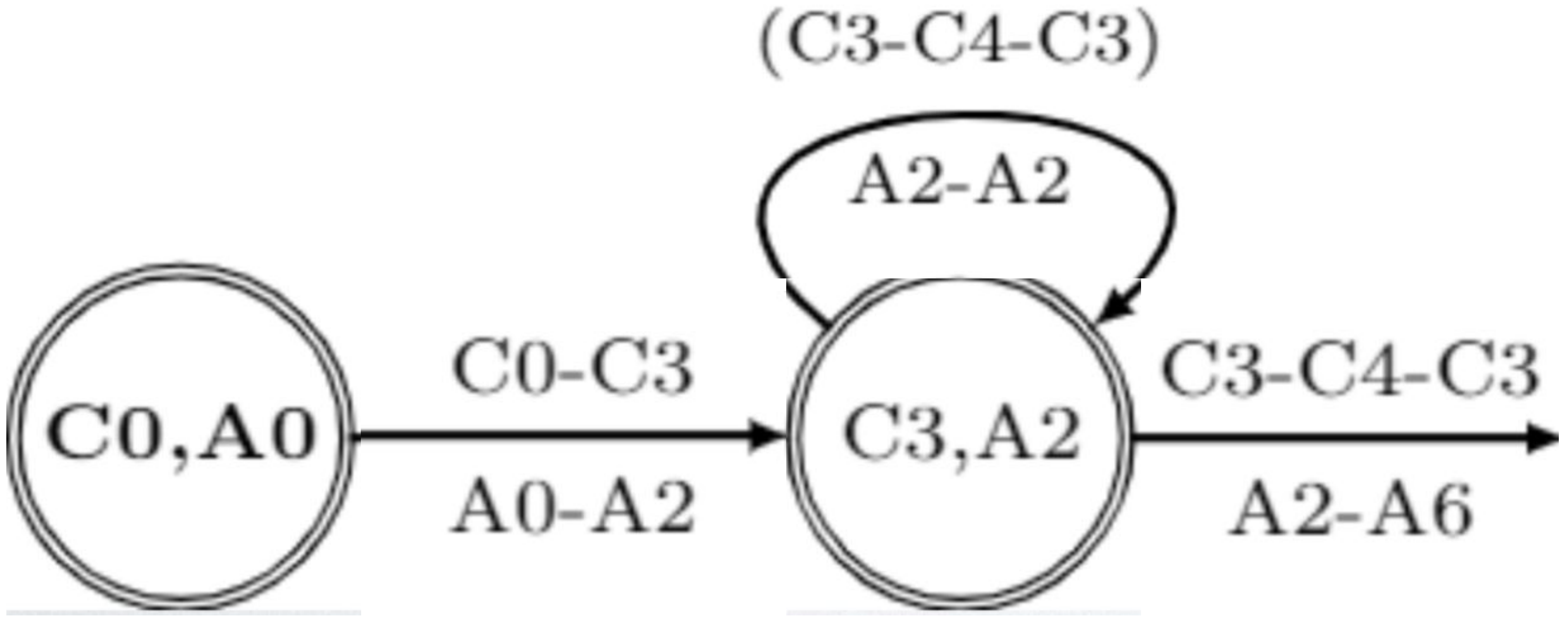
Infer Invariants at (C3,A2)

Incremental Construction of the Product CFG

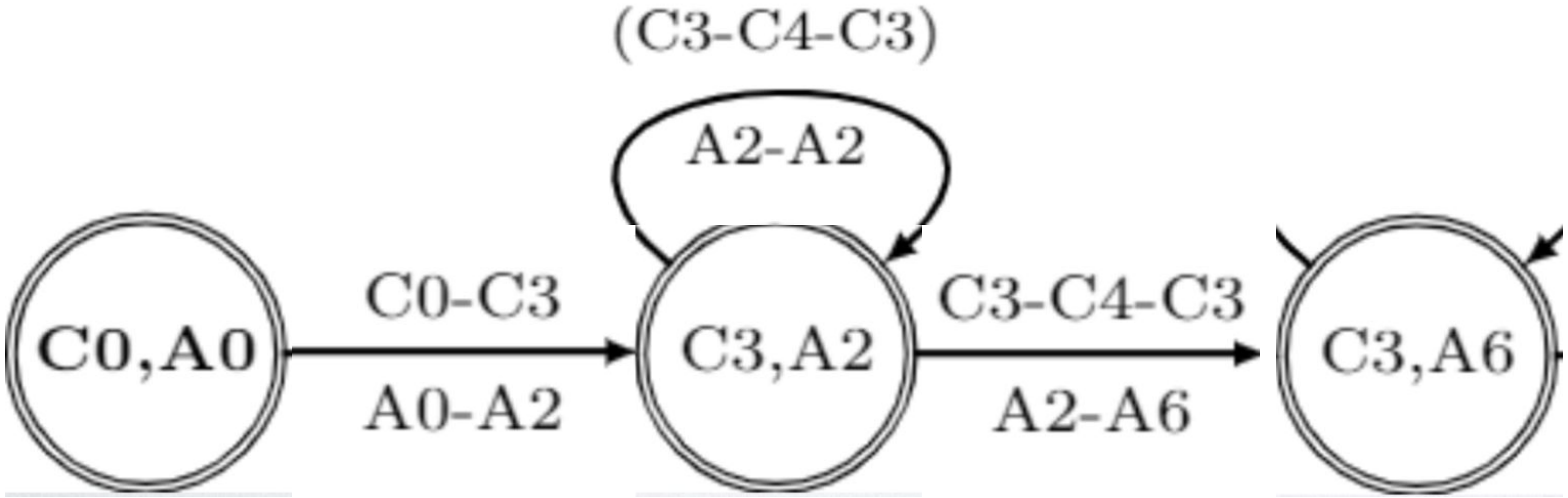


Relax Invariants
at (C3,A2)

Incremental Construction of the Product CFG

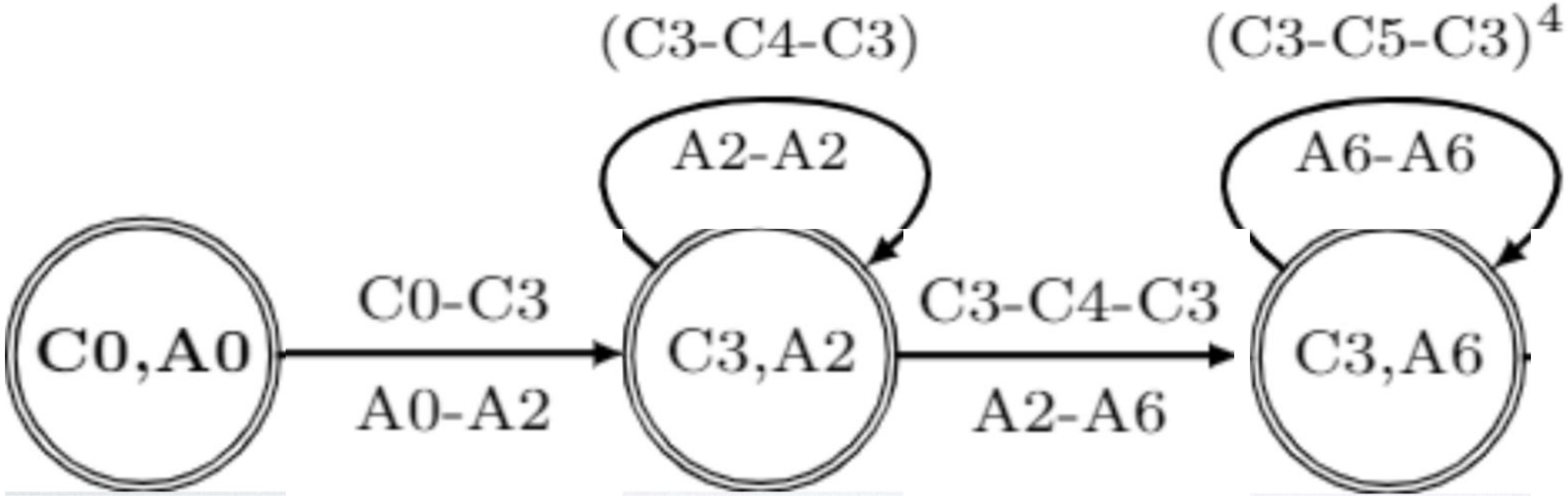


Incremental Construction of the Product CFG



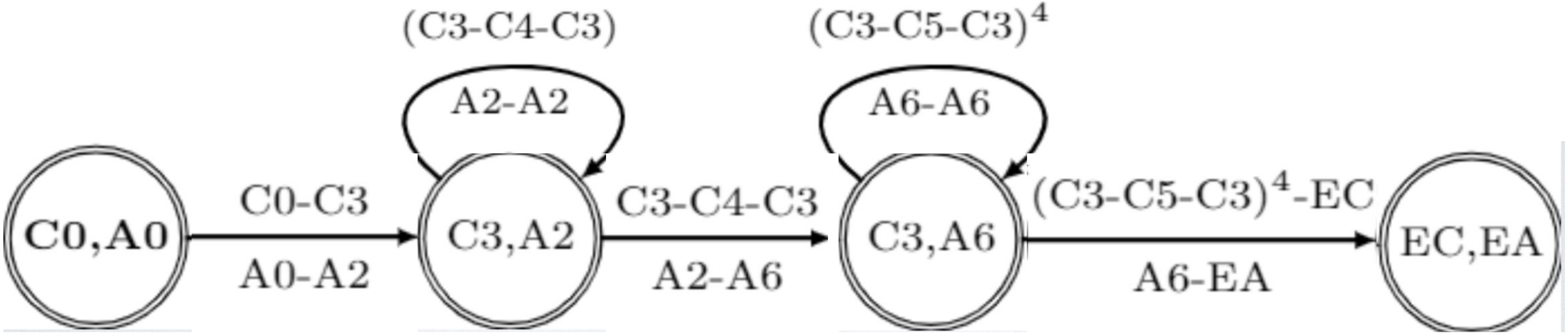
Infer Invariants at (C3,A6)

Incremental Construction of the Product CFG



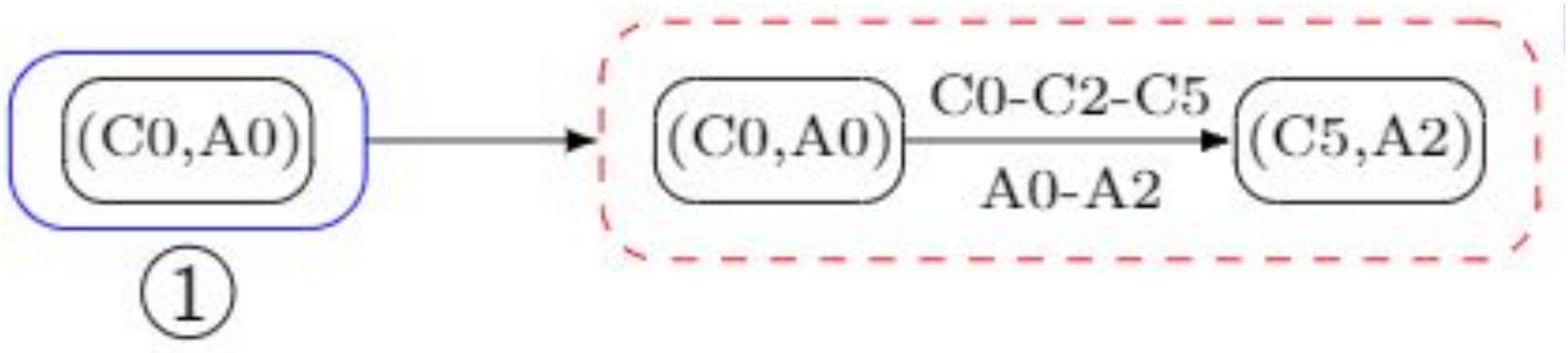
Relax Invariants
at (C3,A6)

Incremental Construction of the Product CFG

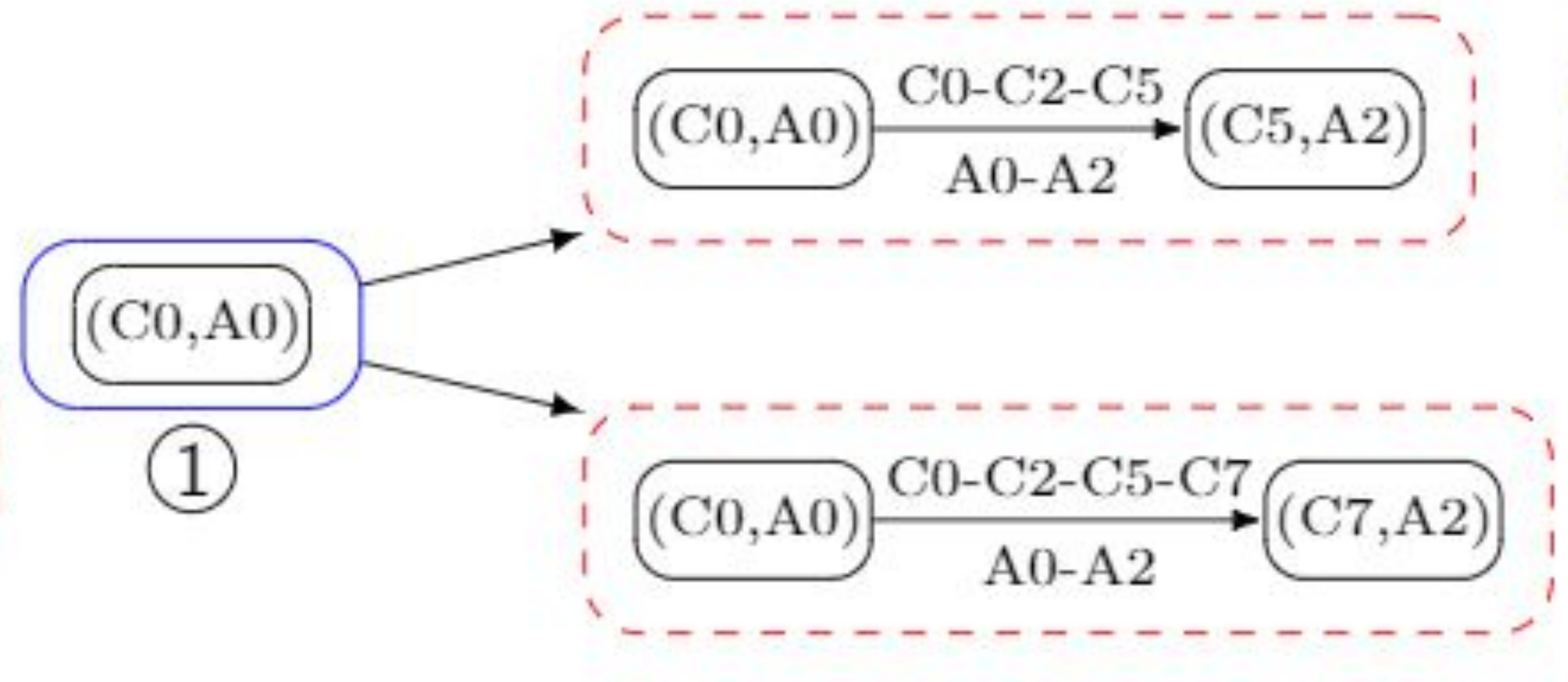


Check equivalence of return values under inferred invariants

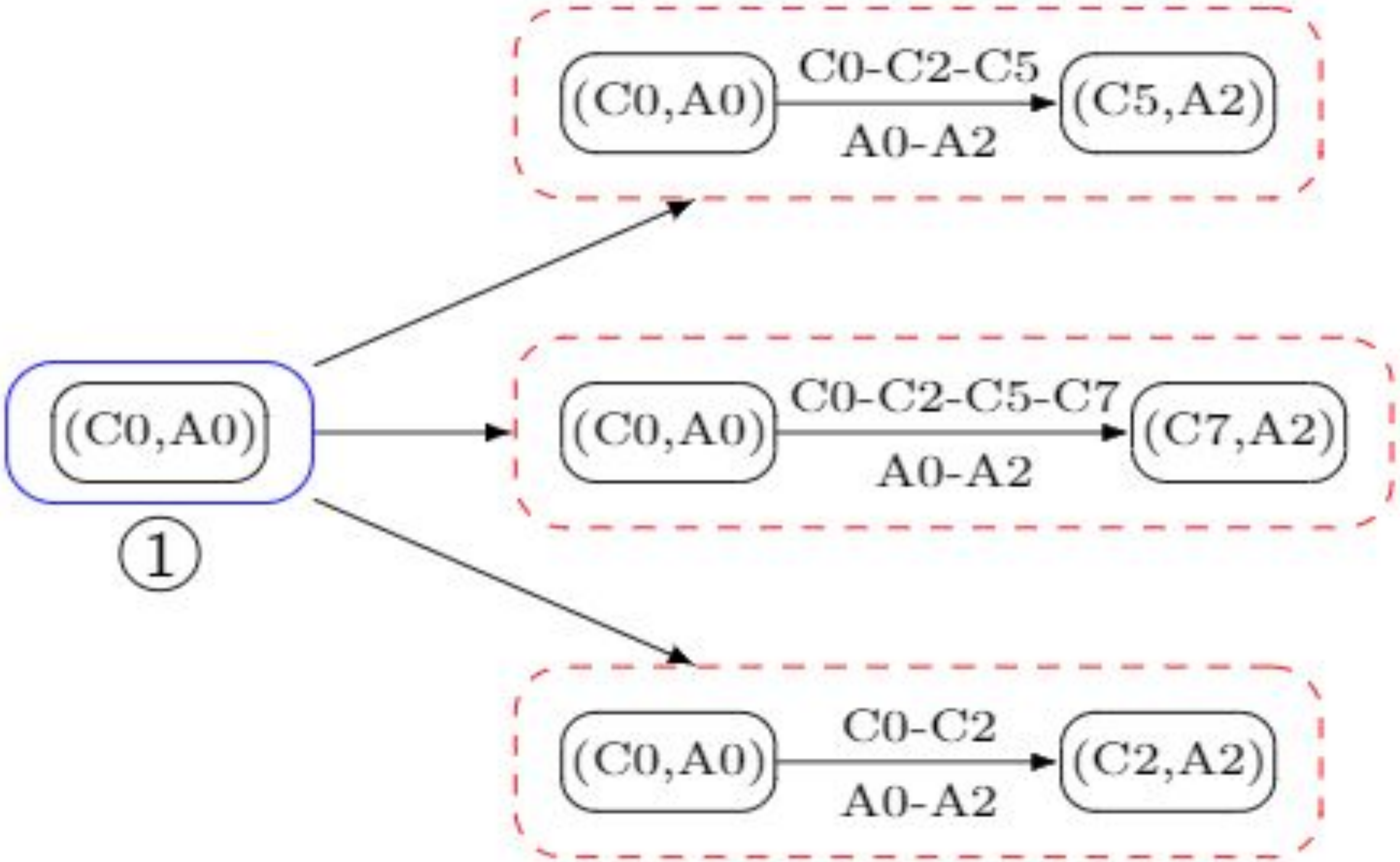
SEARCH SPACE



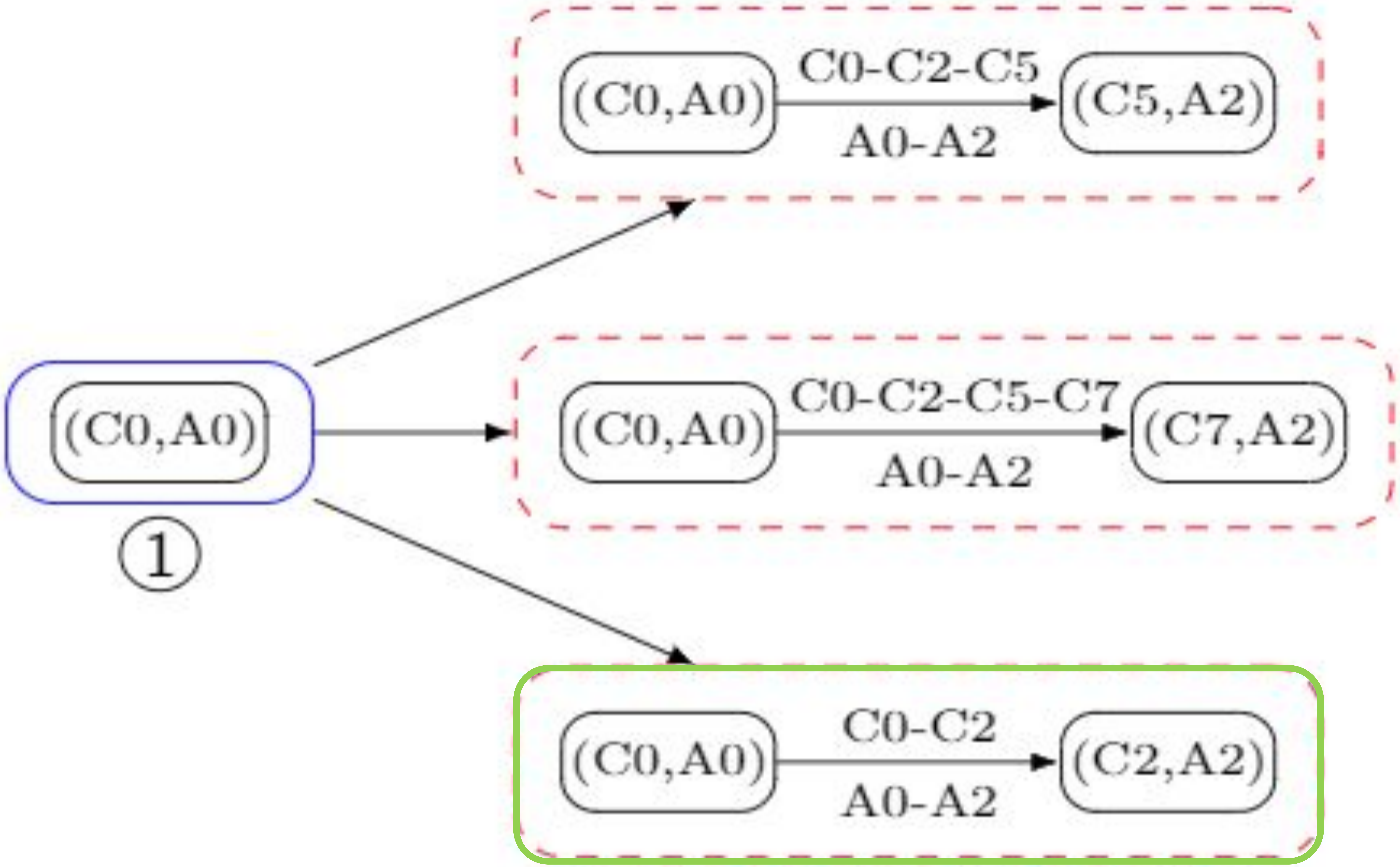
SEARCH SPACE



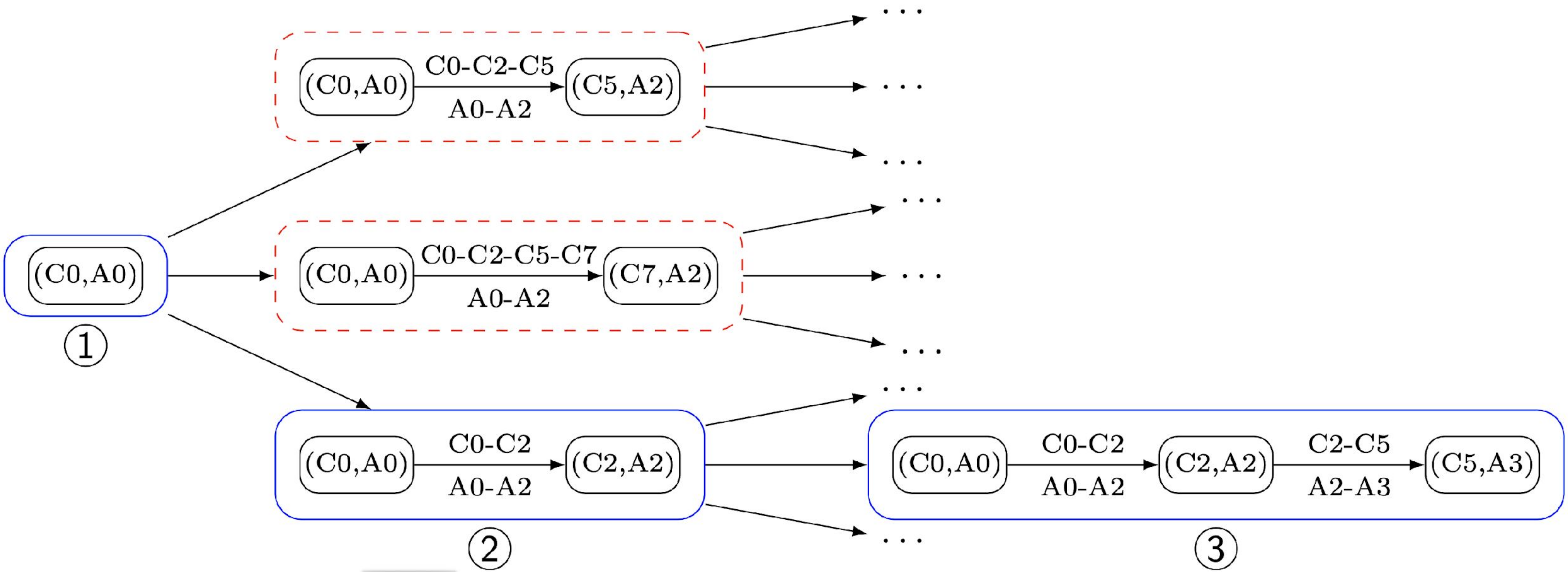
SEARCH SPACE



SEARCH SPACE

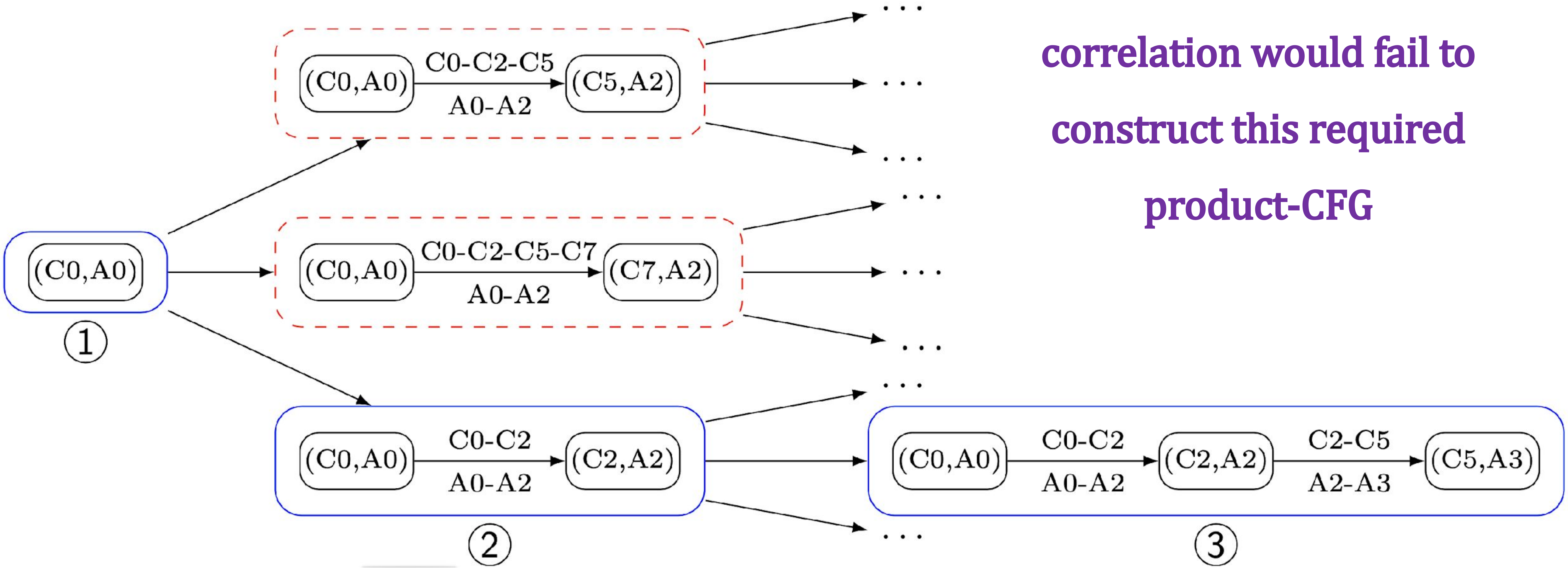


SEARCH SPACE



Exhaustive search would take millions of years to compute equivalence

SEARCH SPACE



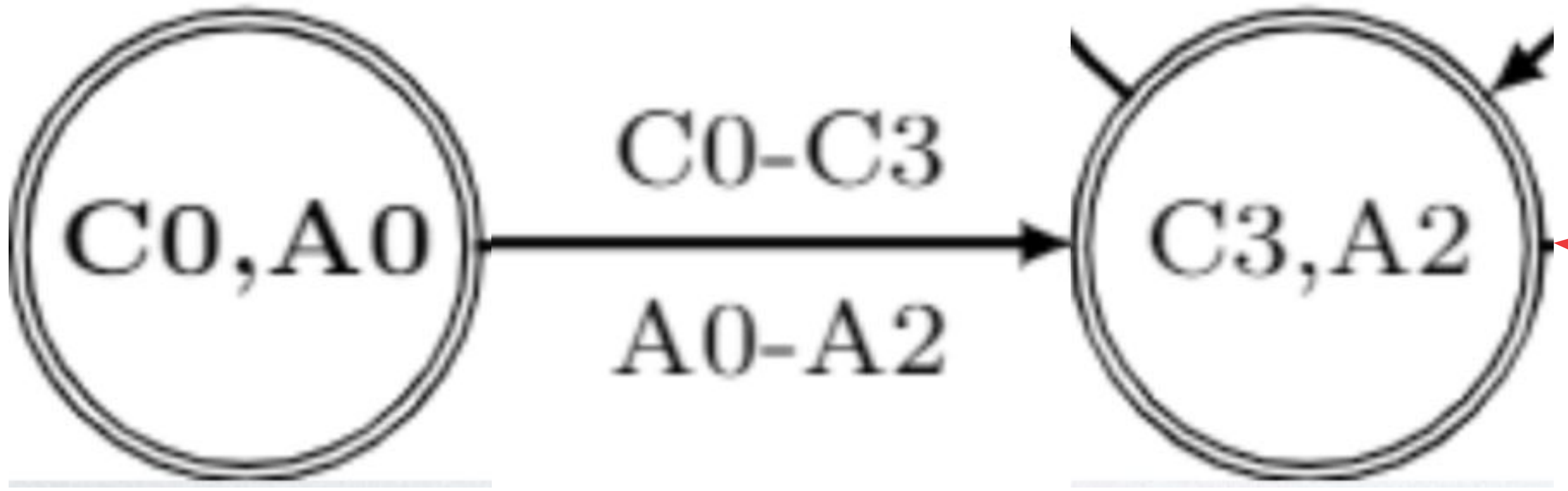
Prior work on data driven correlation would fail to construct this required product-CFG

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



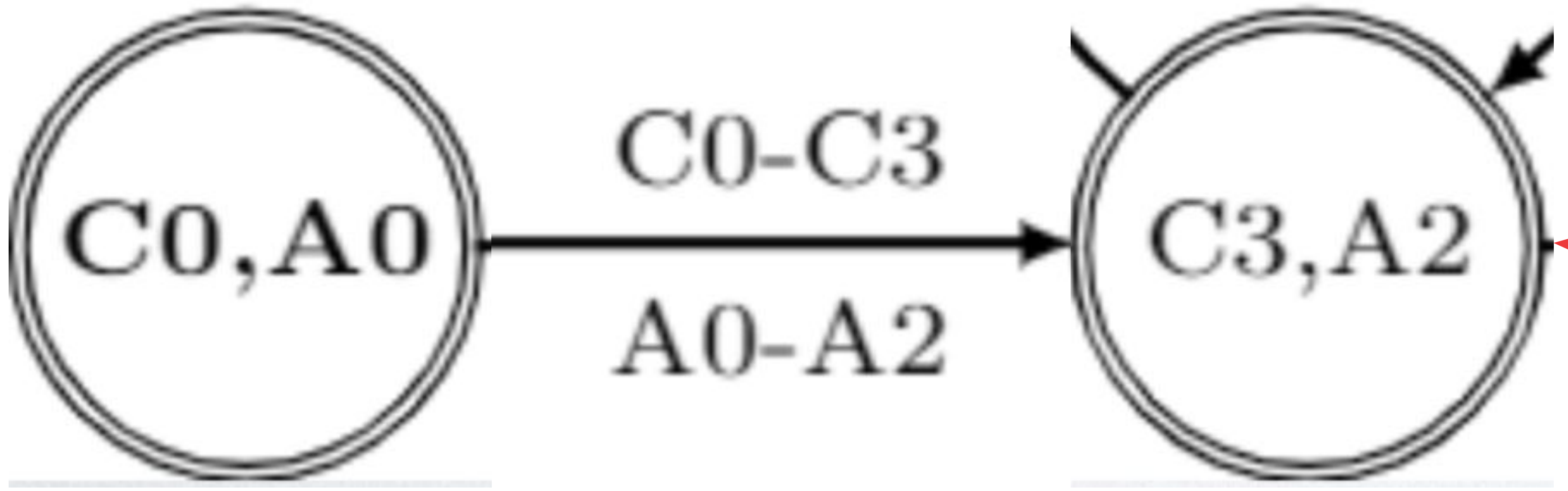
Counterexamples identified at (C3,A2) during invariant inference

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.

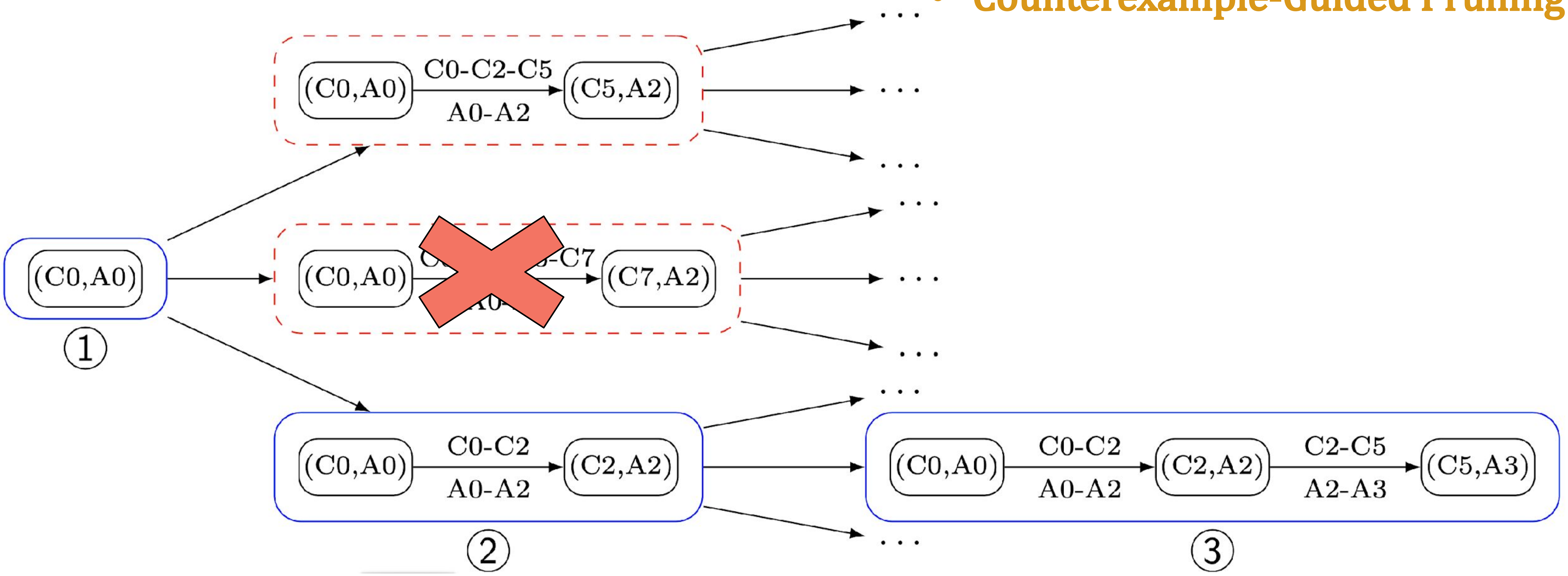


Counterexample1: I=5, R1=5
Counterexample2: I=1003, R1=1003

Infer Invariants at (C3,A2)

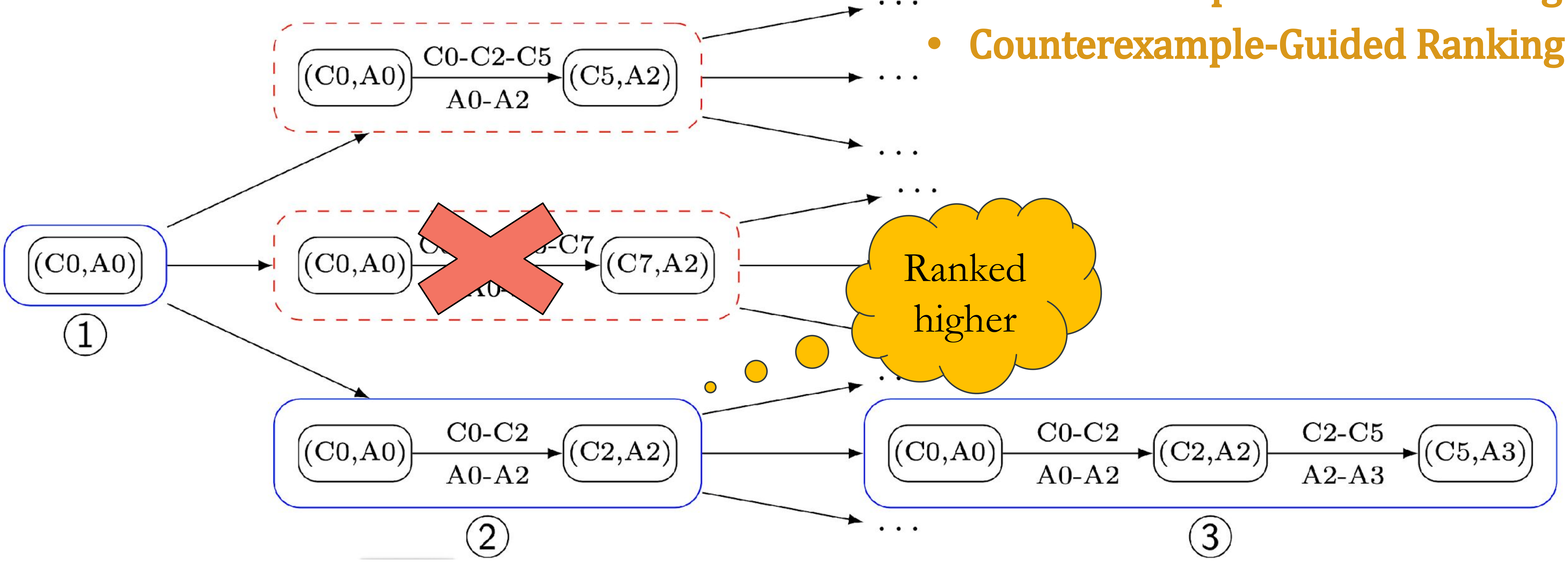
COUNTEREXAMPLE GUIDED BEST-FIRST SEARCH

- Counterexample-Guided Pruning

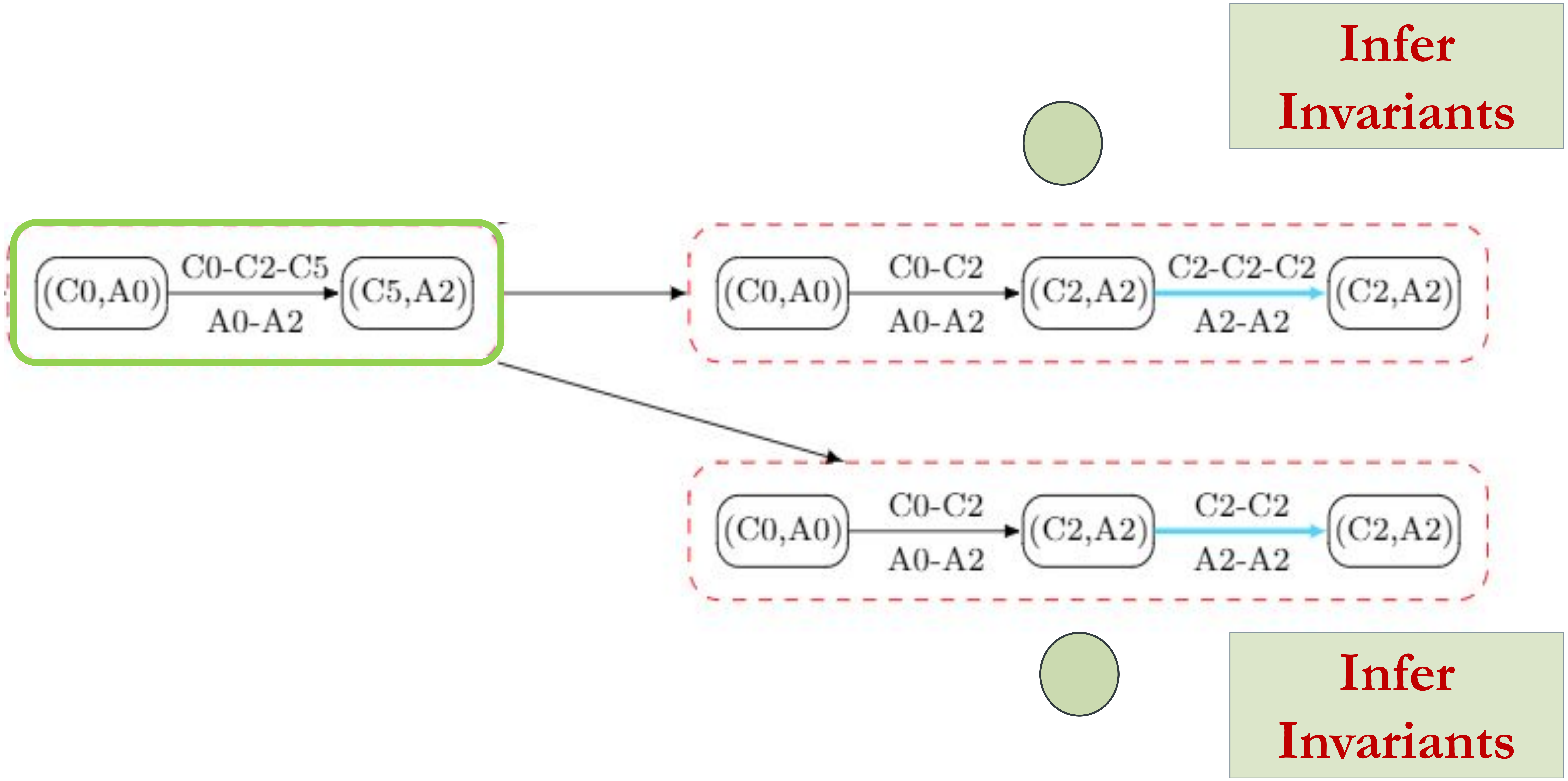


COUNTEREXAMPLE GUIDED BEST-FIRST SEARCH

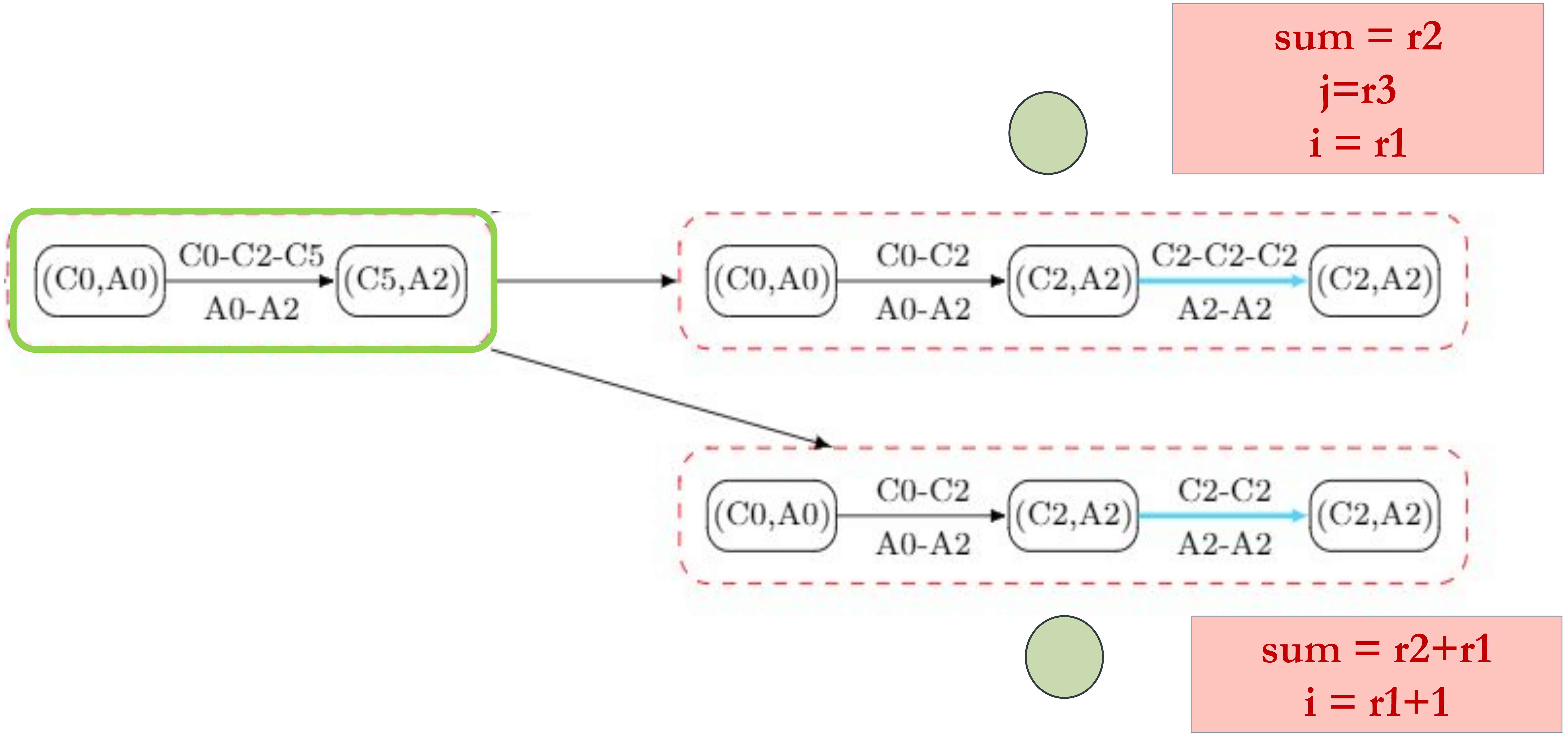
- Counterexample-Guided Pruning
- Counterexample-Guided Ranking



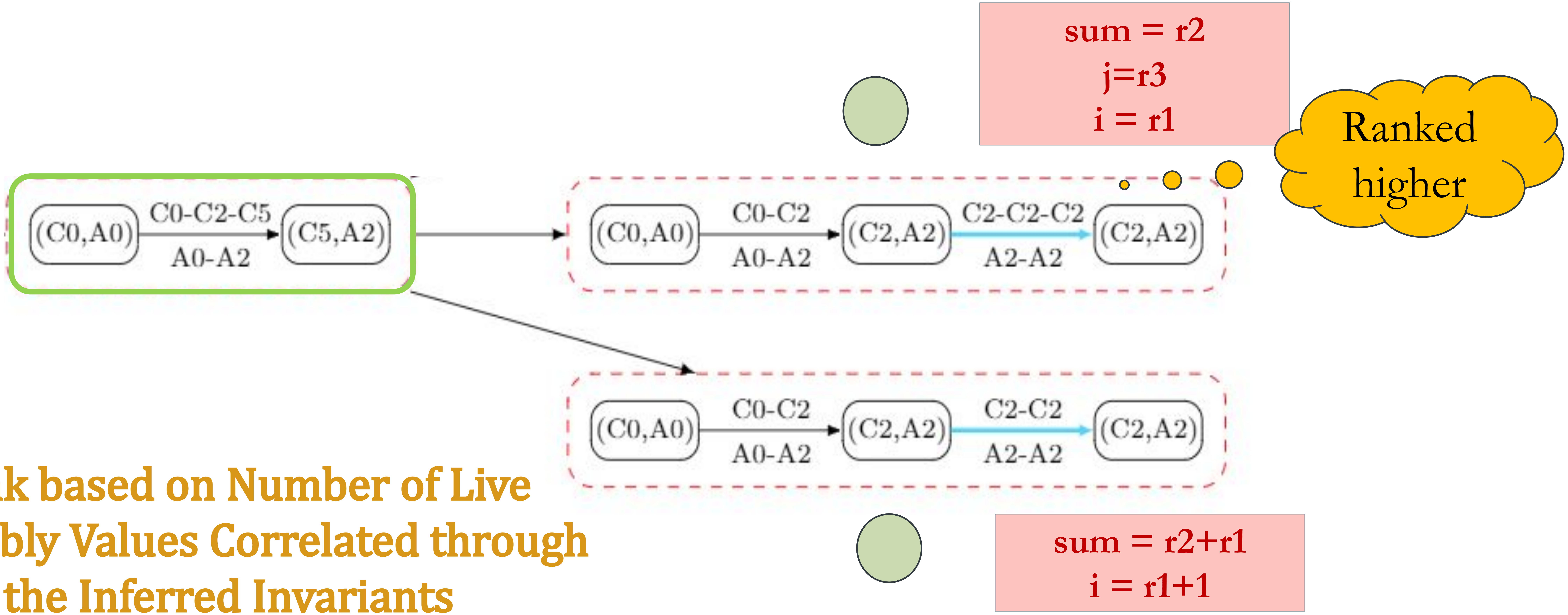
Infer Invariant Covers for Executed Counterexamples



Infer Invariant Covers for Executed Counterexamples

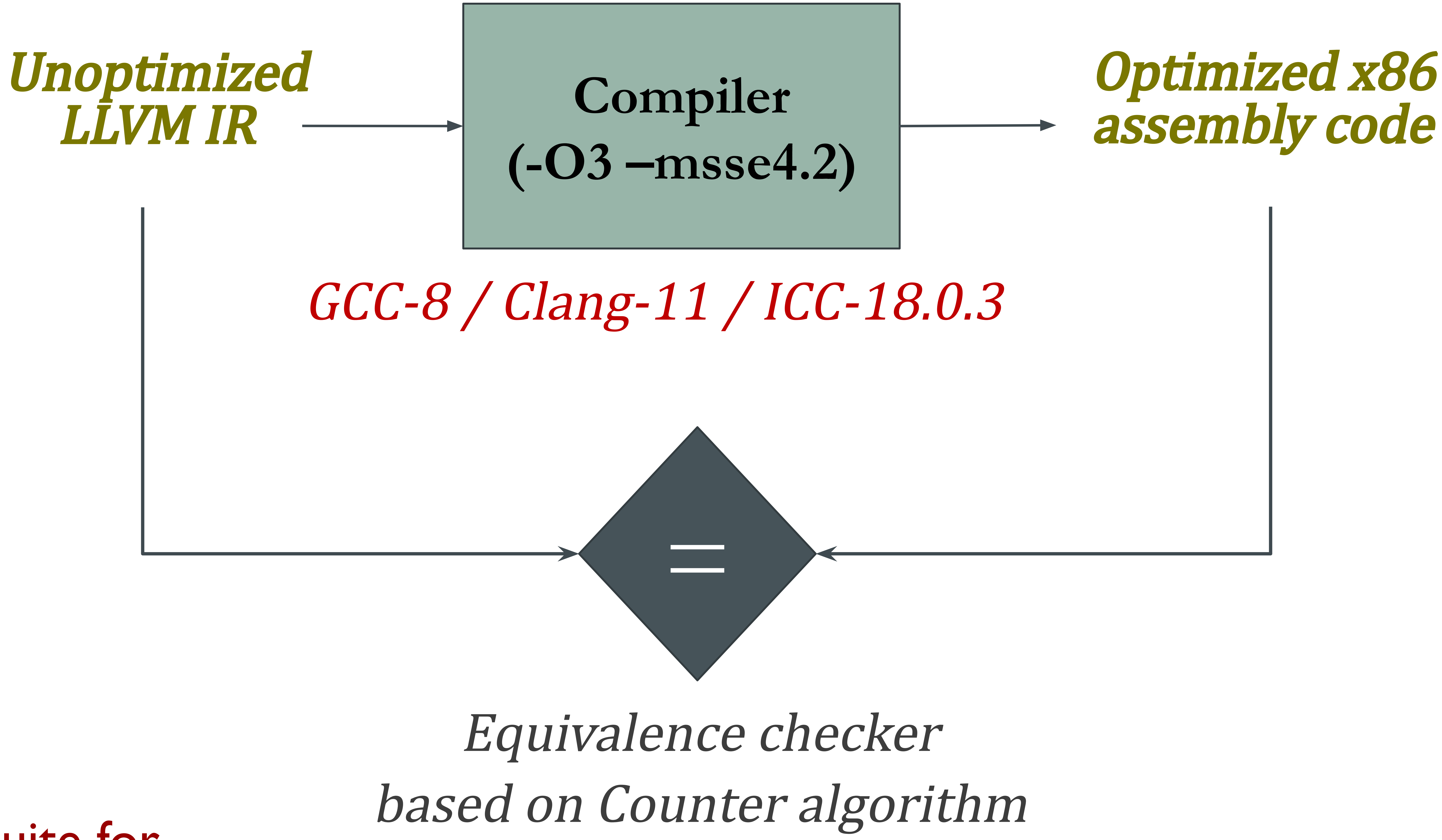


Infer Invariant Covers for Executed Counterexamples



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
- Three bugs in DietLibc related to missing unsigned-to-signed typecasts
- Bug in the Yices SMT Solver related to incorrect query result

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:  }
```

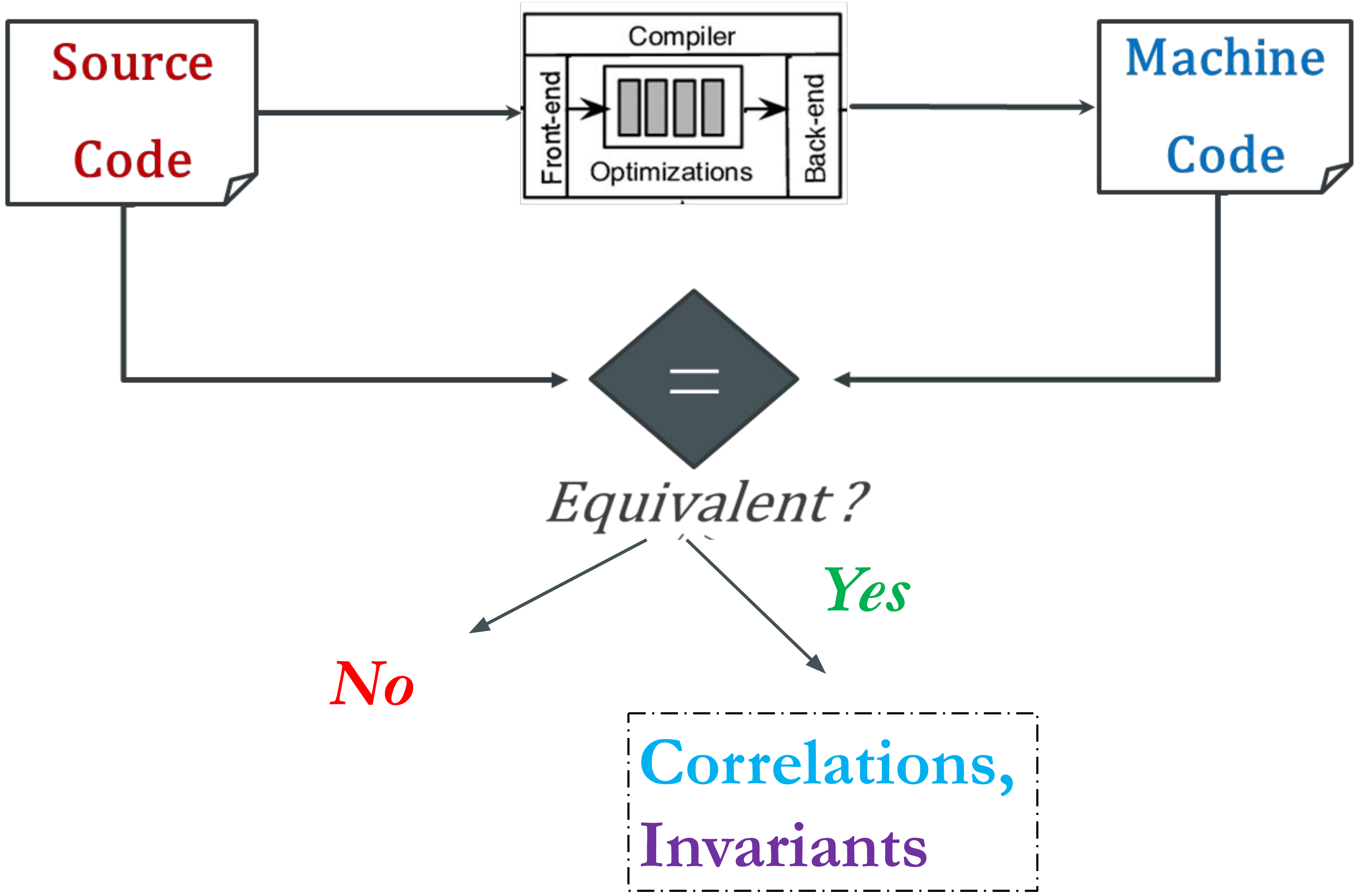
C Program

```
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
```

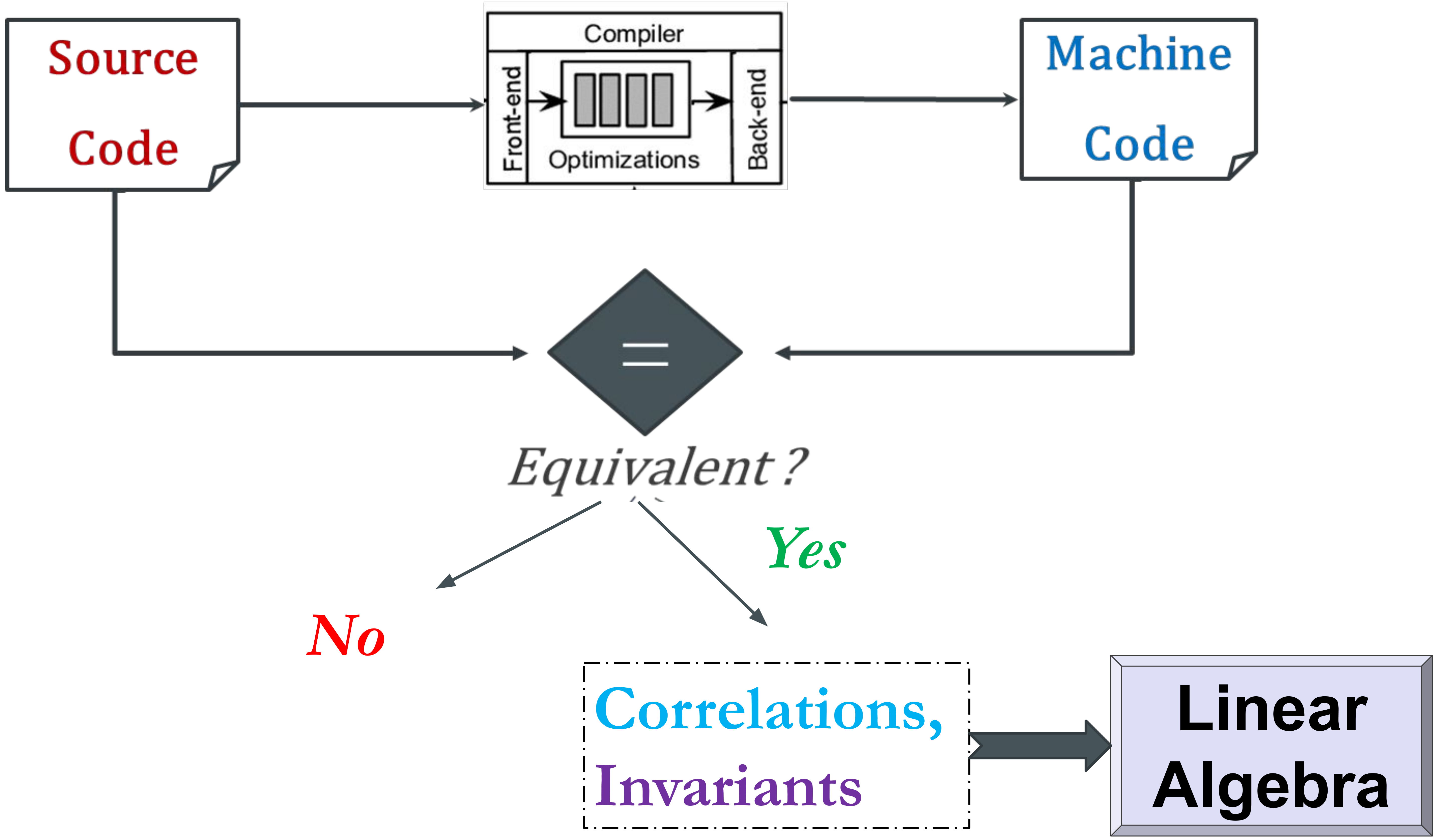
(abstracted) Assembly

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

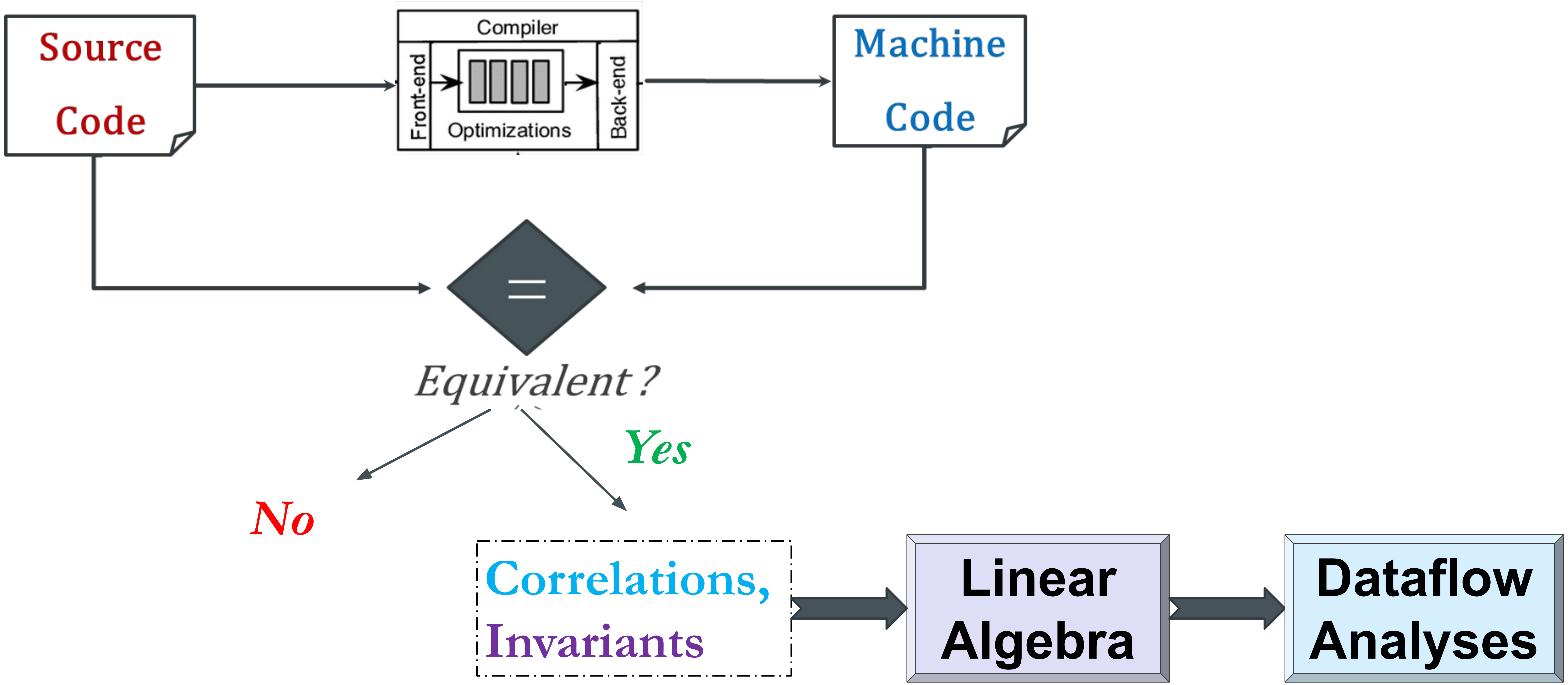
Automatic Generation of Debug Headers



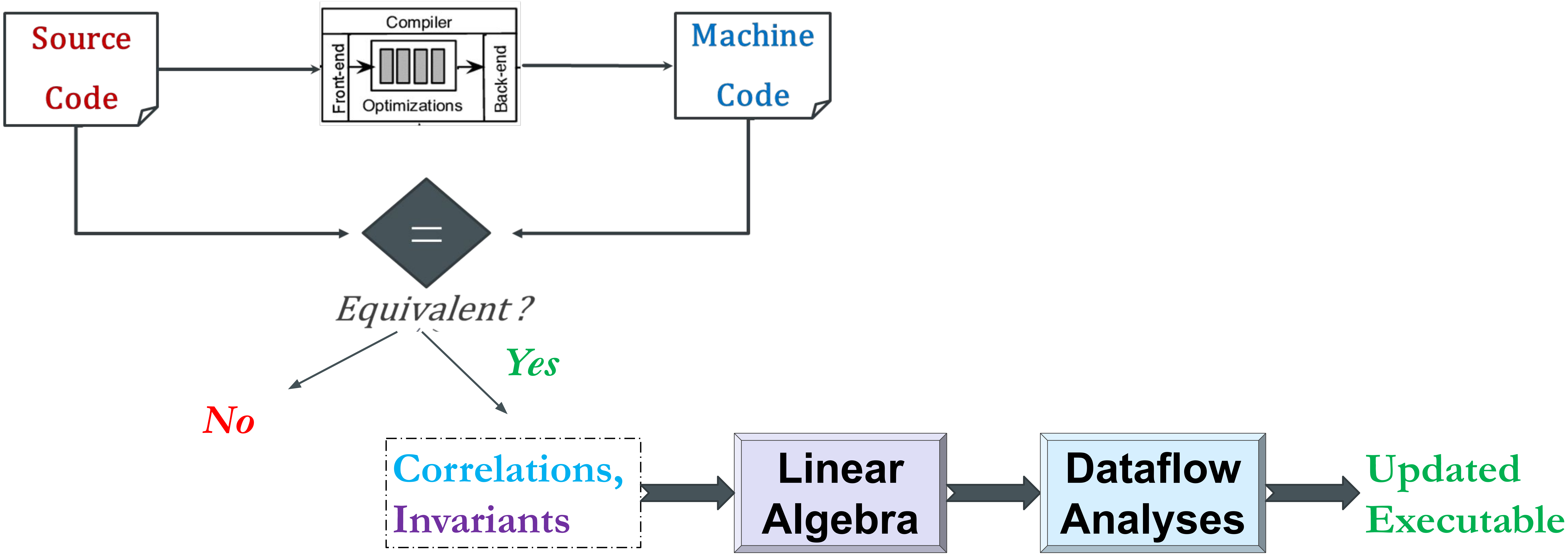
Automatic Generation of Debug Headers



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

CONCLUSIONS

- Compiler Development is Hard but Increasingly Important
- Superoptimization is a Plausible Solution
- Equivalence Checking is an Important Pre-Requisite
- Scalable Algorithms for Equivalence Checking are Possible
  - And have several other applications

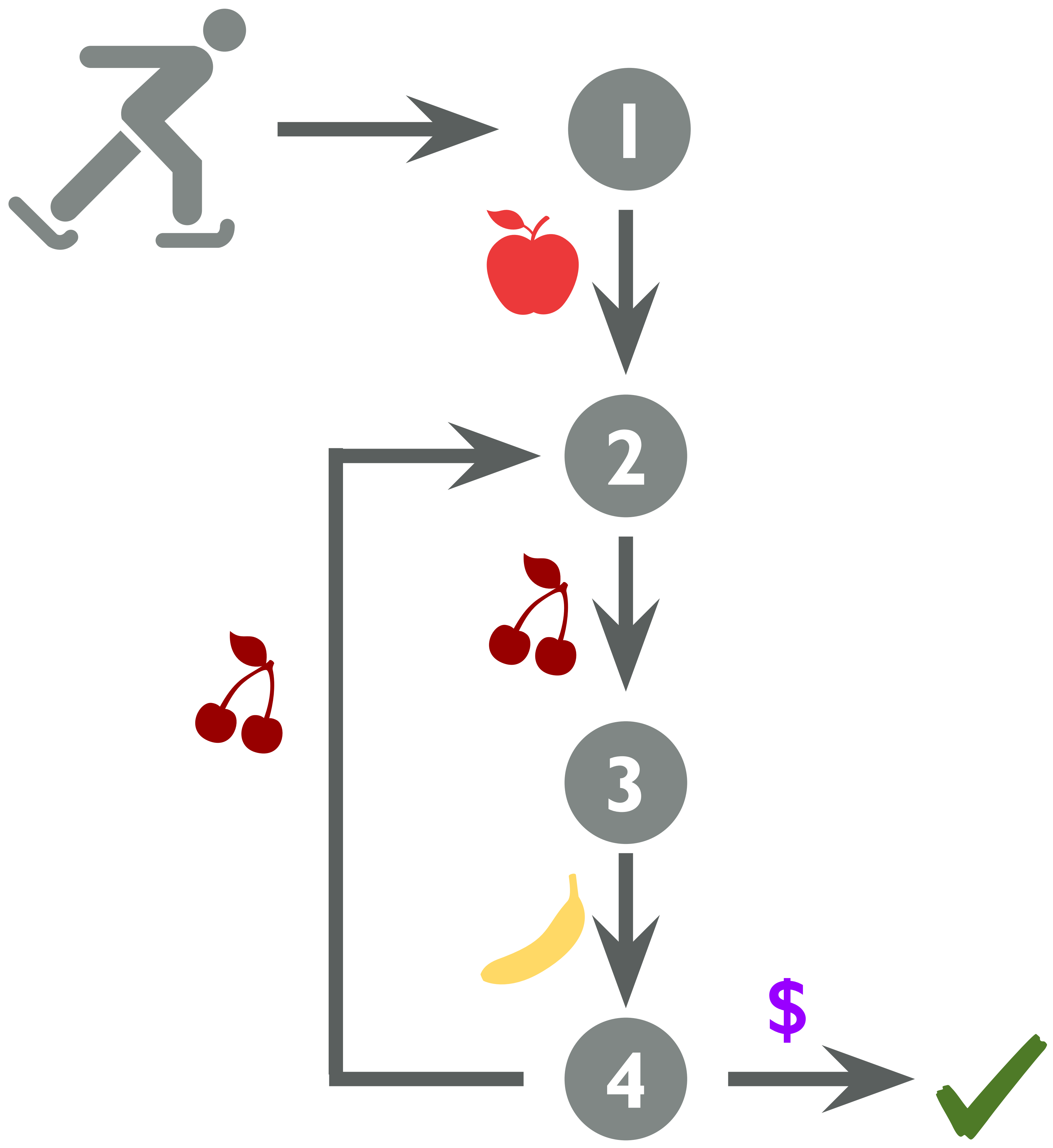
COMPILERAI

COMPILERAI

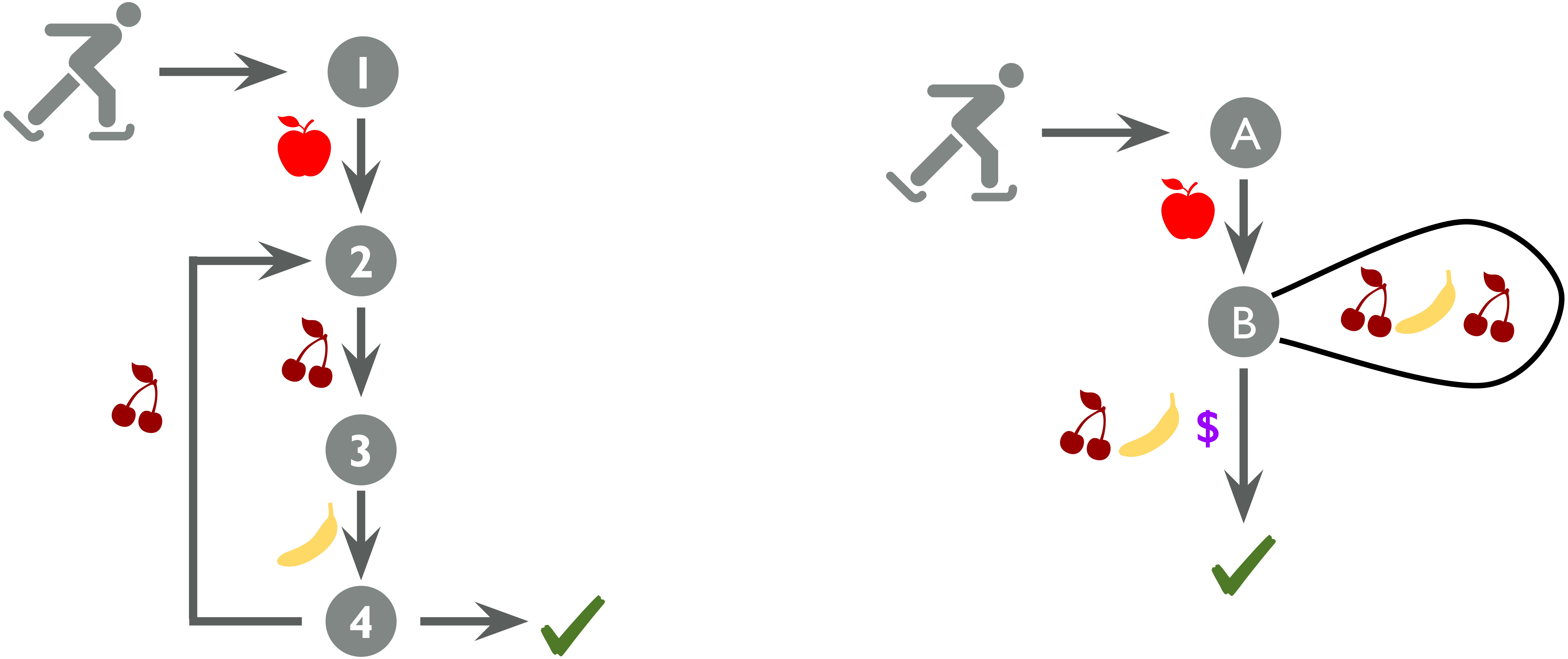
https://compiler.ai

INNOVATION TO INDUSTRY

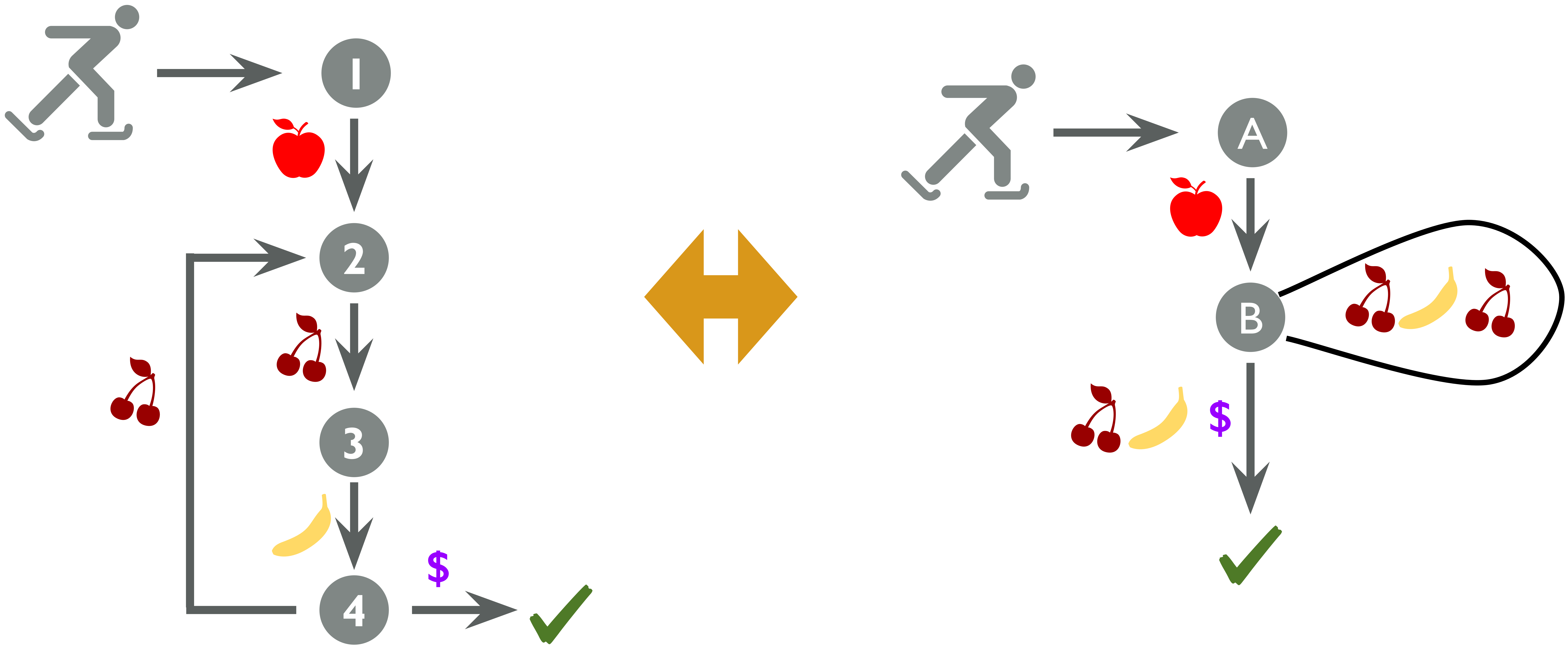
DETERMINISTIC FINITE AUTOMATON



TWO DFAS



EQUIVALENCE



BISIMULATION AS A PRODUCT DFA

IMPERATIVE LANGUAGE SYNTAX

```
if eat() != apple   //Head1
   ERROR
loop forever {   //Body 1
   if eat() != cherry
      ERROR
   if eat() != banana
      ERROR
   next = eat()
   if next == cherry
      CONTINUE
   if next == $
      STOP
}
```

```
if eat() != apple  //Head2
   ERROR
loop forever {  //Body 2
   n1 = eat()
   n2 = eat()
   n3 = eat()
   if       n1 == cherry
     && n2 == banana
     && n3 == cherry
      CONTINUE
   else if  n1 == cherry
        && n2 == banana
        && n3 == $
      STOP
   else ERROR
}
```

```
Head1
Head2
loop forever {
   Body1
   Body2
}
```

IMPERATIVE LANGUAGE SYNTAX

```
if eat() != apple   //Head1
   ERROR
loop forever {   //Body 1
   if eat() != cherry
      ERROR
   if eat() != banana
      ERROR
   next = eat()
   if next == cherry
      CONTINUE
   if next == $
      STOP
}
```

```
if eat() != apple  //Head2
   ERROR
loop forever {  //Body 2
   n1 = eat()
   n2 = eat()
   n3 = eat()
   if       n1 == cherry
      && n2 == banana
      && n3 == cherry
      CONTINUE
   else if  n1 == cherry
        && n2 == banana
        && n3 == $
      STOP
   else ERROR
}
```

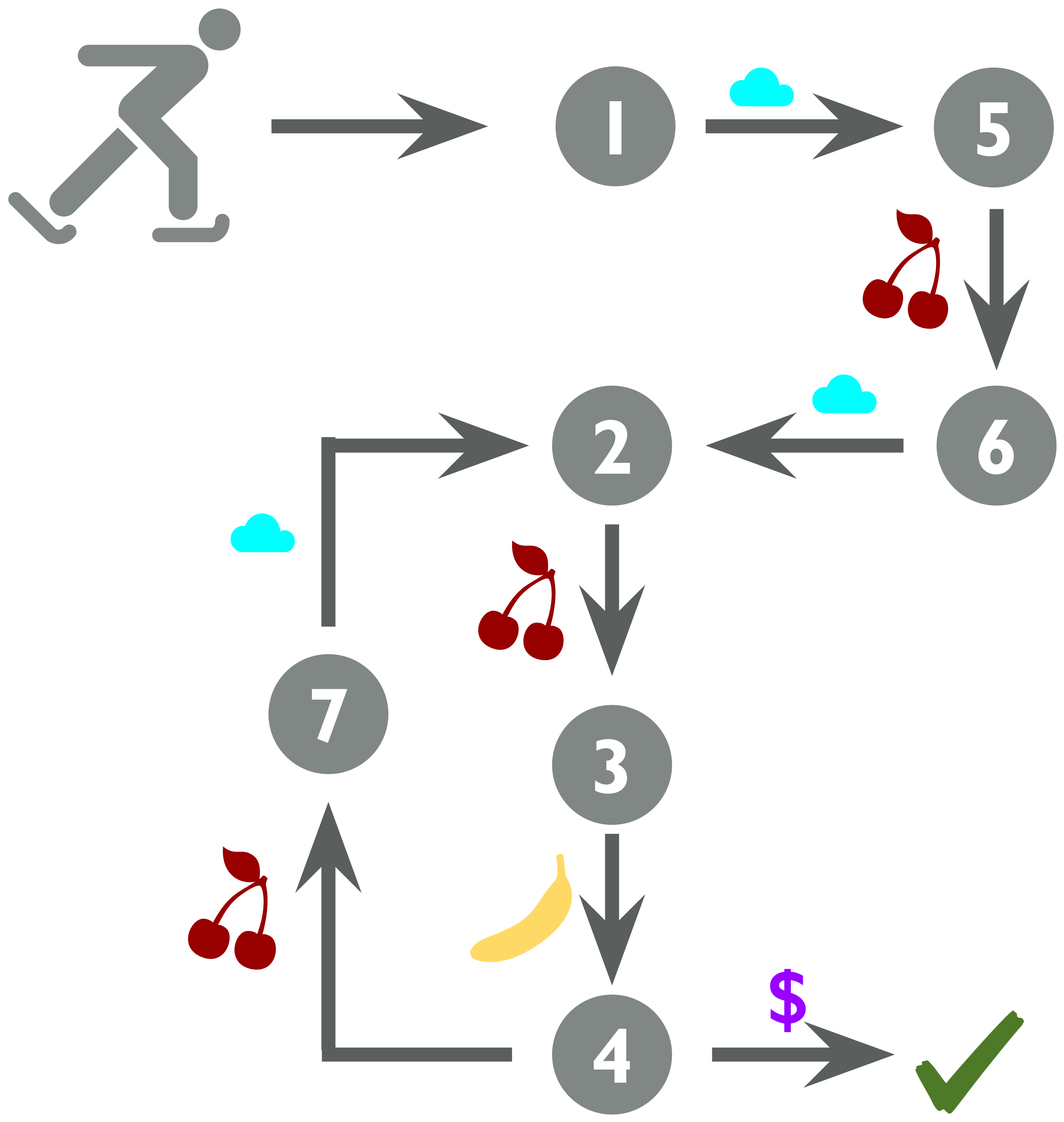
```
Head1
Head2
loop forever {
   Body1
   Body2
}
```

✓

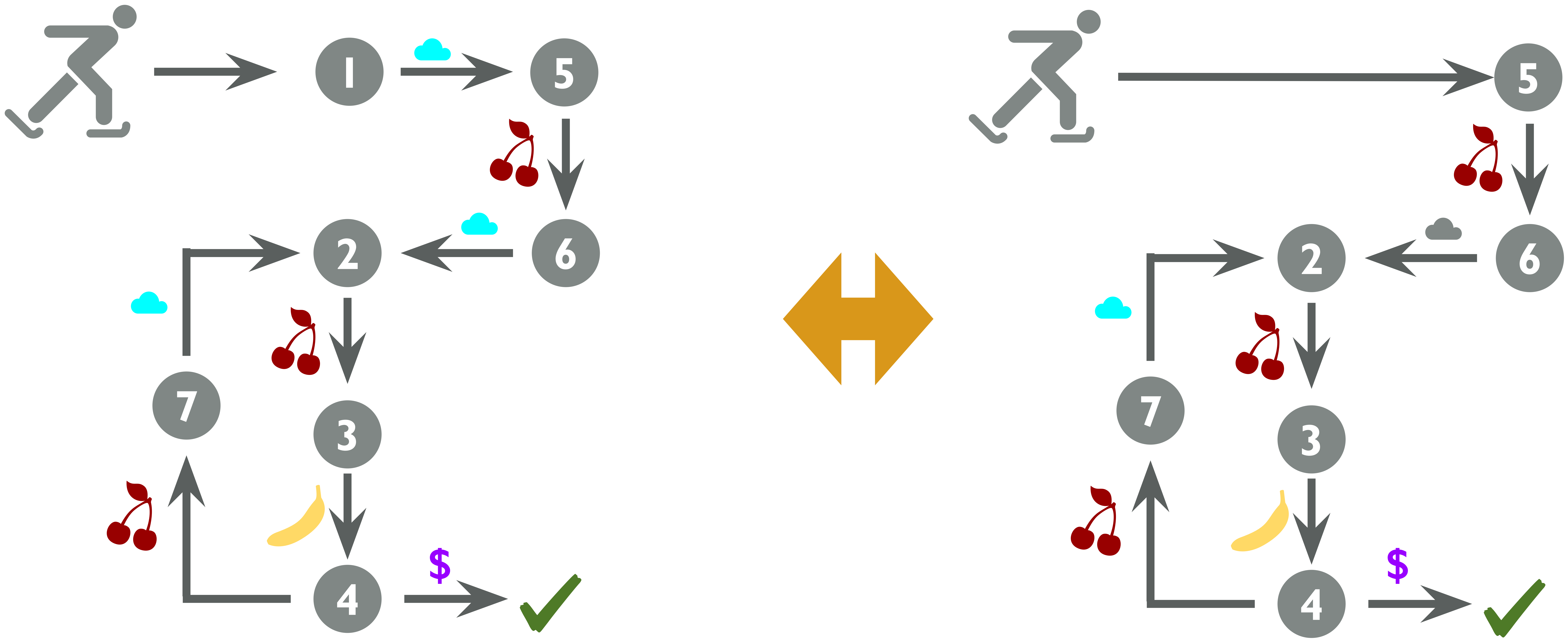
```
Head1
Head2
loop forever {
   Body1
}
loop forever {
   Body2
}
```

✗

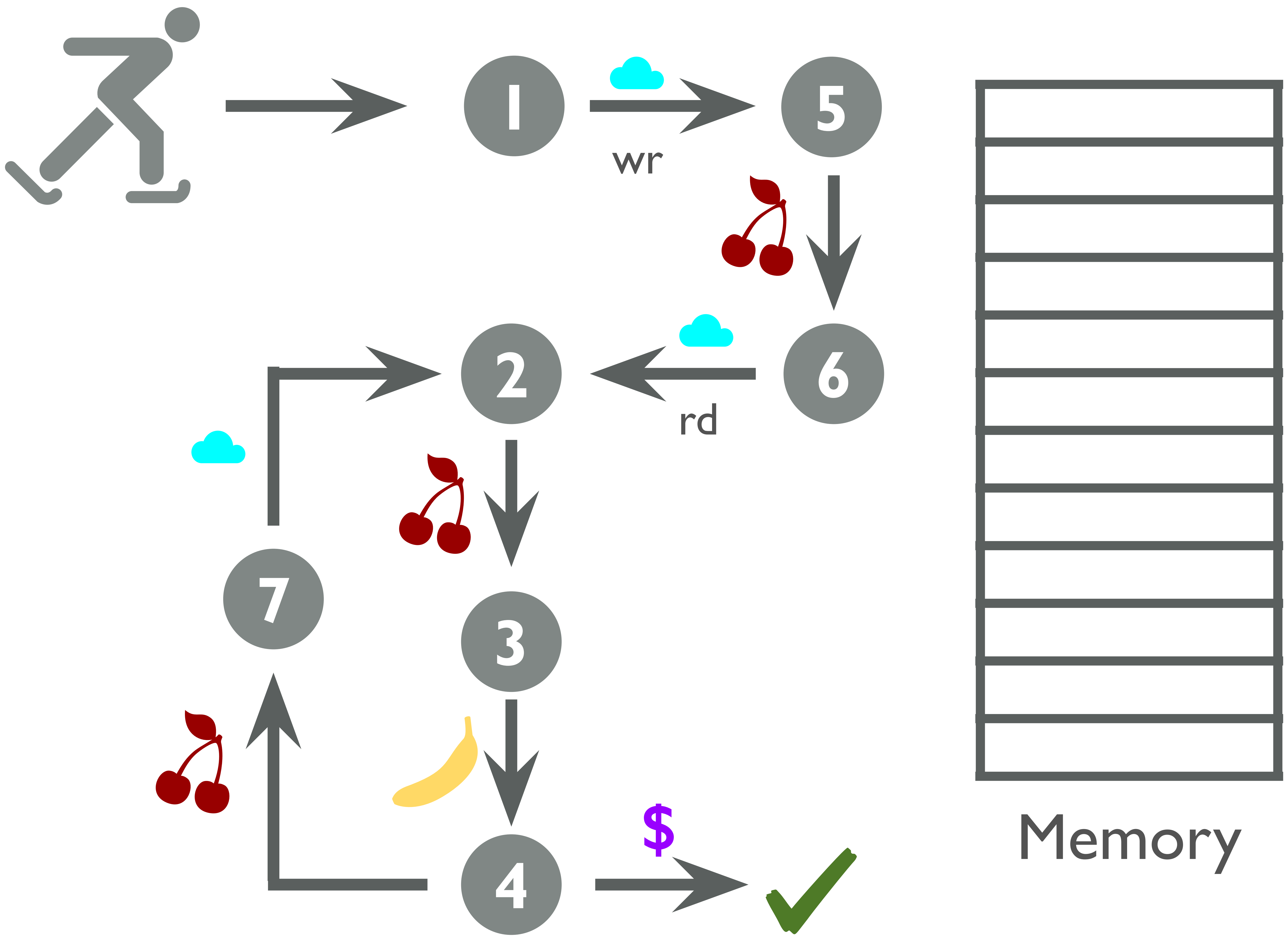
INTERNAL ACTION



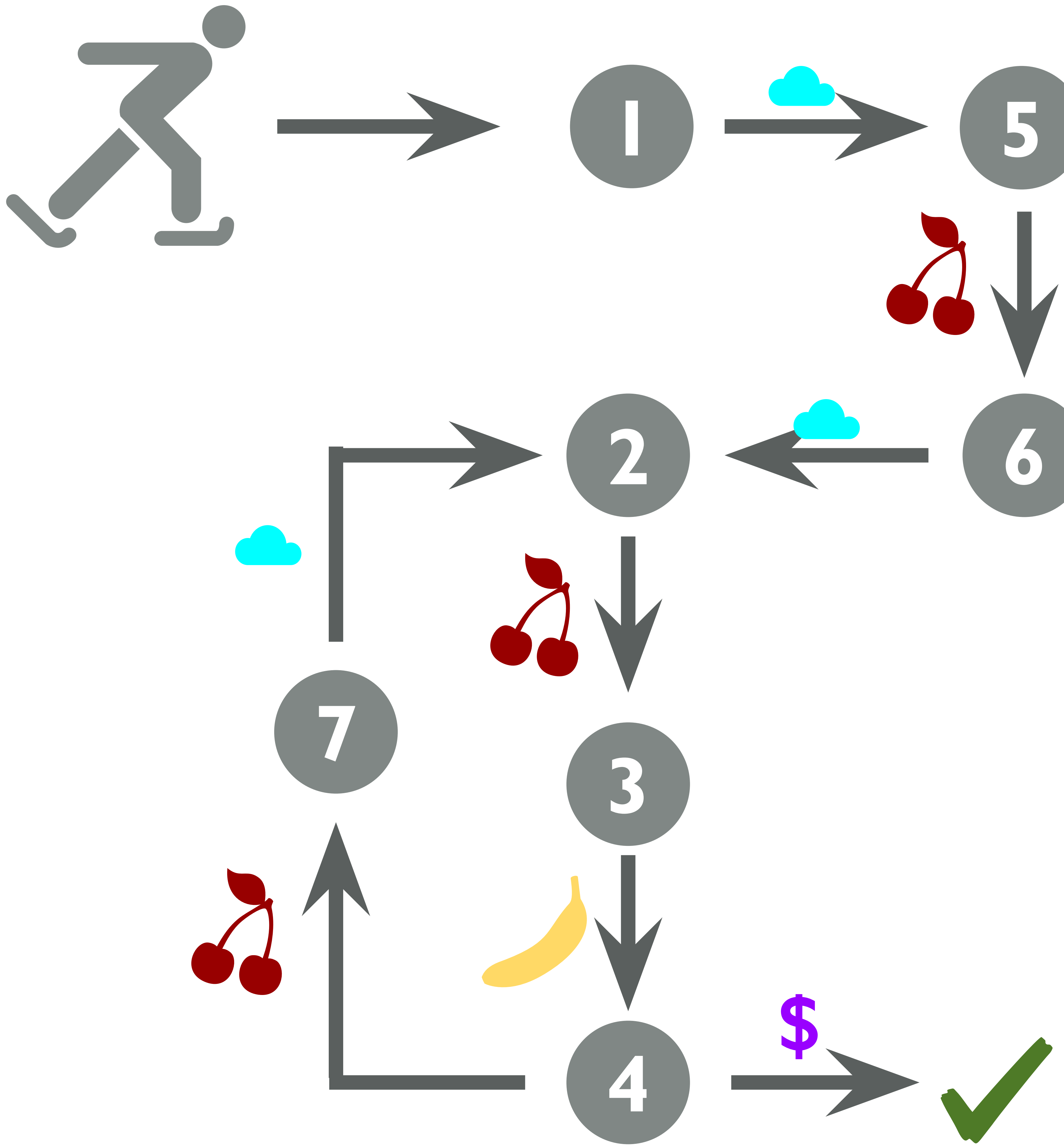
EQUUIVALENCE IN THE PRESECE OF INTERNAL ACTIONS

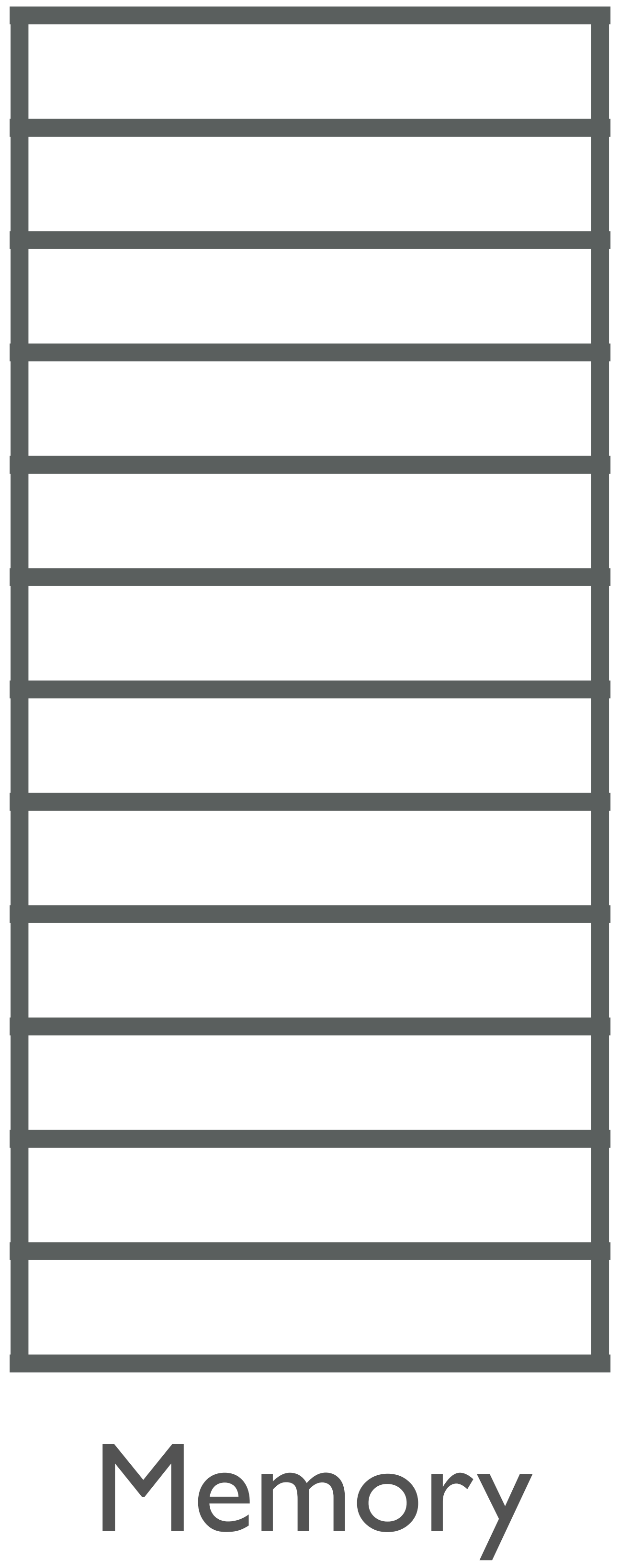


MEMORY

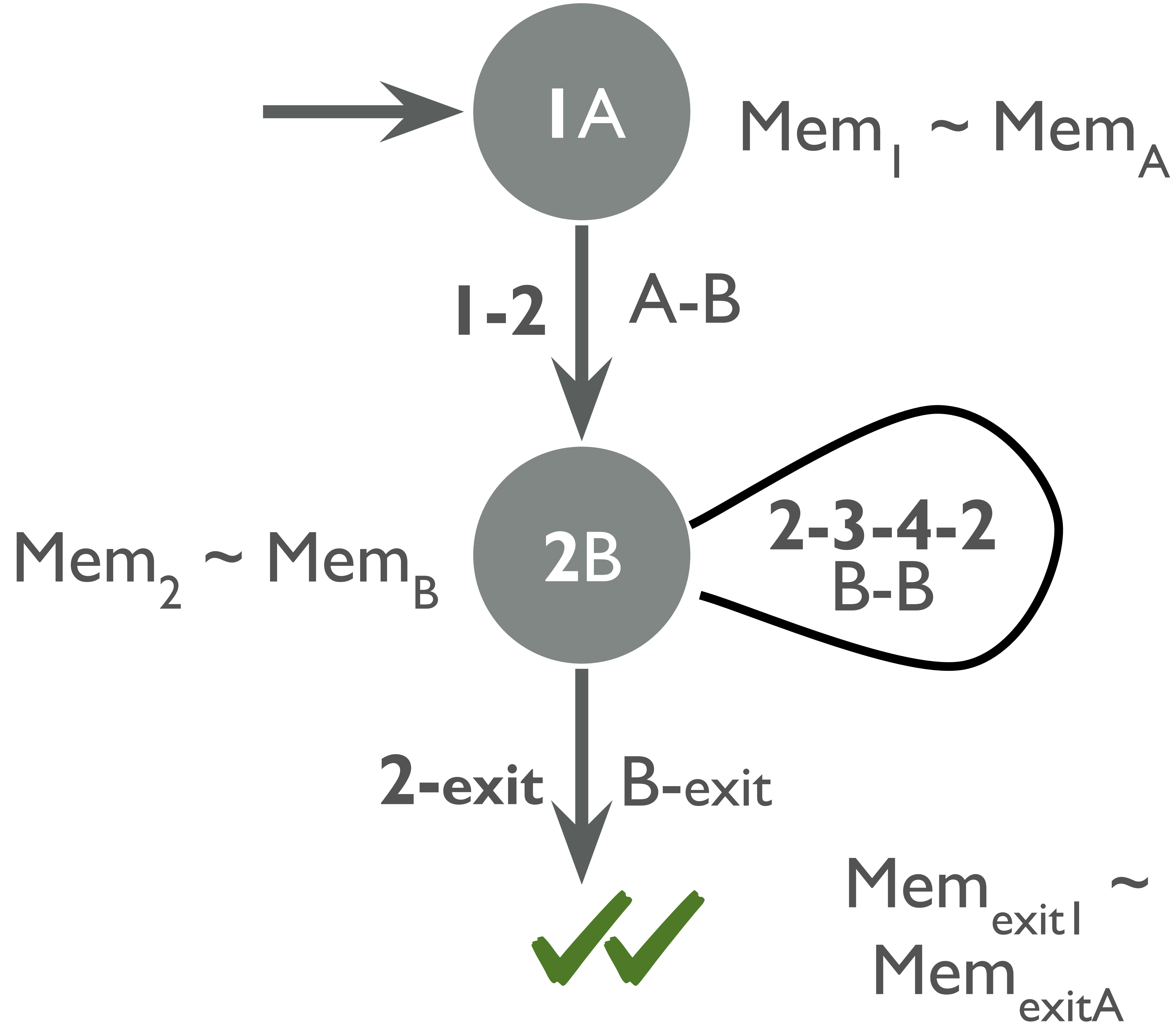


MEMORY WITH LOOPS

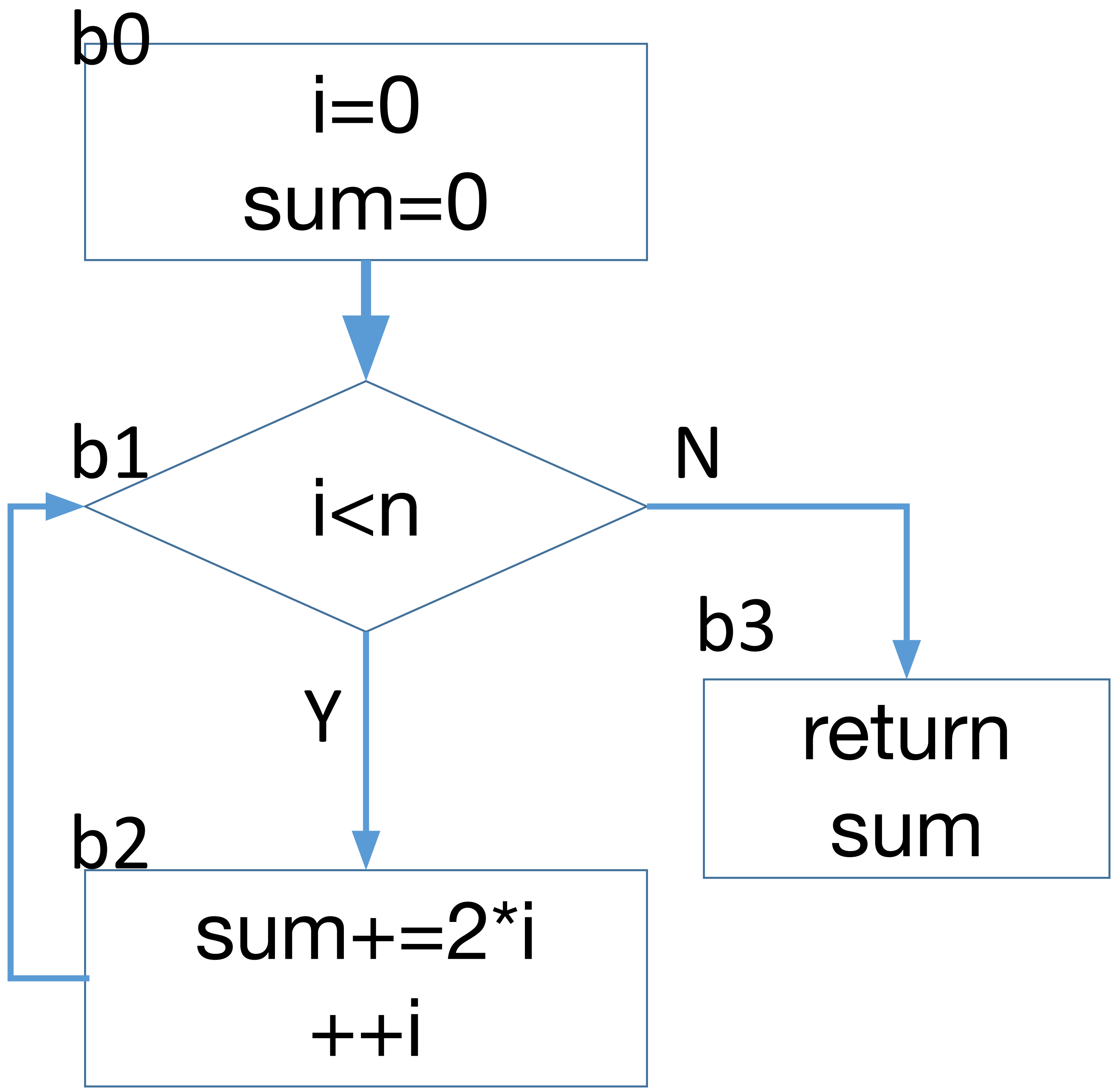




BISIMULATION WITH MEMORY RELATIONS

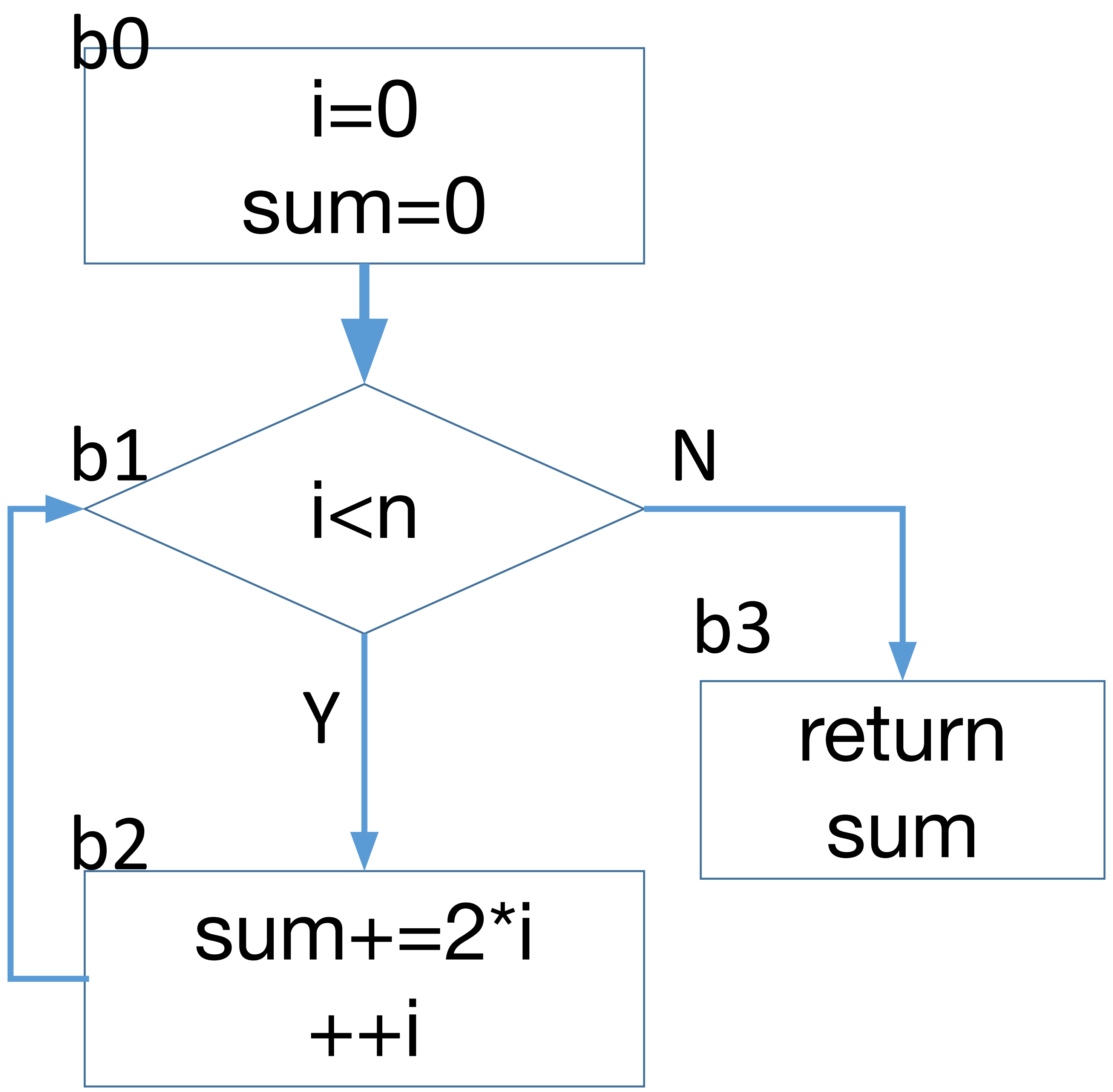


EXAMPLE 1

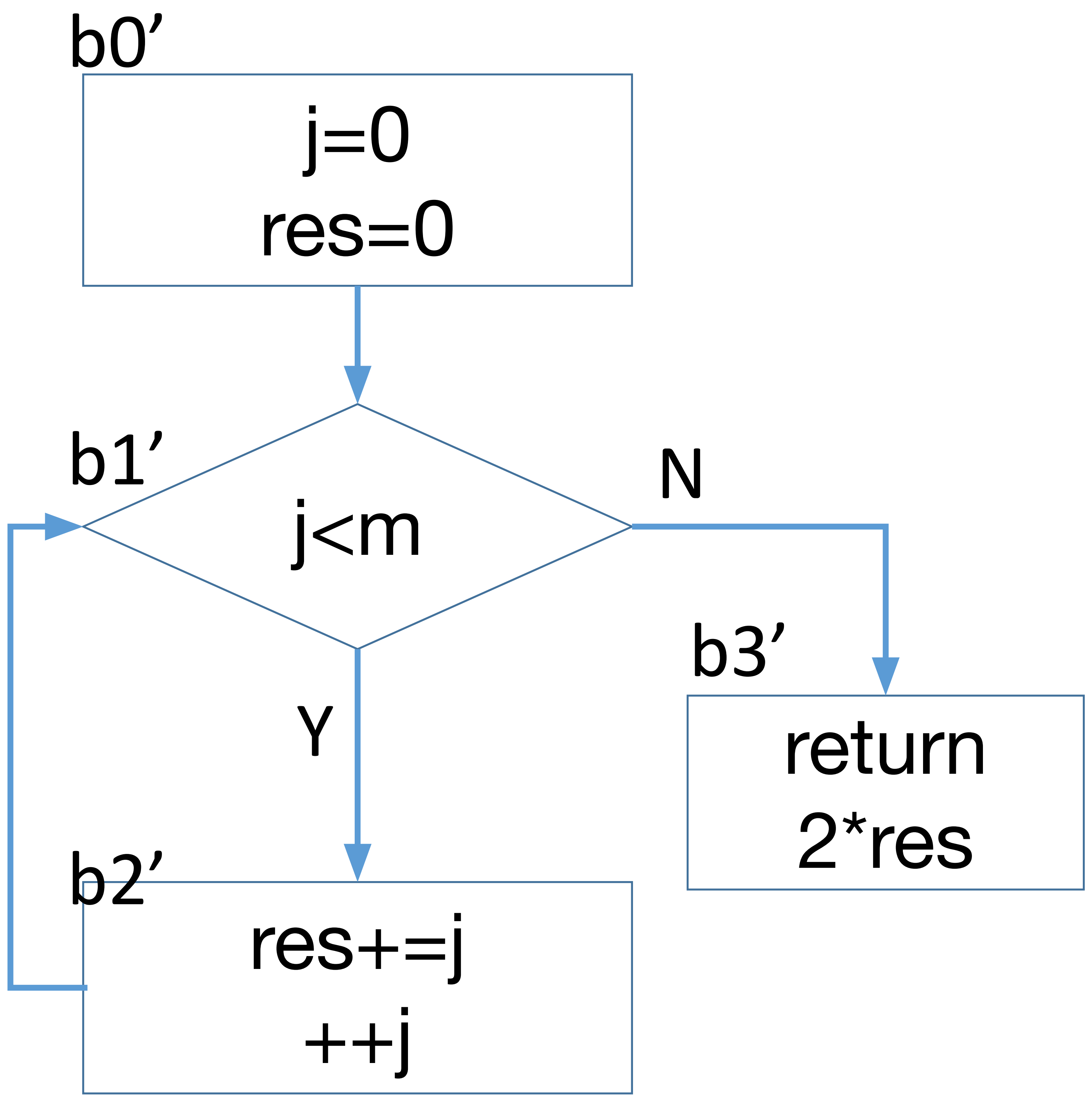


Program 1:
Computes: **Σ (2*i)**
For i ϵ [0, n)

EXAMPLE 1

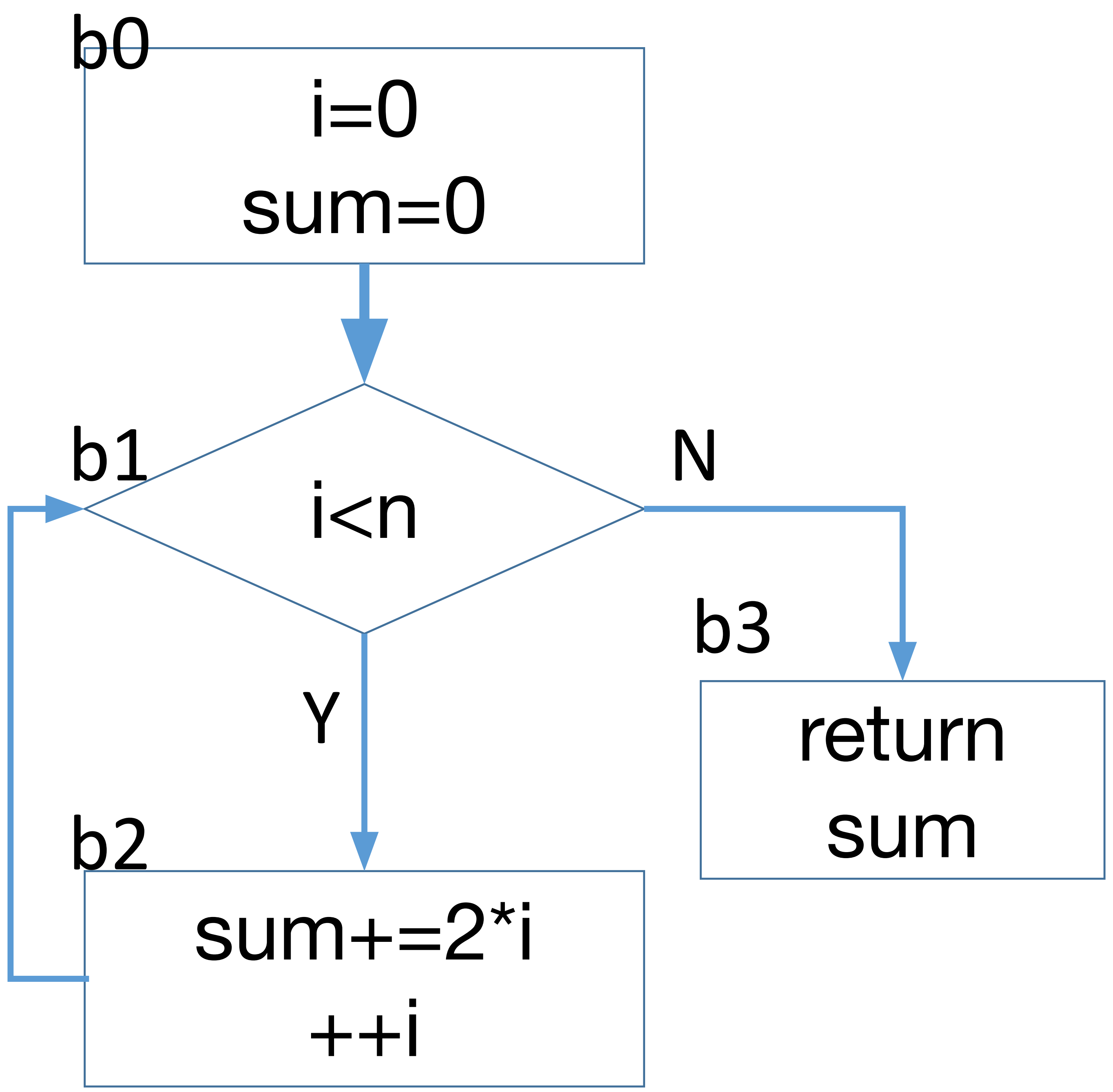


Program 1:
Computes: **Σ (2*i)**
For i ϵ [0, n)

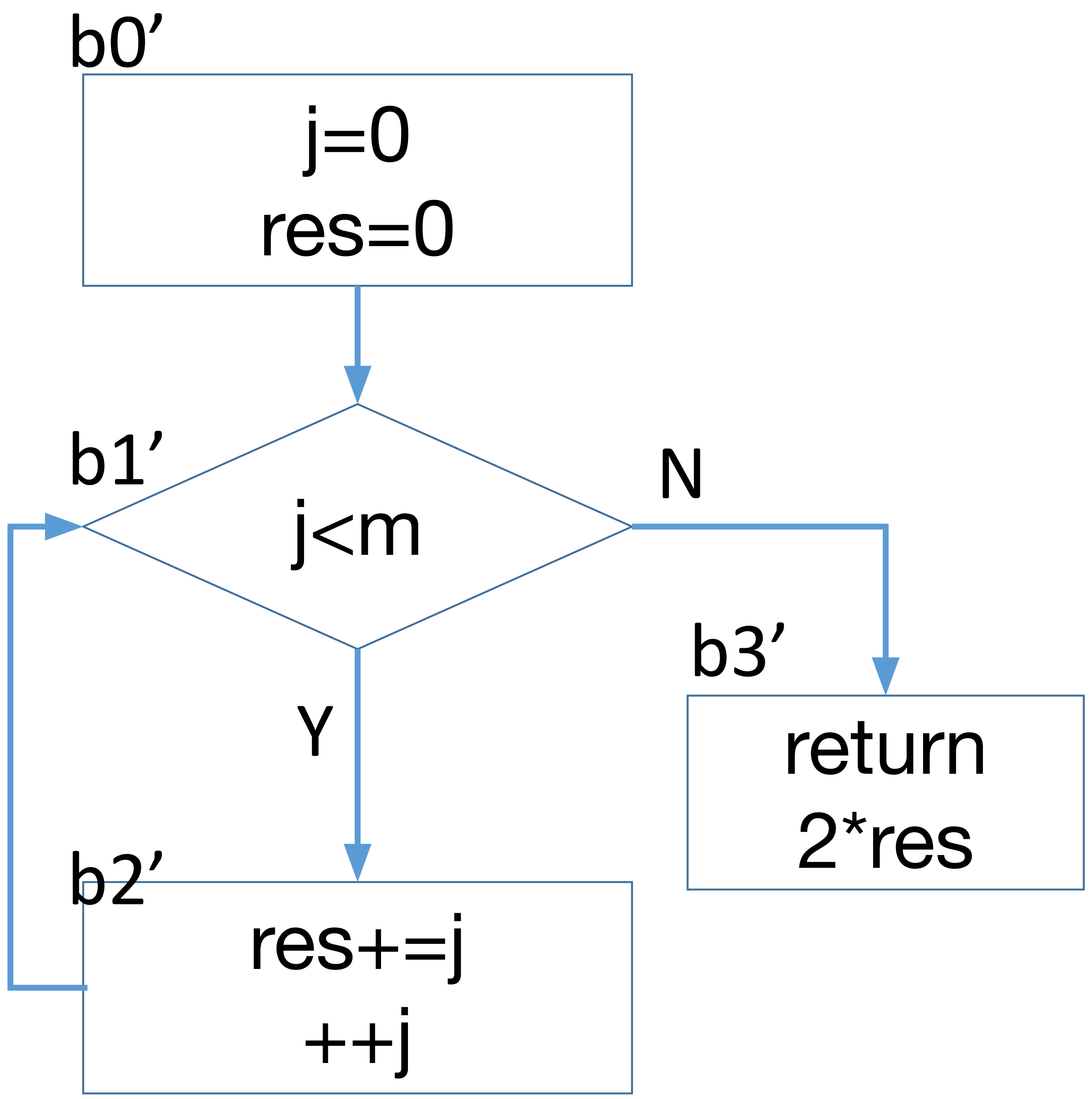


Program 2:
Computes:
2 * Σj
For j ϵ [0, m)

EXAMPLE 1



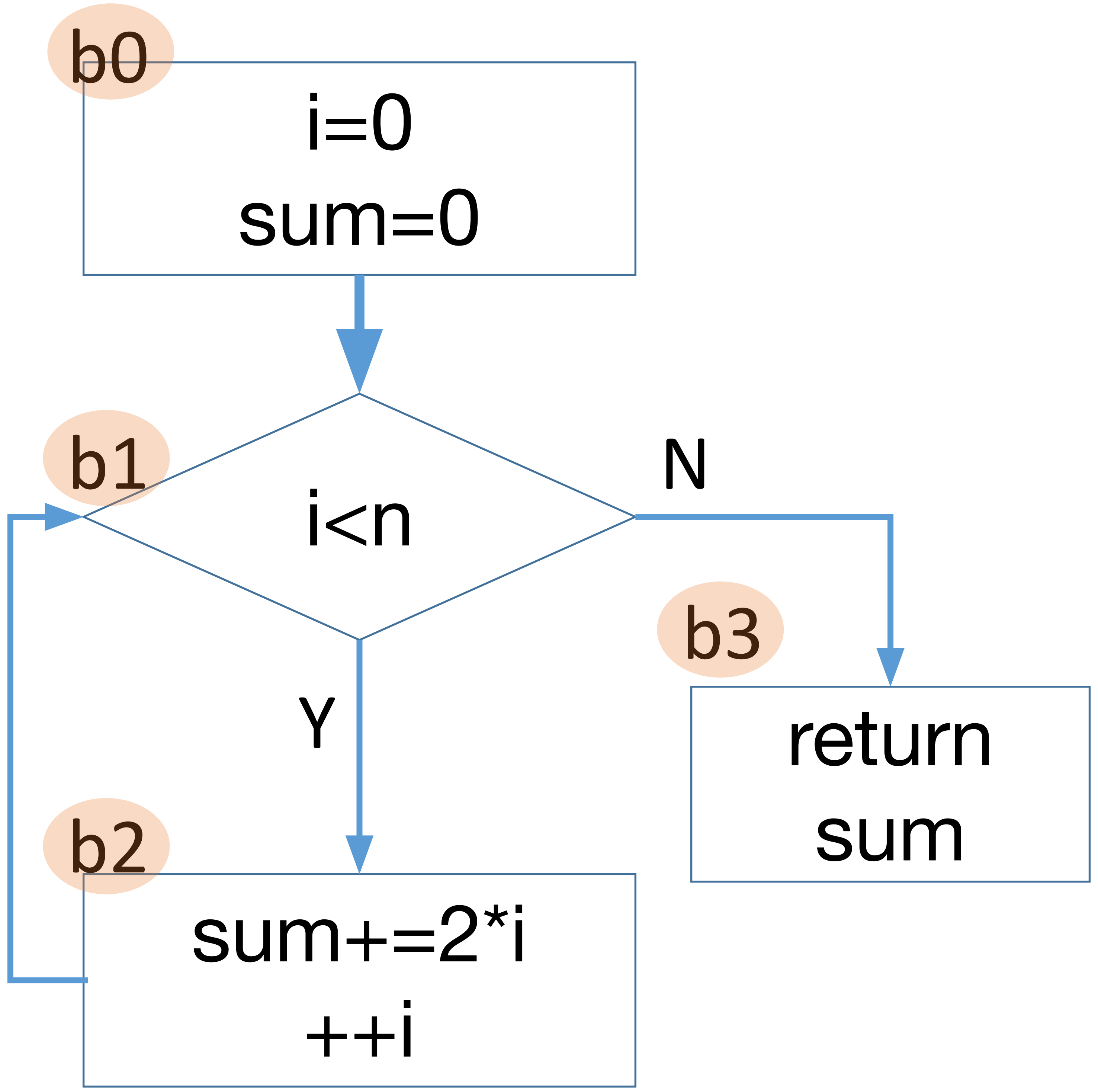
Program 1:
Computes: **Σ (2*i)**
For i ϵ [0, n)



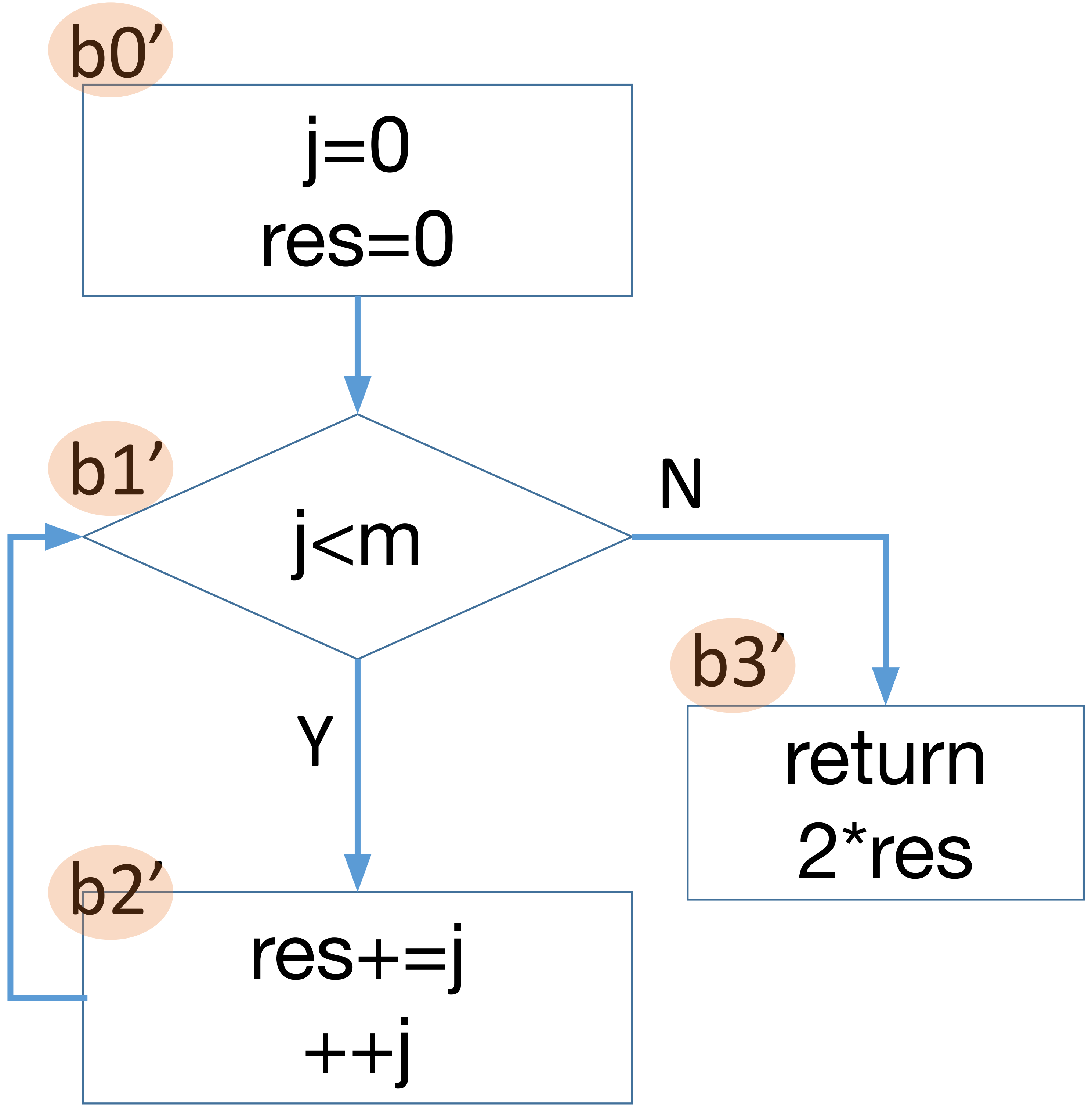
Program 2:
Computes:
2 * Σj
For j ϵ [0, m)

Correlation nodes	Invariants
(b0, b0’)	n = m
(b1, b1’)	i = j, n = m, sum = 2*res
(b3, b3’)	sum = 2*res

EXAMPLE I



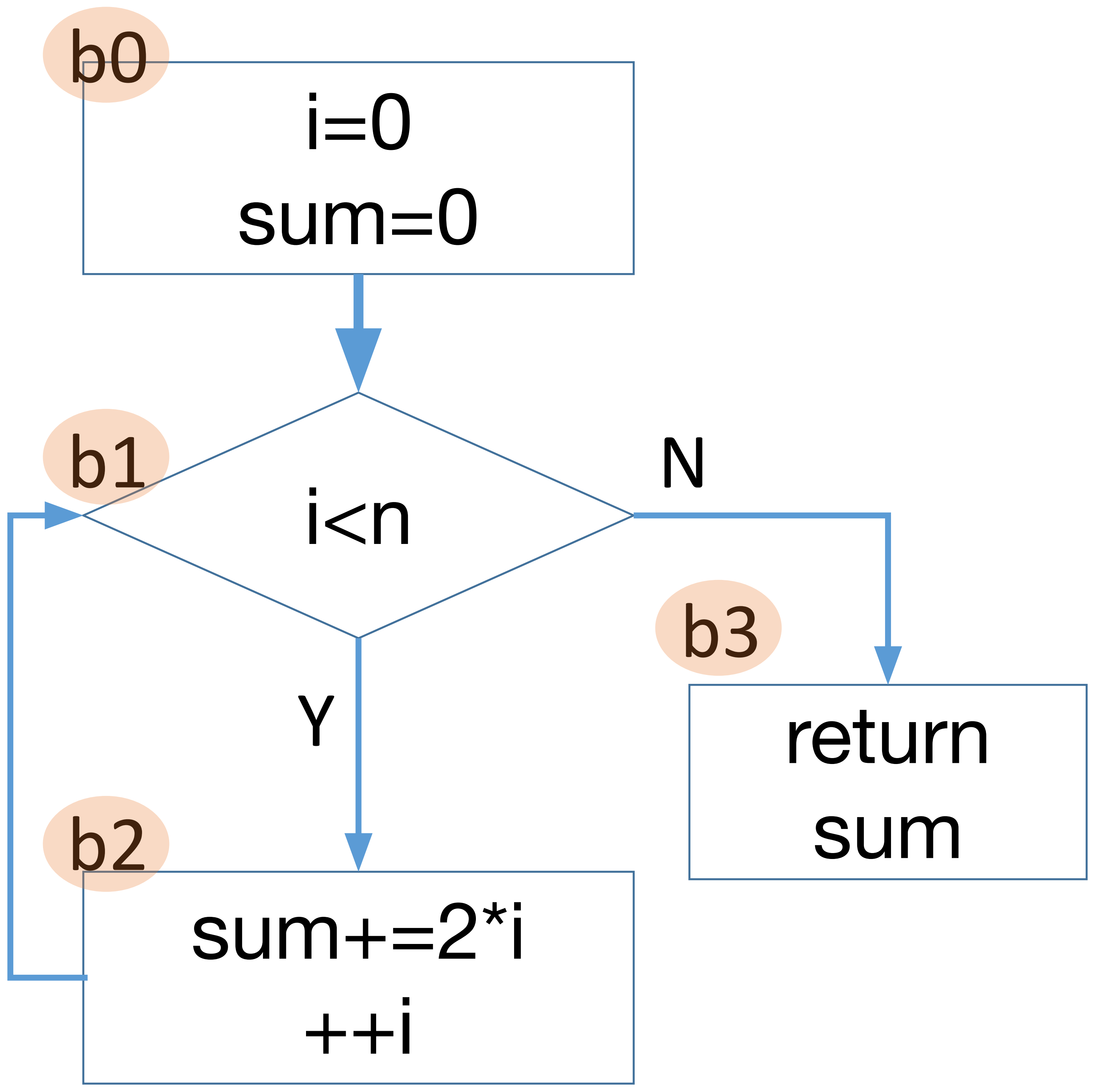
Program 1:
Computes: **Σ (2*i)**
For i ϵ [0, n)



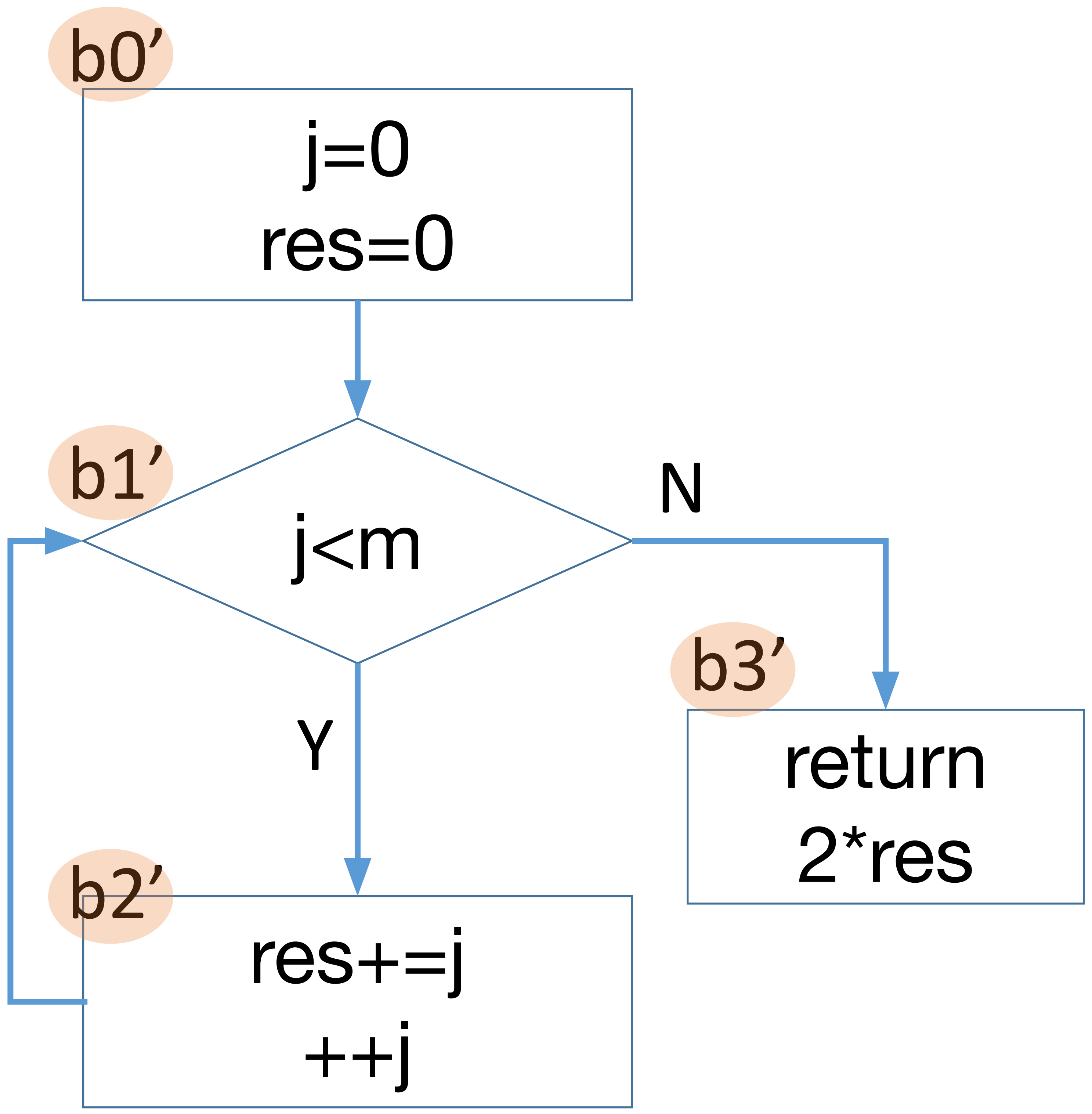
Program 2:
Computes:
2 * Σj
For j ϵ [0, m)

Correlation nodes	Invariants
(b0, b0’)	n = m
(b1, b1’)	i = j, n = m, sum = 2*res
(b3, b3’)	sum = 2*res

EXAMPLE I



Program 1:
Computes: **Σ (2*i)**
For i ϵ [0, n)

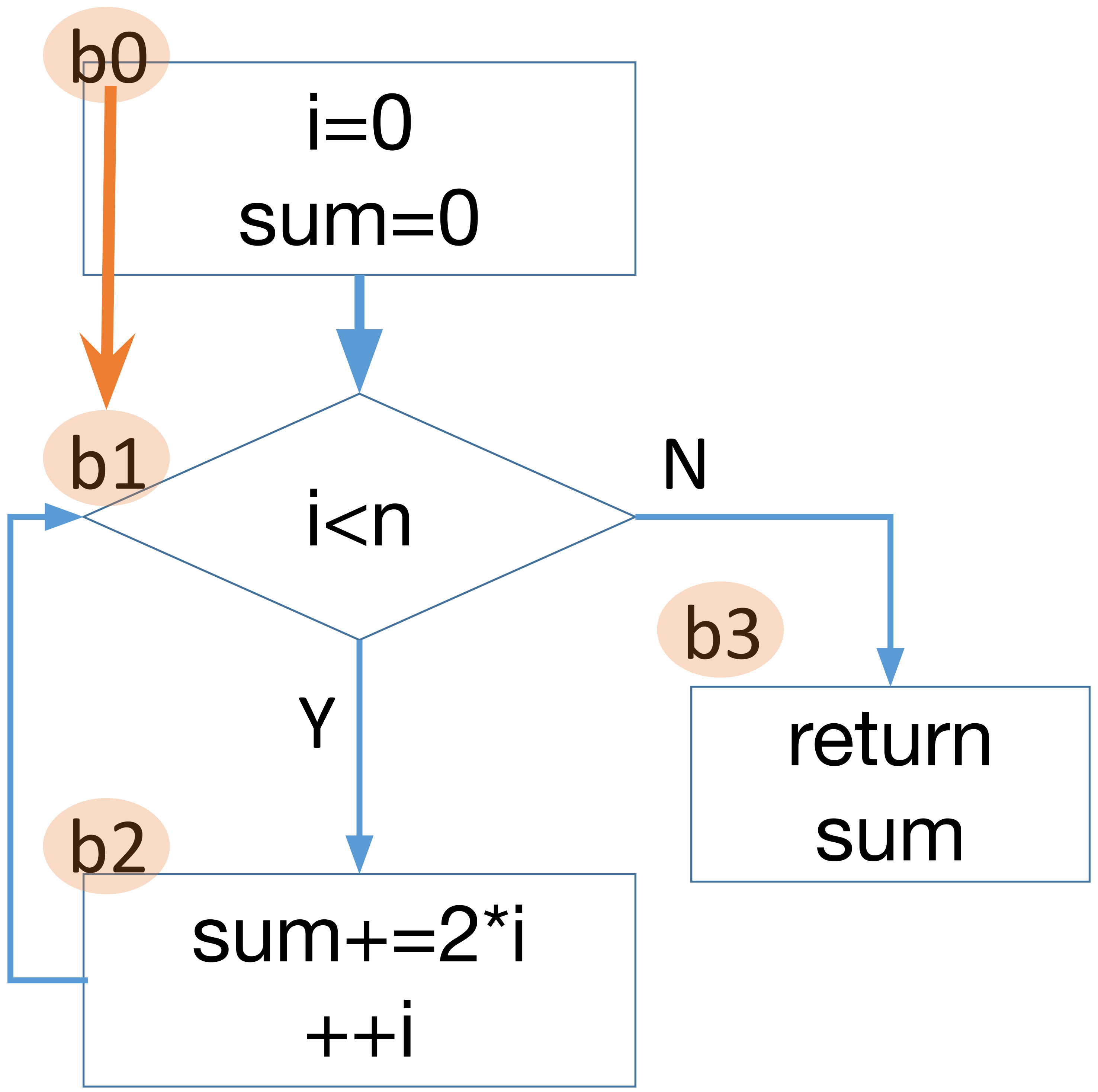


Program 2:
Computes:
2 * Σj
For j ϵ [0, m)

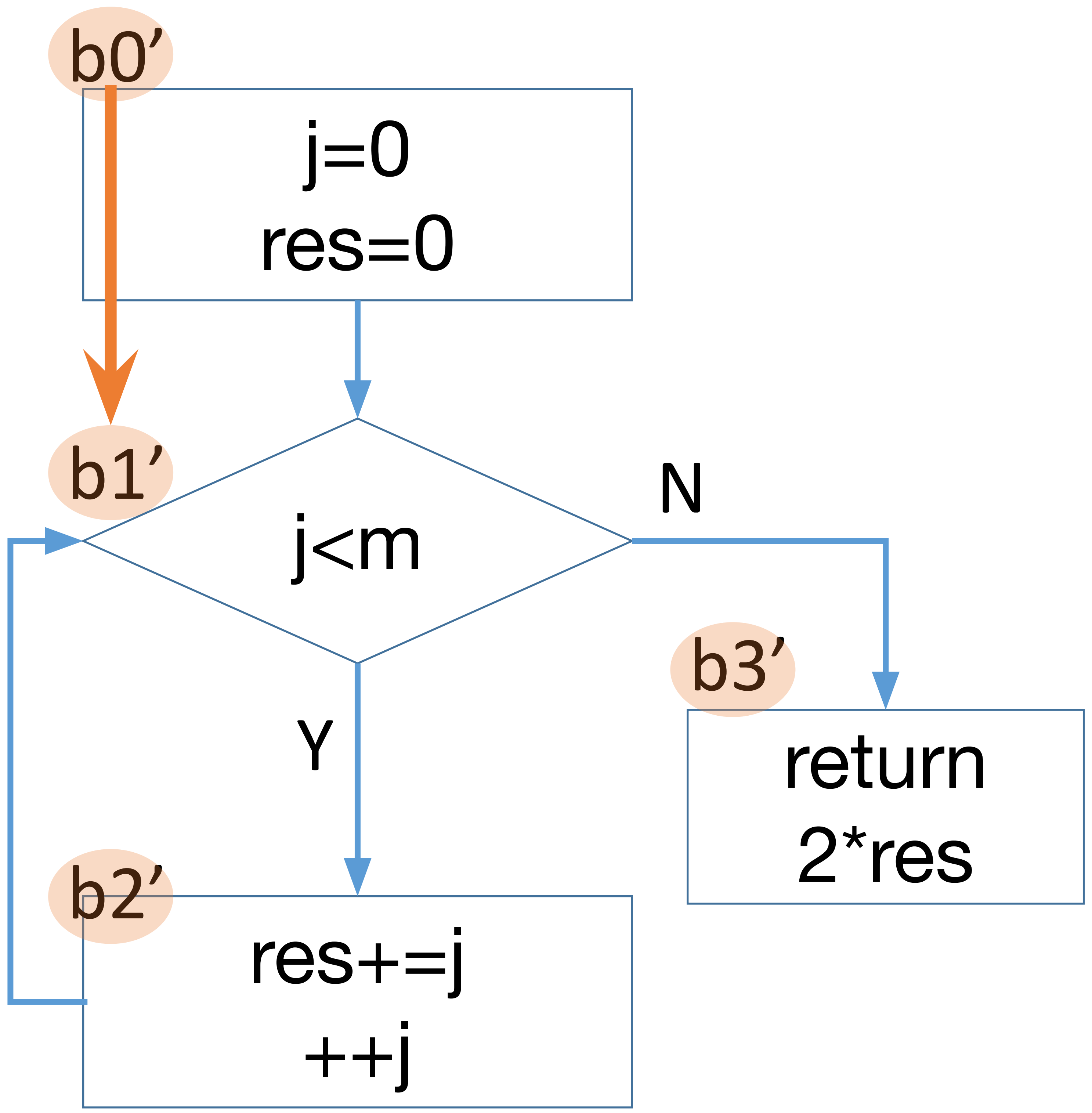
Assumption:
Input equivalence:
n = m

Correlation nodes	Invariants
(b0, b0’)	n = m
(b1, b1’)	i = j, n = m, sum = 2*res
(b3, b3’)	sum = 2*res

EXAMPLE I



Program 1:
Computes: **Σ (2*i)**
For i ϵ [0, n)

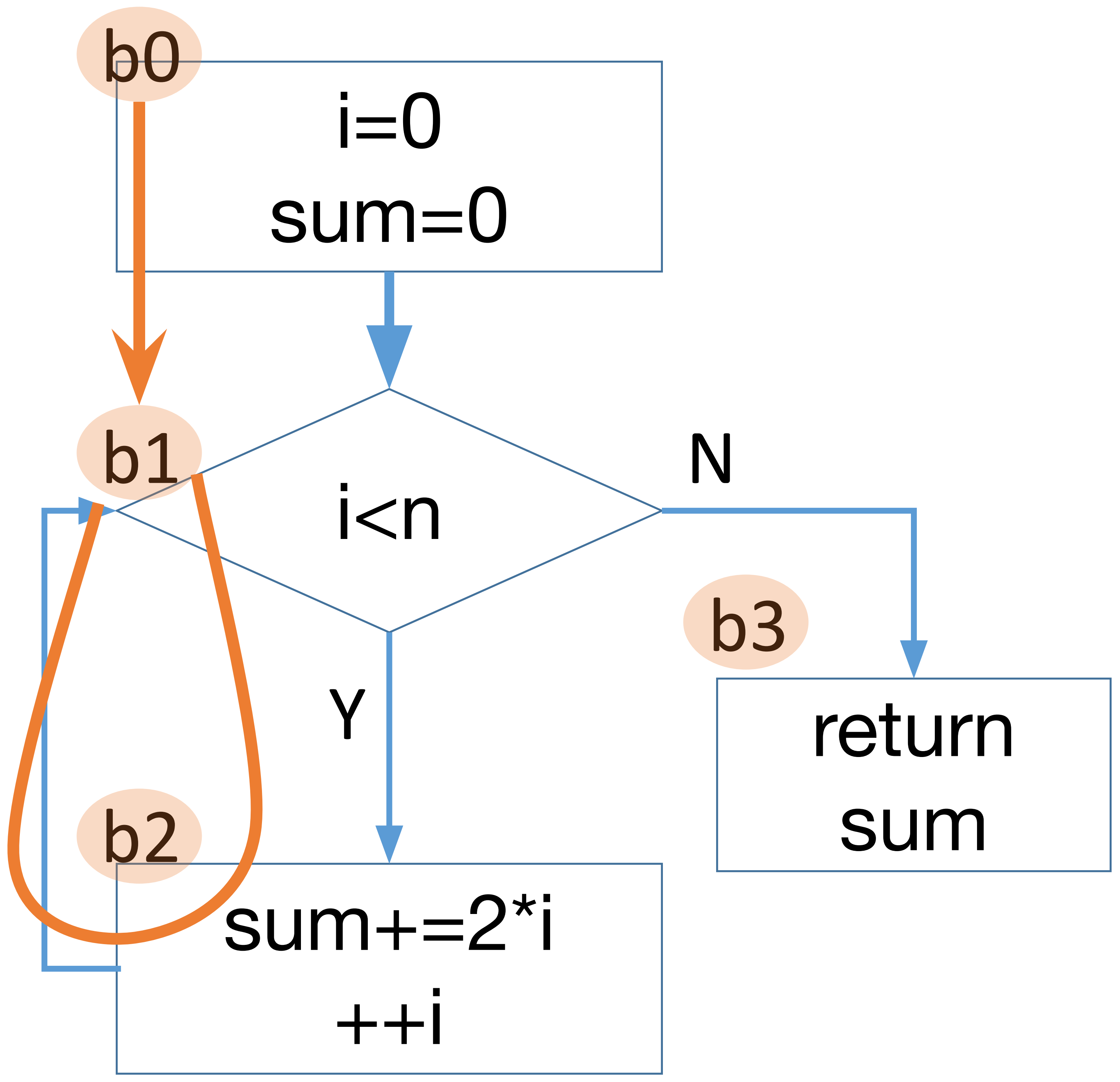


Program 2:
Computes:
2 * Σj
For j ϵ [0, m)

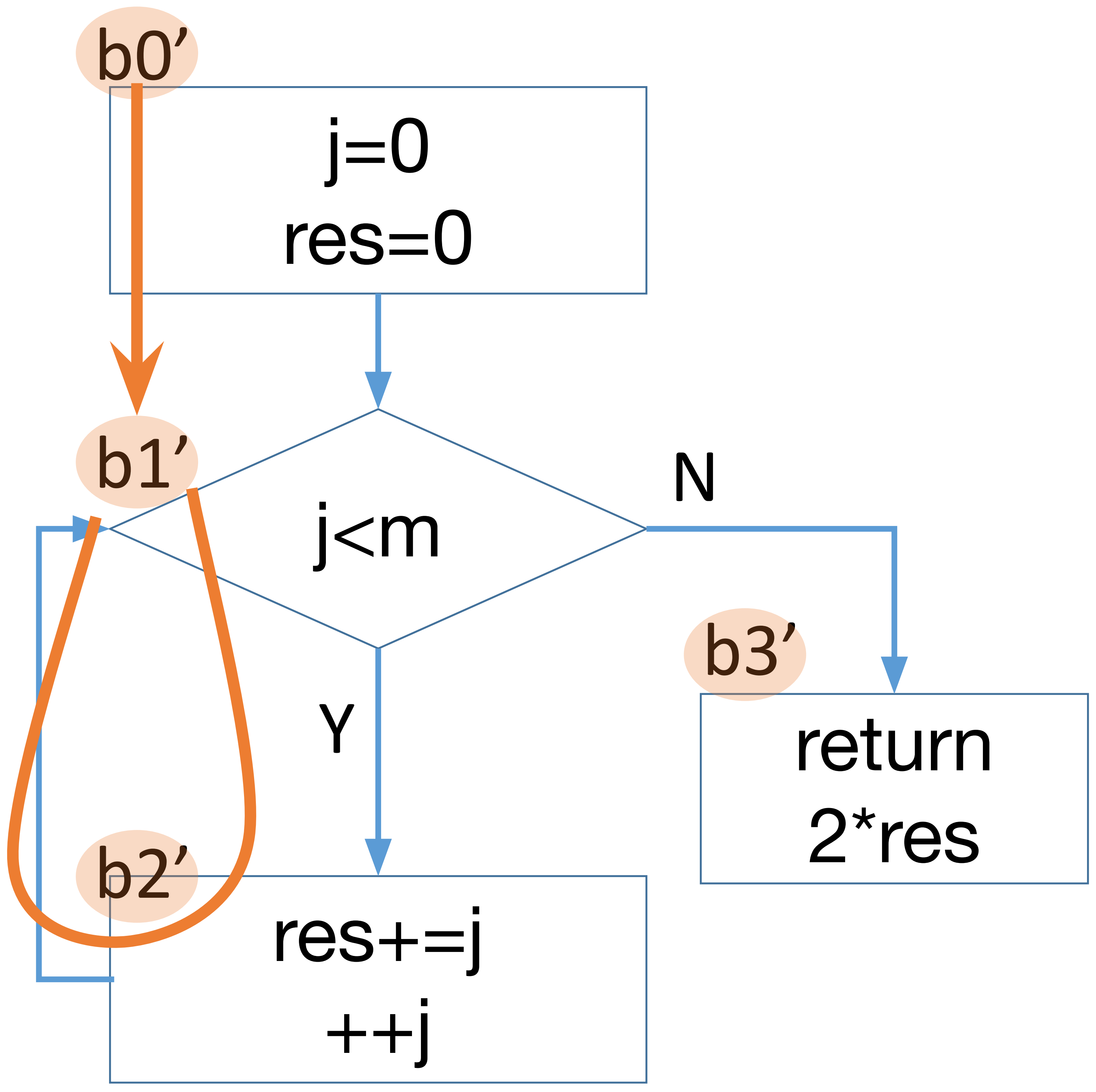
Assumption:
Input equivalence:
n = m

Correlation nodes	Invariants
(b0, b0’)	n = m
(b1, b1’)	i = j, n = m, sum = 2*res
(b3, b3’)	sum = 2*res

EXAMPLE 1



Program 1:
Computes: **Σ (2*i)**
For i ϵ [0, n)

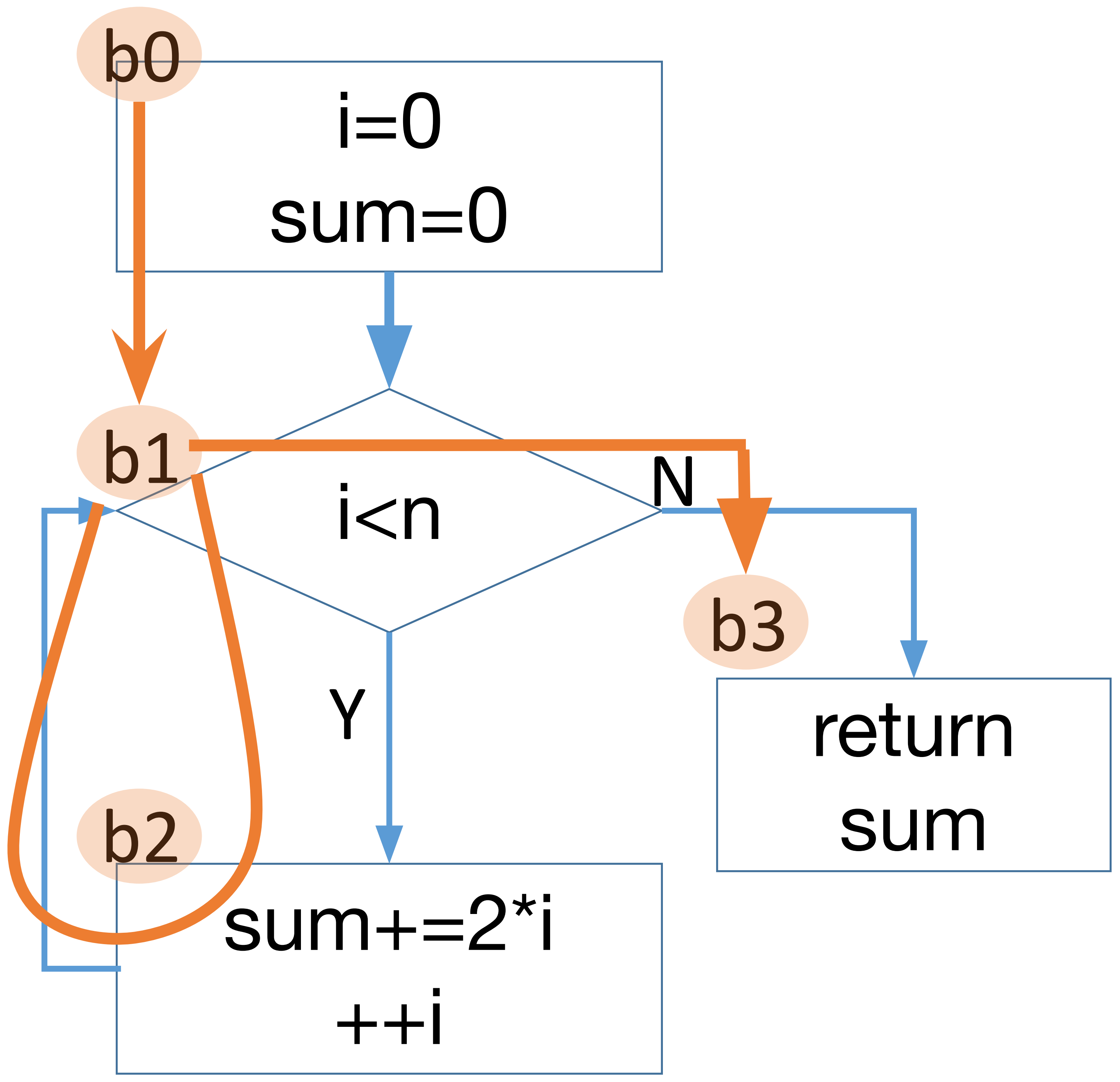


Program 2:
Computes:
2 * Σj
For j ϵ [0, m)

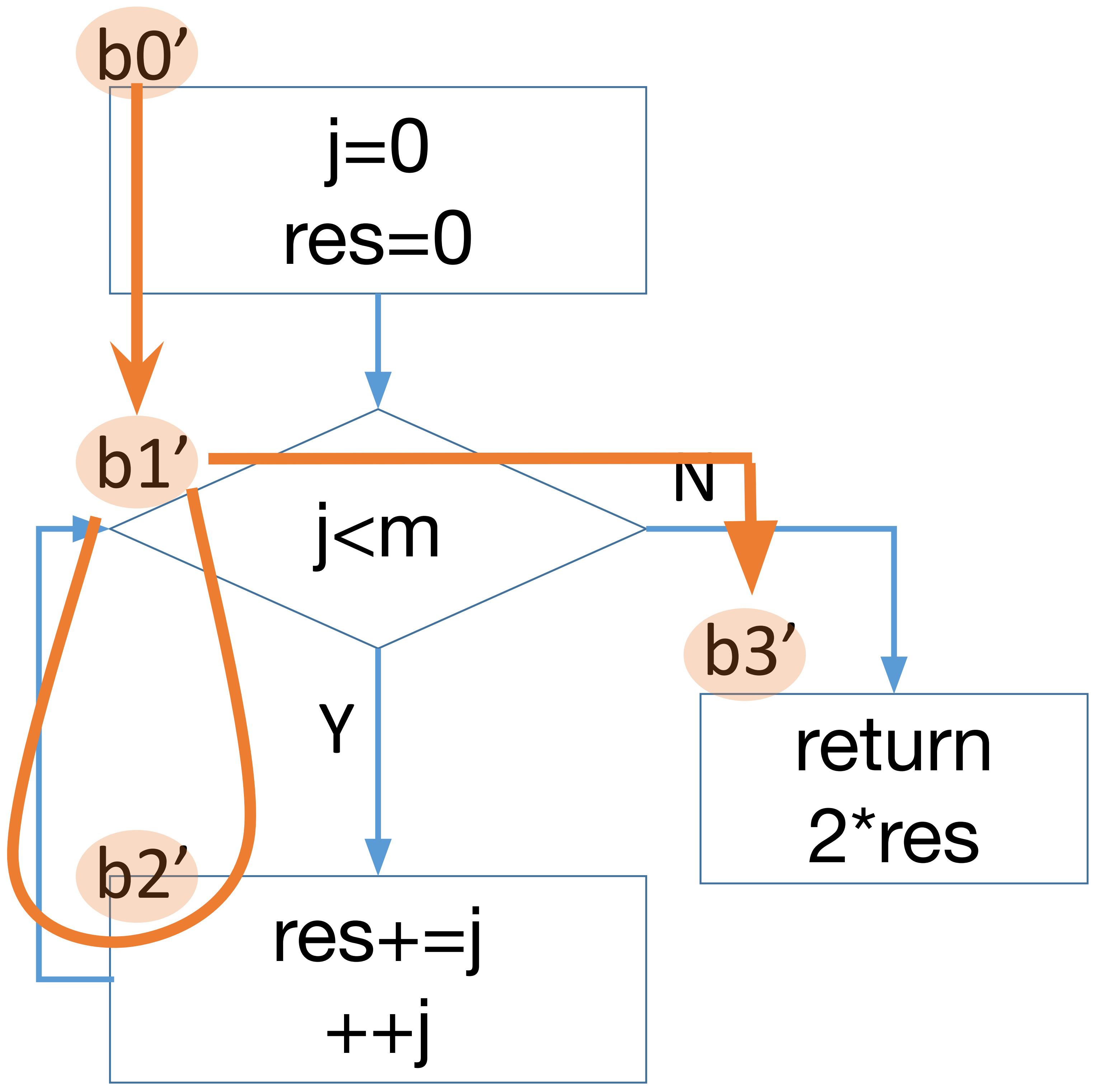
Assumption:
Input equivalence:
n = m

Correlation nodes	Invariants
(b0, b0’)	n = m
(b1, b1’)	i = j, n = m, sum = 2*res
(b3, b3’)	sum = 2*res

EXAMPLE I



Program 1:
Computes: **Σ (2*i)**
For i ϵ [0, n)



Program 2:
Computes:
2 * Σj
For j ϵ [0, m)

Assumption:
Input equivalence:
n = m

Correlation nodes	Invariants
(b0, b0’)	n = m
(b1, b1’)	i = j, n = m, sum = 2*res
(b3, b3’)	sum = 2*res

STEPPING BACK…

An Equivalence Checker is a proof finder

This talk

An Inequivalence Checker is a bug finder

1. Try to Find an Equivalence Proof

2. Try to Find a Distinguishing Input

3. Neither found, give up :-(

EQUIVALENCE CHECKING LITERATURE

Theoretical basis

Simulation Relation, cut points

A. Turing. Checking a large routine. In The early British computer conferences, pages 70–72. MIT Press, Cambridge, MA, USA, 1989 (reproduction)
Milner, R., "Program Simulation: An Extended Formal Notion", Memo 17, Computers and Logic Research Group, University College of Swansea, U.K. (1961).
Milner, R., "A Formal Notion of Simulation Between Programs", Memo 14, Computers and Logic Research Croup, University College of Swansea, U.K. (1970).
Milner, R., "An algebraic definition of simulation between programs", IJCAI'71 Proceedings of the 2nd international joint conference on Artificial intelligence (1971)
Manna, Z., "The Correctness of Programs", J. of Computer and Systems Sciences, Vol. 3, No. 2. 119-127 (1969).
…

Loop free code

D. W. Currie, A. J. Hu, and S. P. Rajan. Automatic formal veri?cation of DSP software. In DAC, pages 130–135, 2000.
X. Feng and A. J. Hu. Automatic formal veri?cation for scheduled VLIW code. In LCTES-SCOPES, pages 85–92, 2002.
X. Feng and A. J. Hu. Cutpoints for formal equivalence veri?cation of embedded software. In EMSOFT, pages 307– 316, 2005.
T. Matsumoto, H. Saito, and M. Fujita. Equivalence checking of C programs by locally performing symbolic simulation on dependence graphs. In ISQED, pages 370–375, 2006.
T. Arons, E. Elster, L. Fix, S. Mador-Haim, M. Mishaeli, J. Shalev, E. Singerman, A. Tiemeyer, M. Y. Vardi, and L. D. Zuck. Formal veri?cation of backward compatibility of microcode. In CAV, pages 185–198, 2005.
Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Provably correct peephole optimizations with alive. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 22–32. PLDI 2015, ACM (2015)
…

Application specific/limitations

Partial equivalence (with bounded unrolling)

D.Jacksonand, D.A.Ladd. SemanticDiff:A tool for summarizing the effects of modi?cations. In ICSM, pages 243–252, 1994.
Lahiri, S., Hawblitzel, C., Kawaguchi, M., Rebelo, H.: Symdiff: A language-agnostic semantic diff tool for imperative programs. In: CAV ’12. Springer (2012)
Lahiri, S., Sinha, R., Hawblitzel, C.: Automatic rootcausing for program equivalence failures in binaries. In: Computer Aided Veri?cation (CAV’15). Springer (2015)
S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In SIGSOFT FSE, pages 226–237, 2008
D. A. Ramos and D. R. Engler. Practical, low-effort equivalence veri?cation of real code. In CAV, pages 669–685, 2011.
Lopes, N.P., Monteiro, J.: Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. Int. J. Softw. Tools Technol. Transf. 18(4), 359–374 (Aug 2016)
….

Regression verification

Hawblitzel, C., Lahiri, S.K., Pawar, K., Hashmi, H., Gokbulut, S., Fernando, L., Detlefs, D., Wadsworth, S.: Will you still compile me tomorrow? static cross-version compiler validation. In: ESEC/FSE 2013, ACM (2013)
Strichman, O., Godlin, B.: Regression verification - a practical way to verify programs. In: Verified Software: Theories, Tools, Experiments, vol. 4171, pp. 496–501. Springer Berlin Heidelberg (2008)
Felsing, D., Grebing, S., Klebanov, V., Ru¨mmer, P., Ulbrich, M.: Automating regression veri?cation. In: ASE ’14, ACM (2014)
…

Affine programs

Verdoolaege, S., Janssens, G., and Bruynooghe, M. 2009. Equivalence checking of static affine programs using widening to handle recurrences. In Computer Aided Verification 21. Springer, 599–613.
Verdoolaege, S., Janssens, G., and Bruynooghe, Equivalence Checking of Static Affine Programs using Widening to Handle Recurrences (TOPLAS12)
…

Data Driven (test cases are given)

Churchill, B., Sharma, R., Bastien, J., Aiken, A.: Sound loop superoptimization for google native client. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 313–326. ASPLOS ’17, ACM (2017)
Sharma, R., Schkufza, E., Churchill, B., Aiken, A.: Data-driven equivalence checking. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications. pp. 391–406. OOPSLA ’13, ACM (2013)
…

Translation validation (Pass based, knowledge of trasformations)

V.Menon, K.Pingali,andN. Mateev. Fractal symbolic analysis. ACM Trans. Program. Lang. Syst., 25(6):776–813, 2003.
Kundu, S., Tatlock, Z., Lerner, S.: Proving optimizations correct using parameterized program equivalence. In: PLDI ’09, ACM (2009)
Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: a new approach to optimization. In: POPL ’09
Stepp, M., Tate, R., Lerner, S.: Equality-based translation validator for llvm. In: Proceedings of the 23rd International Conference on Computer Aided Verification. pp. 737–742. CAV’11, Springer-Verlag (2011)
B. Goldberg, L. D. Zuck, and C. W. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. Electr.NotesTheor.Comput.Sci.,132(1):53–71,2005.
G. C. Necula. Translation validation for an optimizing compiler. In PLDI, pages 83–94, 2000.
A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In TACAS, pages 151–166, 1998.
Kanade, A., Sanyal, A., Khedker, U.P.: Validation of gcc optimizers through trace generation. Softw. Pract. Exper. 39(6), 611–639 (Apr 2009)
Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems. pp. 151–166. TACAS ’98, Springer-Verlag (1998)
Tristan, J.B., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for llvm. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 295–305. PLDI ’11, ACM (2011)
Zaks, A., Pnueli, A.: Covac: Compiler validation by program analysis of the crossproduct. In: Proceedings of the 15th International Symposium on Formal Methods. pp. 35–51. FM ’08, Springer-Verlag (2008)
…

STILL MISSING…

An Automatic Equivalence Checker

that works across a long and unknown sequence of transformations

for practically useful programs written in commonly-used syntaxes

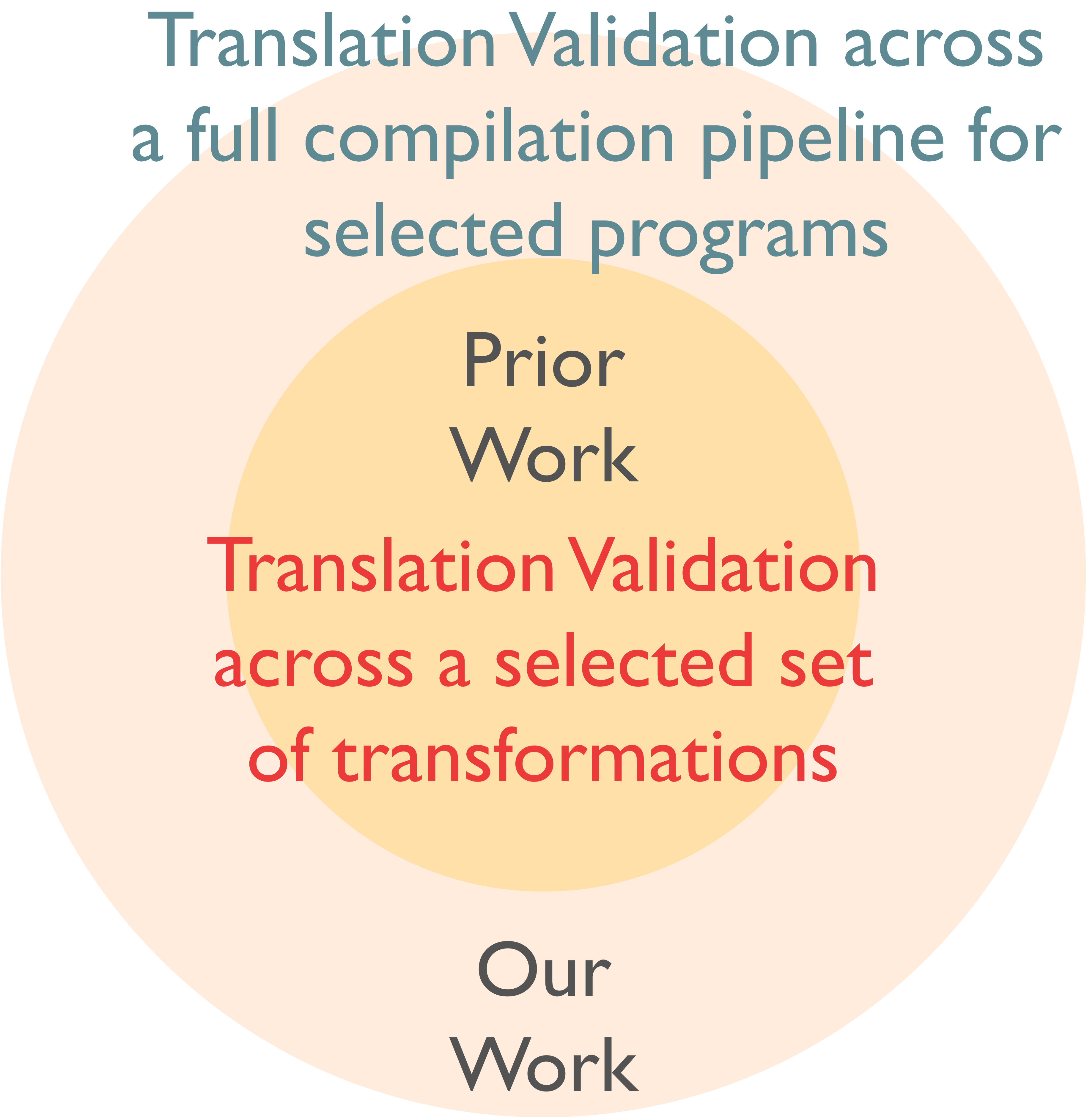
in a scalable way

EQUIVALENCE CHECKING RESEARCH ROADMAP

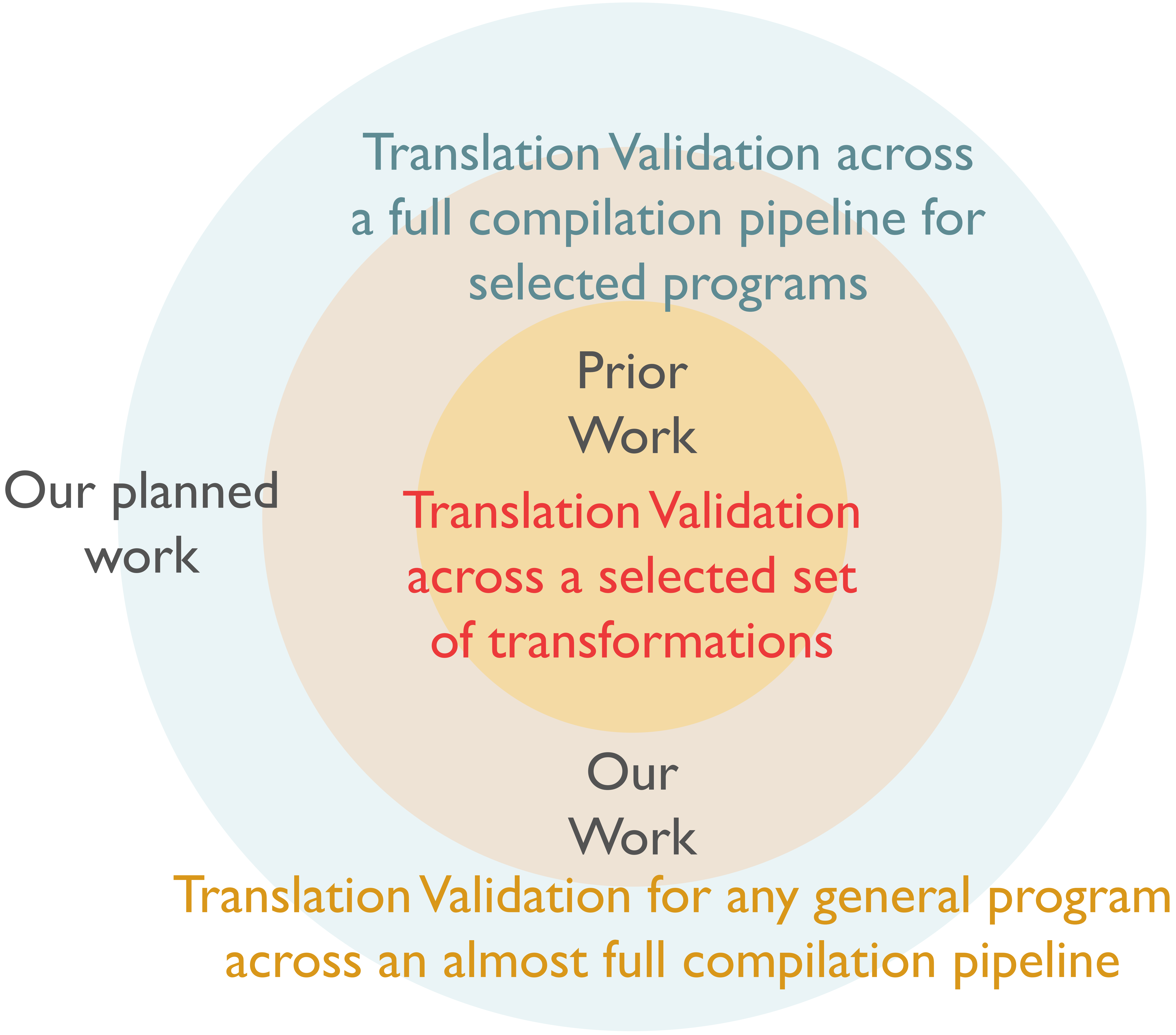
Prior
Work

Translation Validation
across a selected set
of transformations

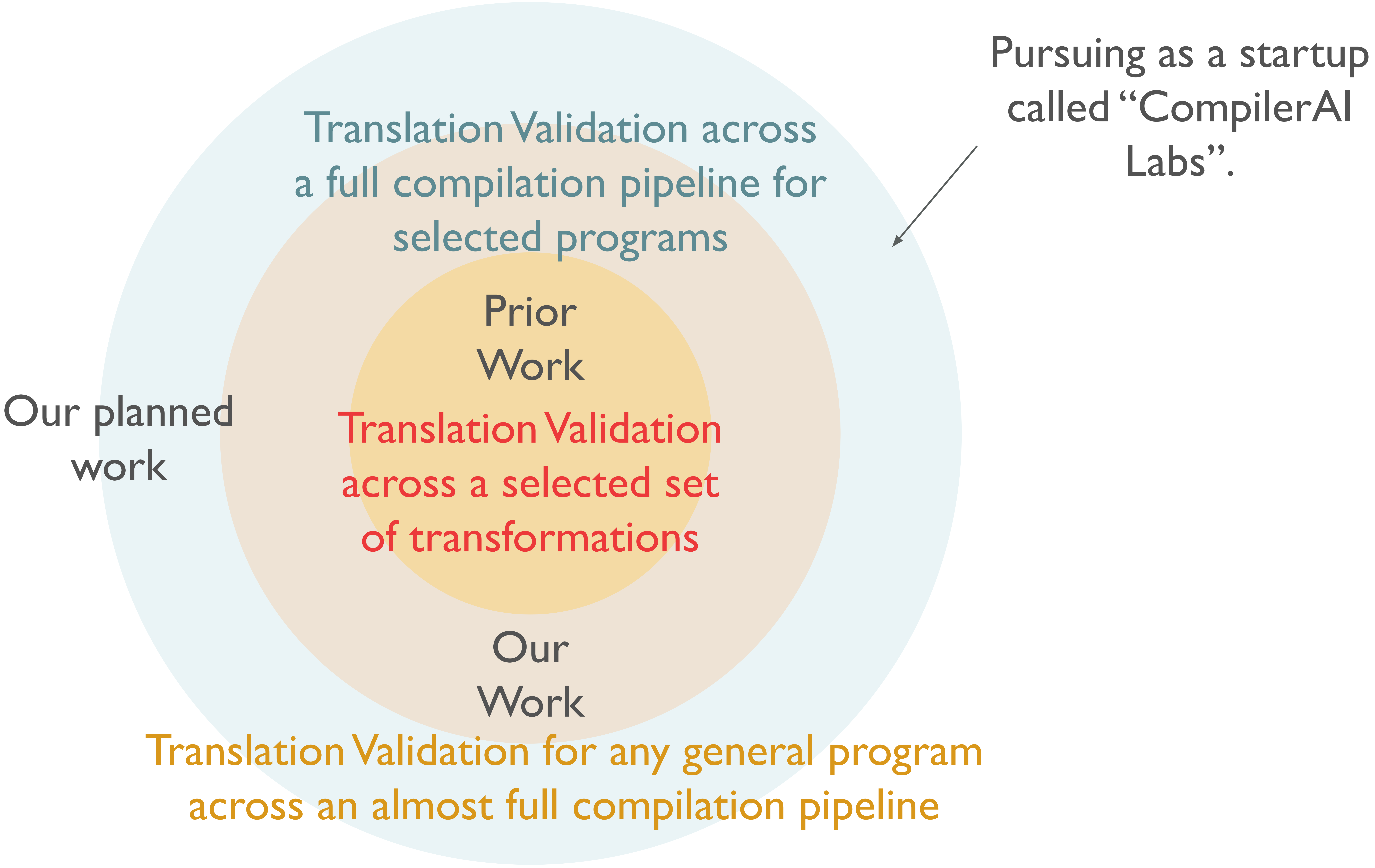
EQUIVALENCE CHECKING RESEARCH ROADMAP



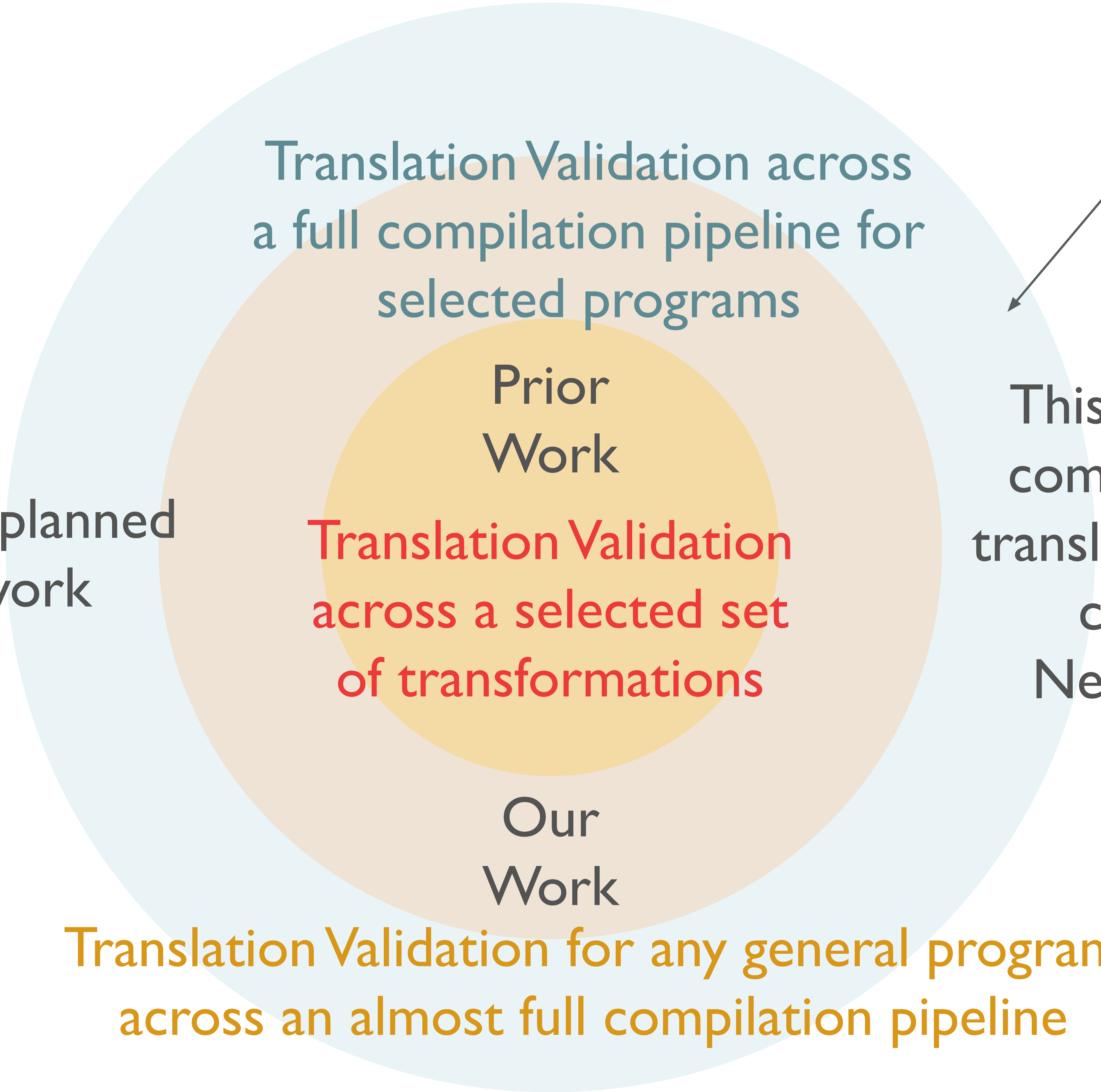
EQUIVALENCE CHECKING RESEARCH ROADMAP



EQUIVALENCE CHECKING RESEARCH ROADMAP



EQUIVALENCE CHECKING RESEARCH ROADMAP



Pursuing as a startup called "CompilerAI Labs".

This will be the first certified compiler that uses automatic translation validation as the sole certification technique. Needed for Aircrafts, Cars, Trains, …

Key Ideas

- **Scalable Search Algorithm for a Bisimulation Relation**
  - Includes Modeling of Undefined Behaviour Semantics
  - Leverages SMT Solvers, e.g., Z3, Yices, CVC4 for discharge of first-order-logic proof obligations

[M. Dahiya et. al.. Blackbox Equivalence Checking Across Compiler Optimizations, , APLAS 2017]
[M. Dahiya et. al., Modeling Undefined Behaviour for Checking Equivalence Across Compiler Optimizations, HVC 2017]

Key Ideas

- **Automatic Identification of bisimulation proofs**
  - Would require millions of years through a naive search
  - Translation Validator from C-to-x86

[S. Gupta et. al., Effective Use of SMT Solvers for Program Equivalence Checking through Invariant Sketching and Query Decomposition, SAT 2018]
[S. Gupta et. al., Counterexample-Guided Correlation Algorithm for Translation Validation, OOPSLA 2020]

EXAMPLE 3: VECTORIZATION

```
int LEN, a[LEN], b[LEN];
int c[LEN], d[LEN];
C0: void s441() {
C1:   for (int i = 0; i < LEN; i++) {
C2:     if (d[i] < 0) {
C3:       a[i] += b[i] * c[i];
C4:     } else if (d[i] == 0) {
C5:       a[i] += b[i] * b[i];
C6:     } else {
C7:       a[i] += c[i] * c[i];
C8:     }
C9:   }
C10:}
```

EXAMPLE 3 :VECTORIZATION

```
int LEN, a[LEN], b[LEN];
int c[LEN], d[LEN];
C0: void s441() {
C1:   for (int i = 0; i < LEN; i++) {
C2:     if (d[i] < 0) {
C3:       a[i] += b[i] * c[i];
C4:     } else if (d[i] == 0) {
C5:       a[i] += b[i] * b[i];
C6:     } else {
C7:       a[i] += c[i] * c[i];
C8:     }
C9:   }
C10:}
```

```
A0:  s441:
A1:   r1 = 0
A2:     xmm1 = a[r1 .. r1+3]
A3:     xmm2 = xmm1 + b[r1 .. r1+3]*c[r1 .. r1+3]
A4:     xmm3 = xmm1 + b[r1 .. r1+3]*b[r1 .. r1+3]
A5:     xmm4 = xmm1 + c[r1 .. r1+3]*c[r1 .. r1+3]
        // pcmpgtd
A6:     xmm0 = (d[r1] < 0), .. , (d[r1+3] < 0)
A7:     xmm1 = xmm0 ? xmm2 : xmm1   // pblendvb
        // pcmpeqd
A8:     xmm0 = (d[r1] == 0), .. , (d[r1+3] == 0)
A9:     xmm1 = xmm0 ? xmm3 : xmm1   // pblendvb
        // pcmpgtd
A10:    xmm0 = (d[r1] > 0), .. , (d[r1+3] > 0)
A11:    xmm1 = xmm0 ? xmm4 : xmm1   // pblendvb
A12:    a[r1 .. r1+3] = xmm1
A13:    r1 += 4
A14:    if (r1 != LEN) goto A2
A15:  ret
```

Key Idea
Counterexample Guided Best-First Search

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

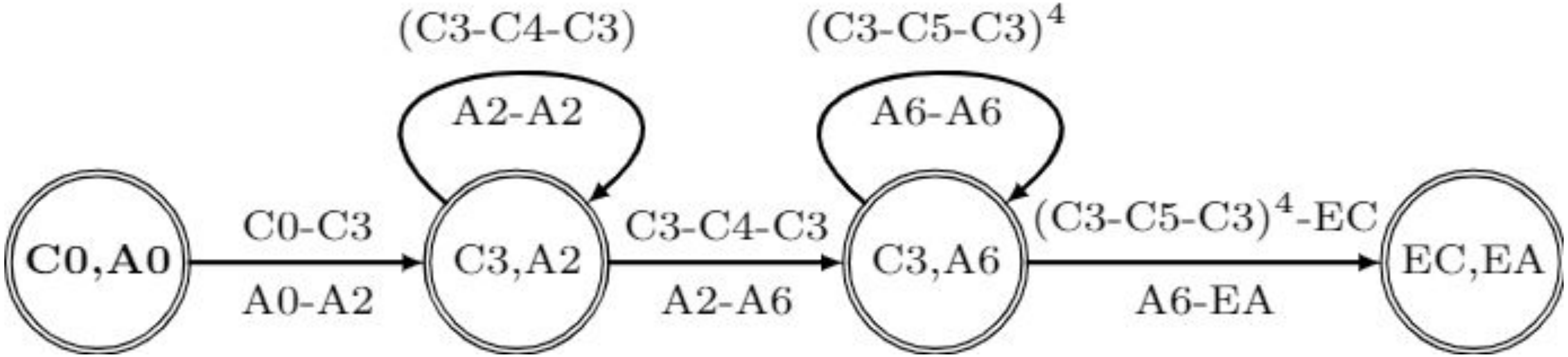
```
int loopUnrolling() {
   int sum = 0;
   int mid = LEN /2;
   for ( int i = 0; i < mid ; i ++) {
     sum += c[a[i]];
   }
   for ( int i = mid; i < LEN ; i +=4) {
     sum += b[ i ];
     sum += b[ i+1 ];
     sum += b[ i +2];
     sum += b[ i +3];
   }
   return sum ;
}
```

```
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
```

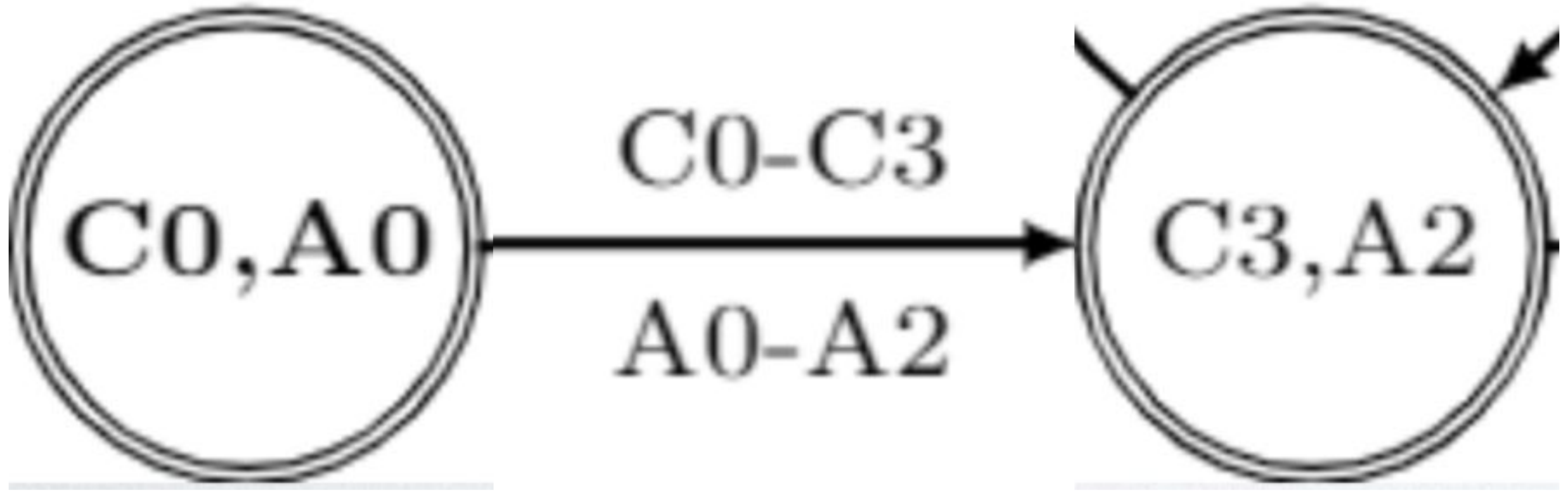


Incremental Construction of the Product CFG

Incremental Construction of the Product CFG

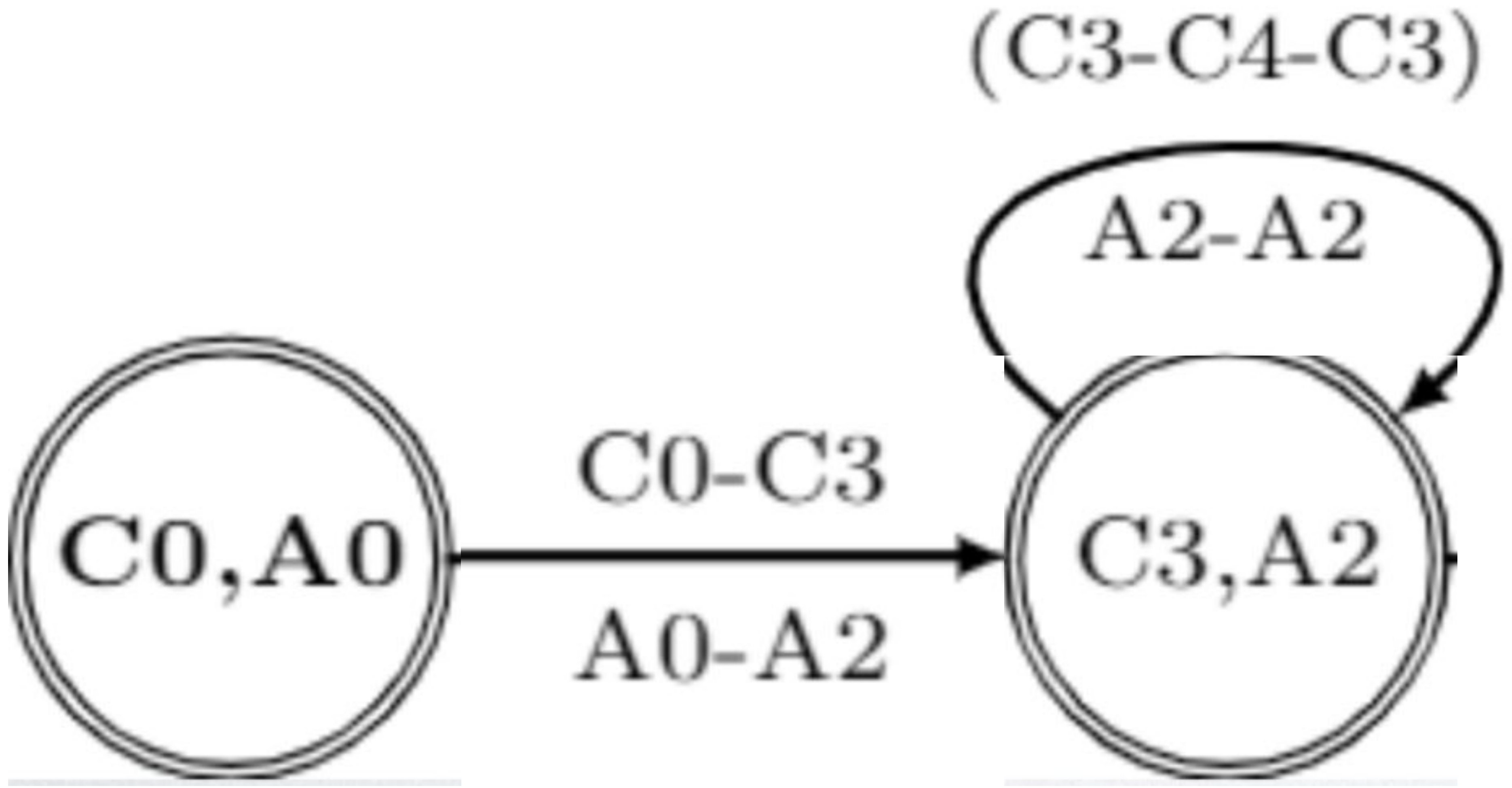
Incremental Construction of the Product CFG

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



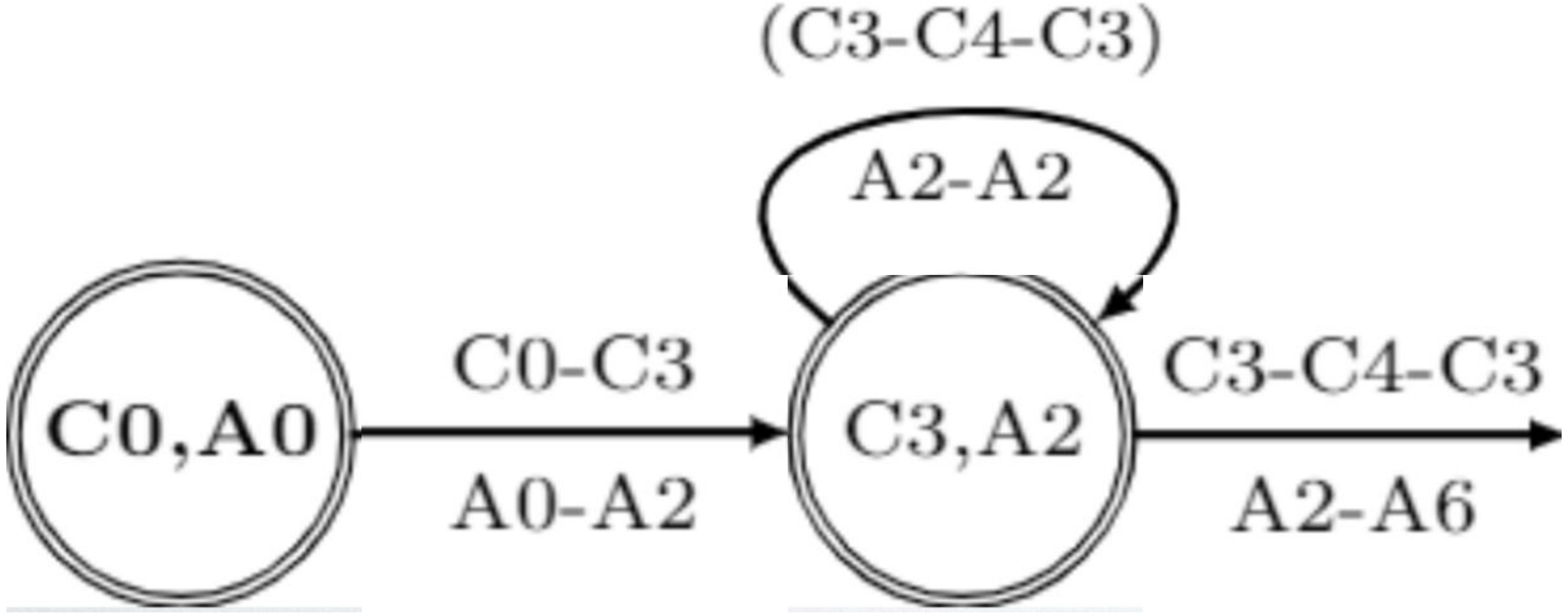
Infer Invariants at (C3,A2)

Incremental Construction of the Product CFG

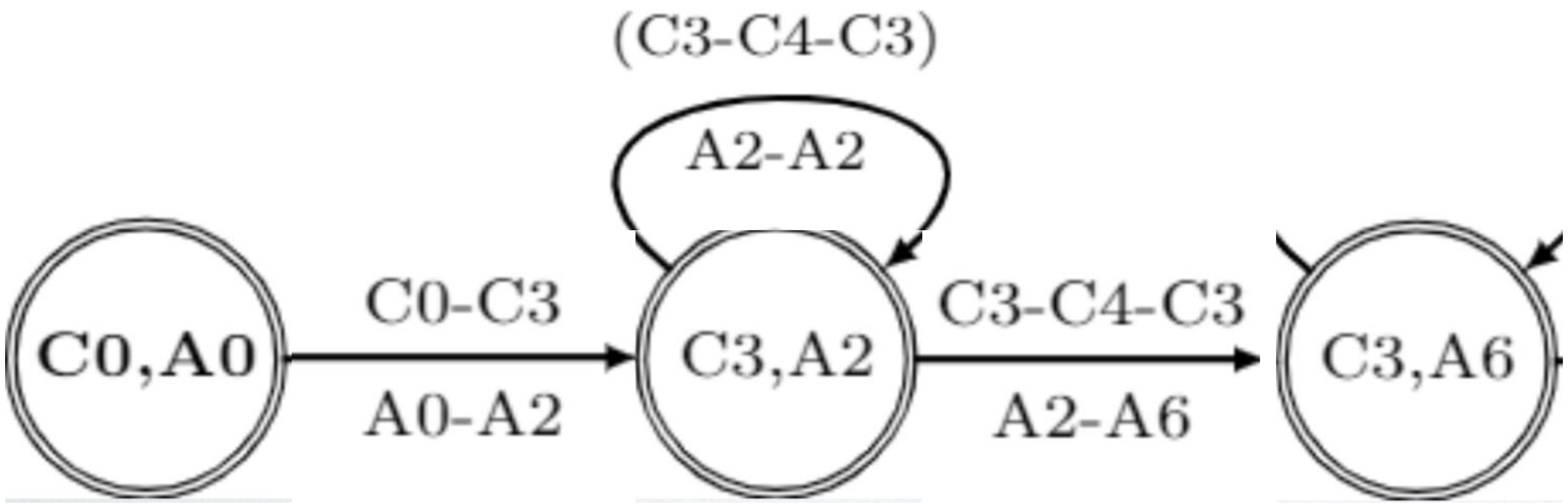


Relax Invariants
at (C3,A2)

Incremental Construction of the Product CFG

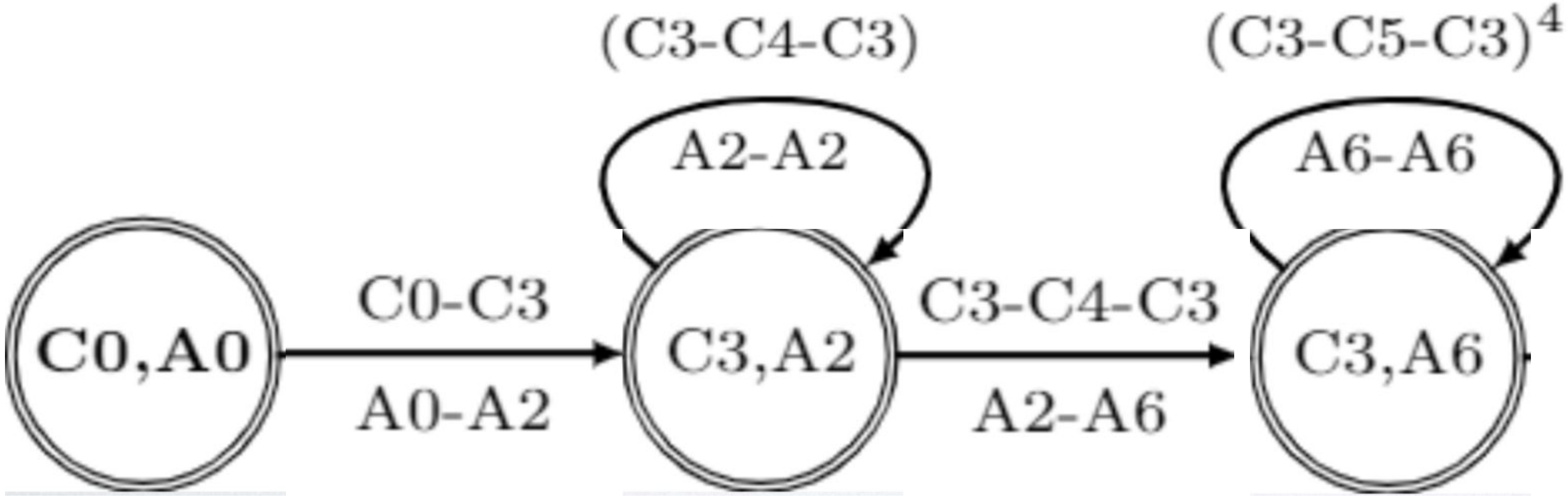


Incremental Construction of the Product CFG



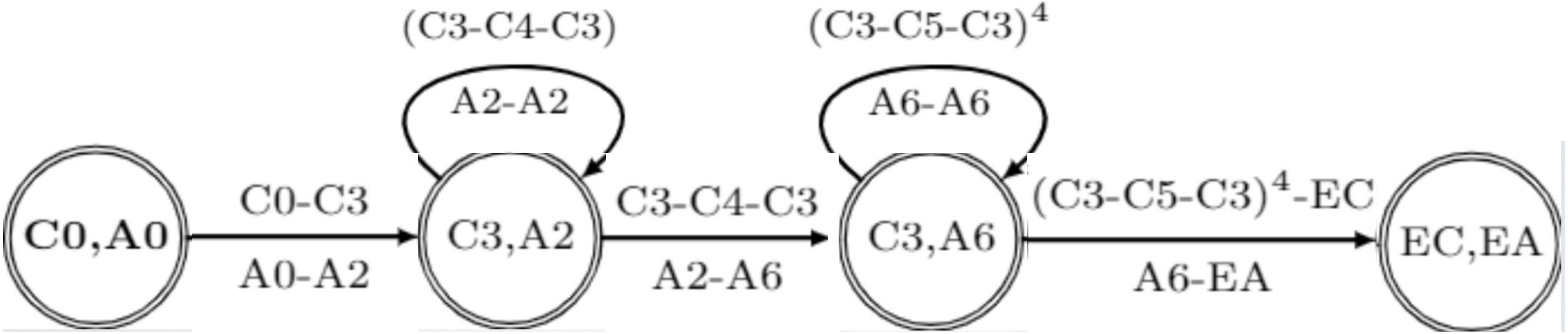
Infer Invariants at
(C3,A6)

Incremental Construction of the Product CFG



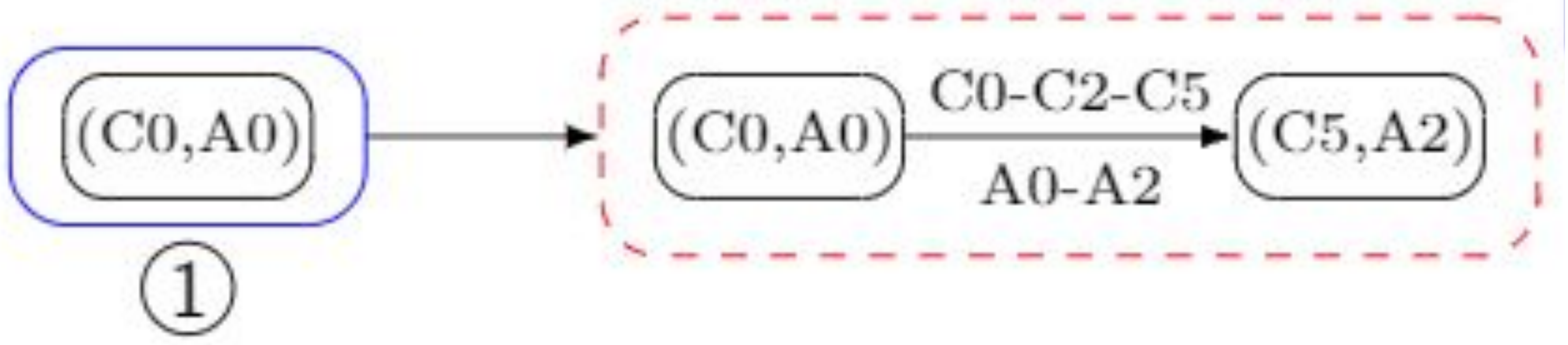
Relax Invariants
at (C3,A6)

Incremental Construction of the Product CFG

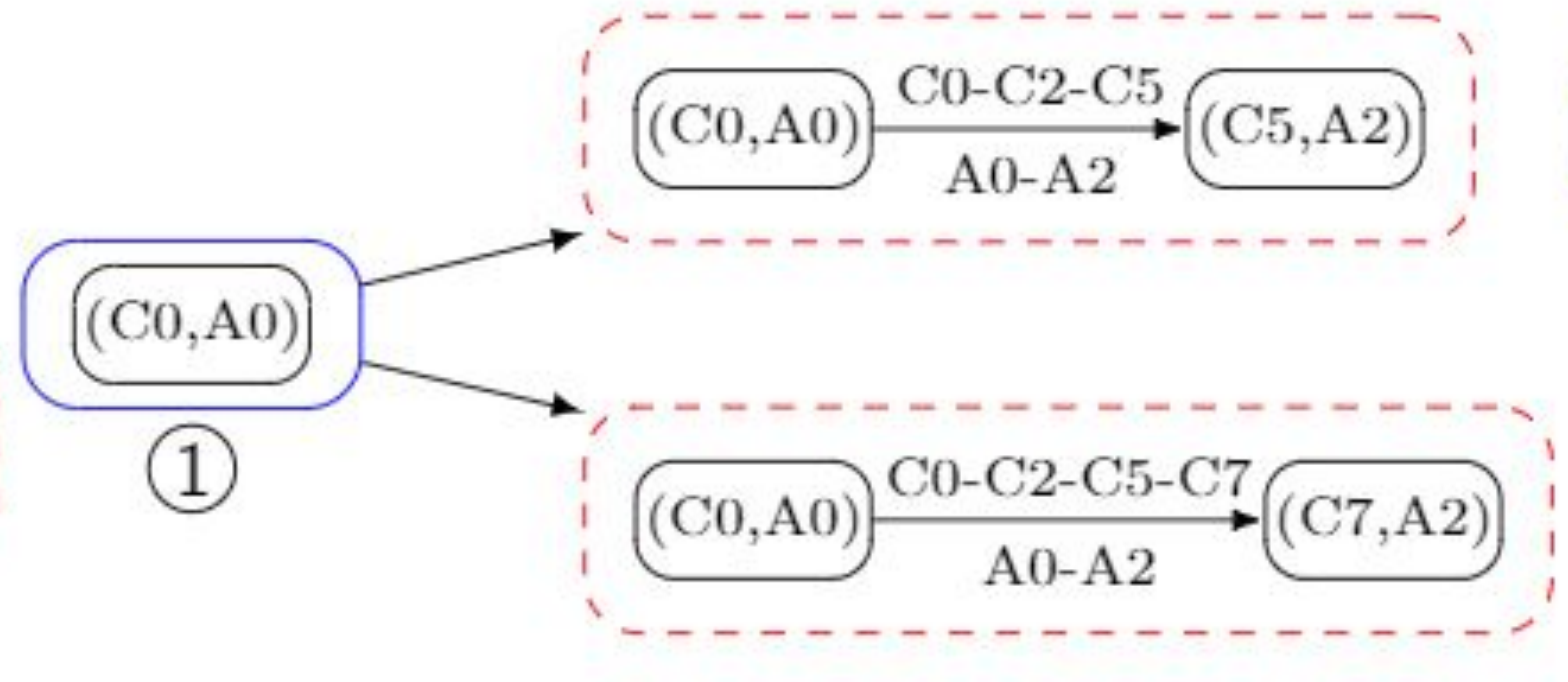


Check equivalence of return values under inferred invariants

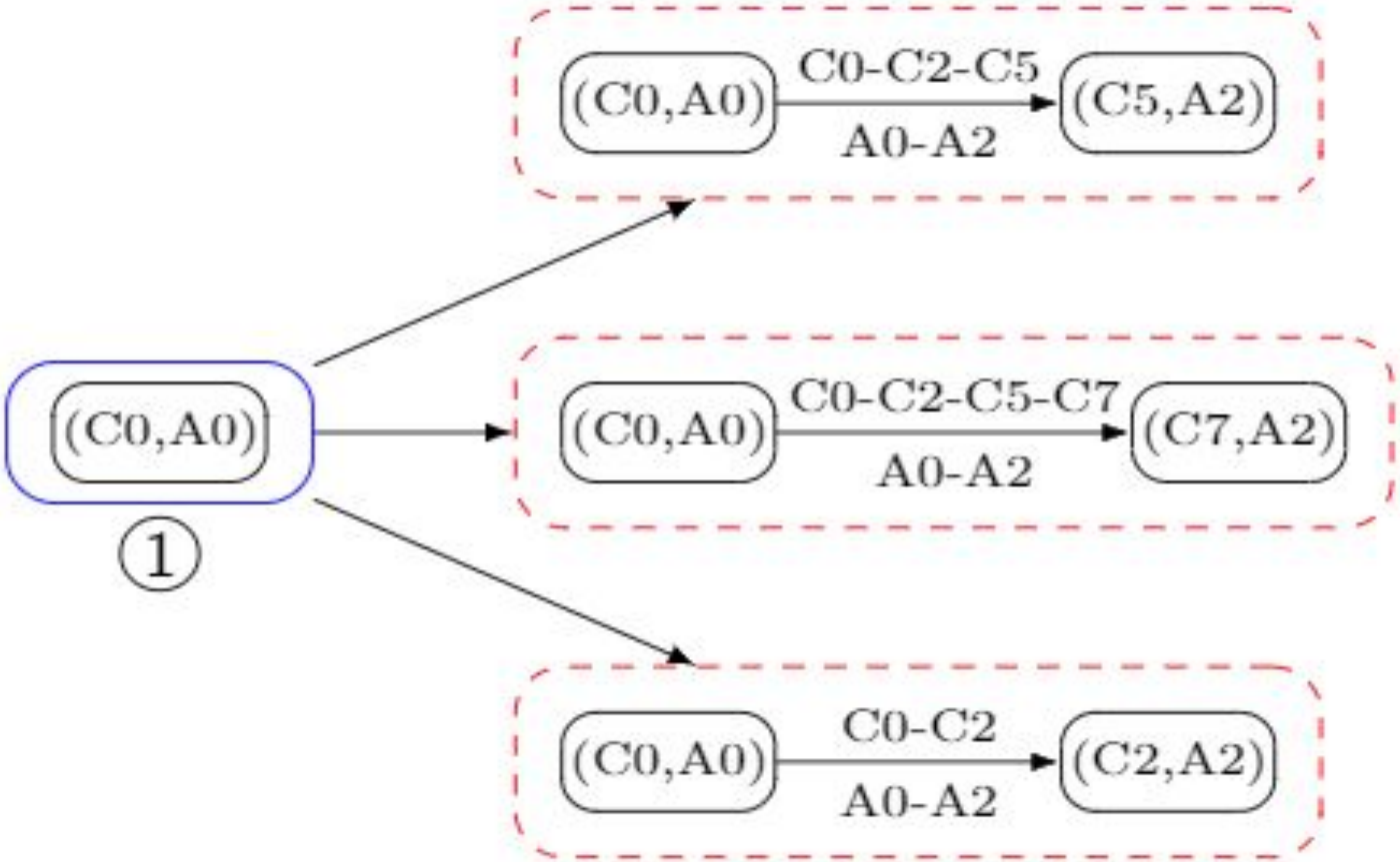
SEARCH SPACE



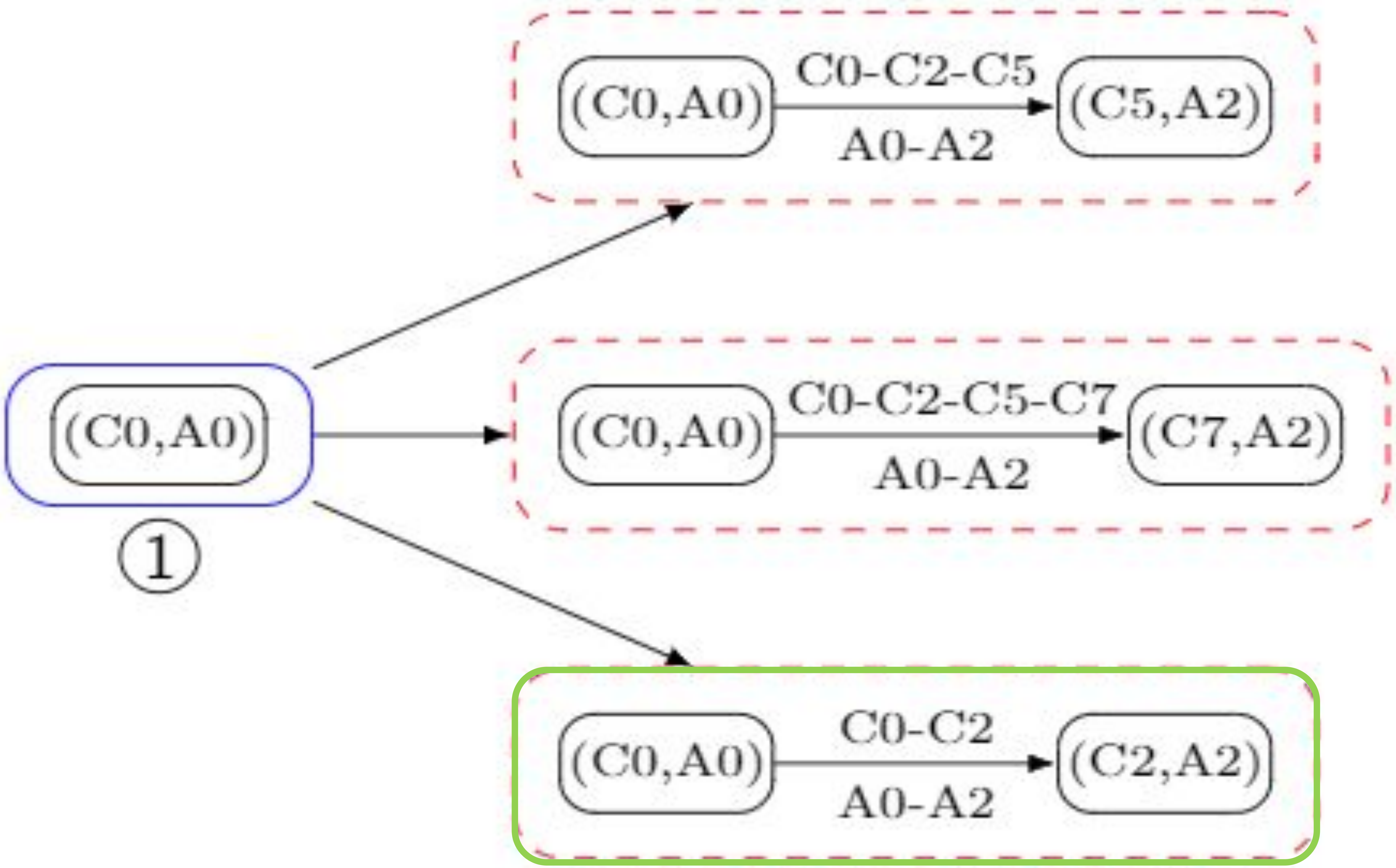
SEARCH SPACE



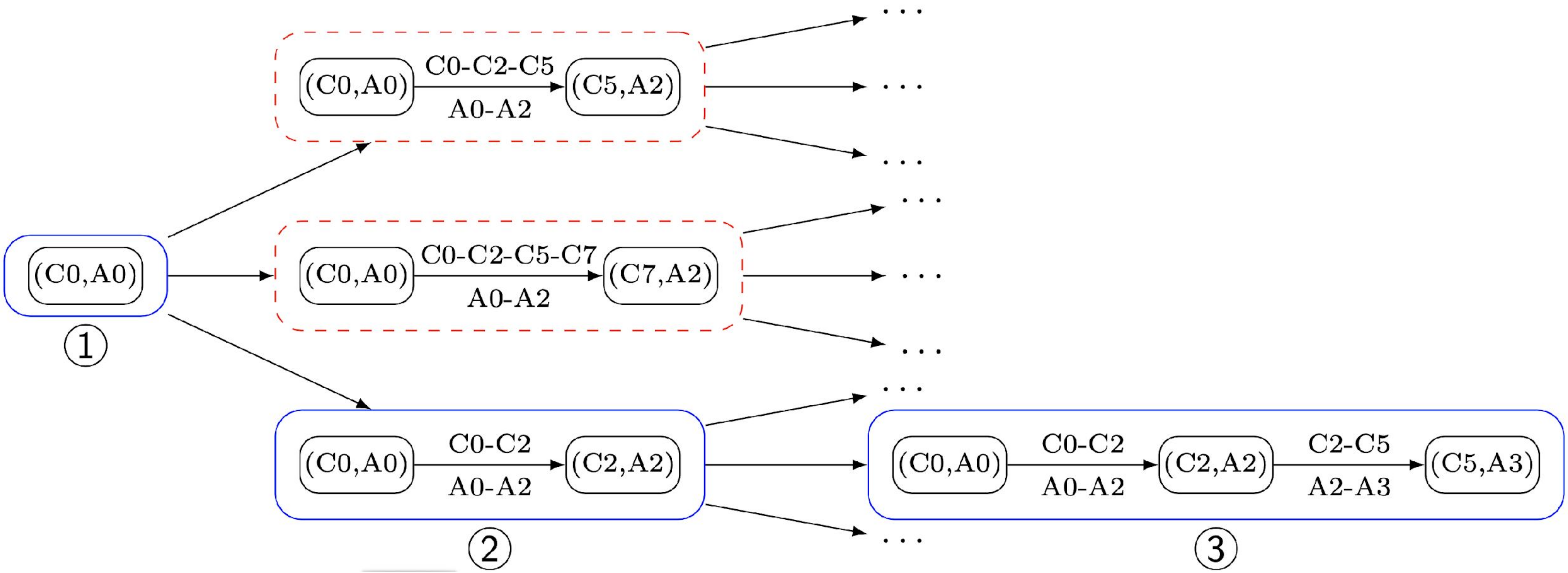
SEARCH SPACE



SEARCH SPACE

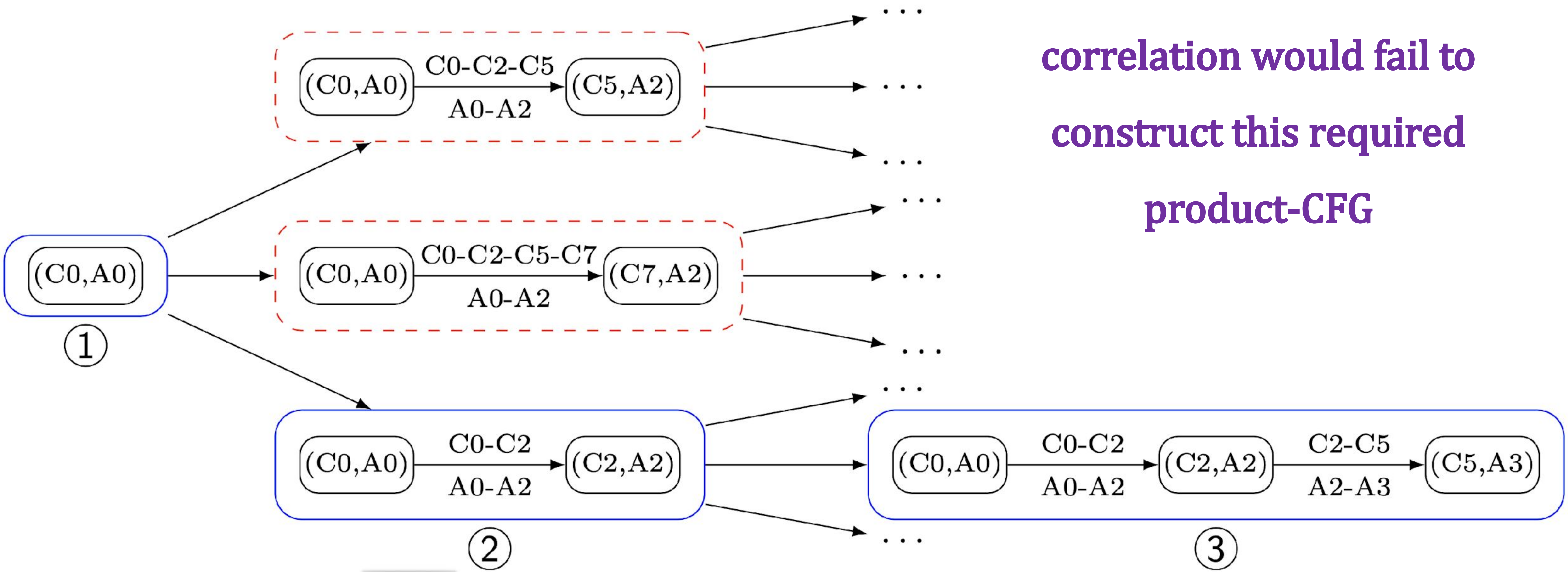


SEARCH SPACE



Exhaustive search would take years to compute equivalence

SEARCH SPACE



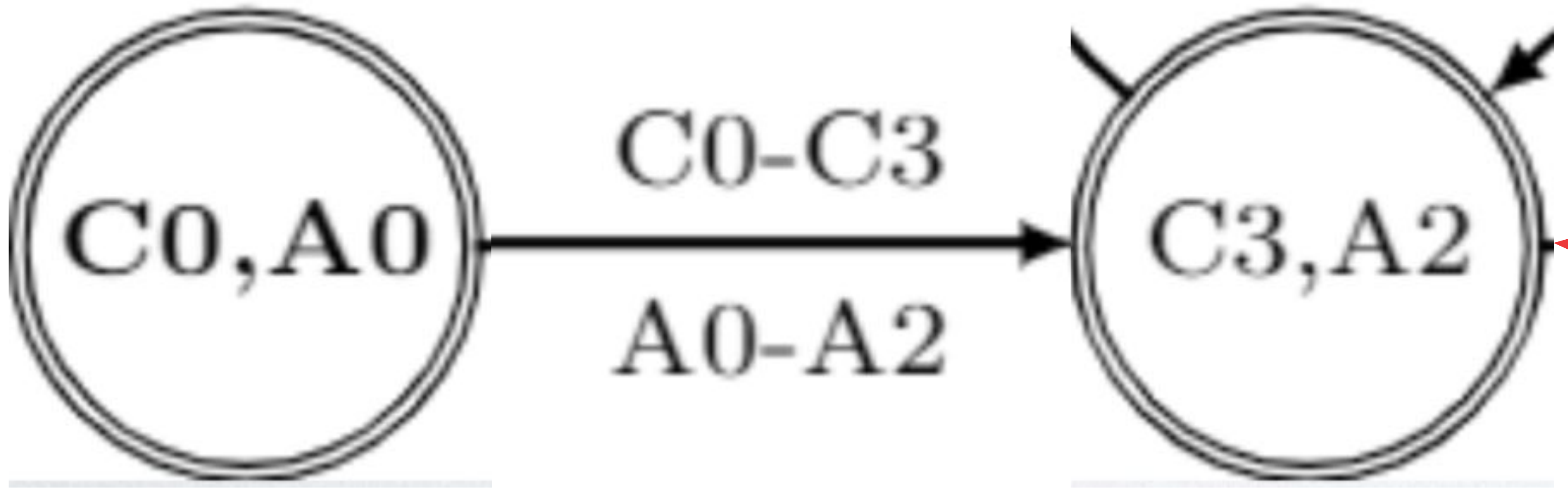
Prior work on data driven correlation would fail to construct this required product-CFG

Exhaustive search would take 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



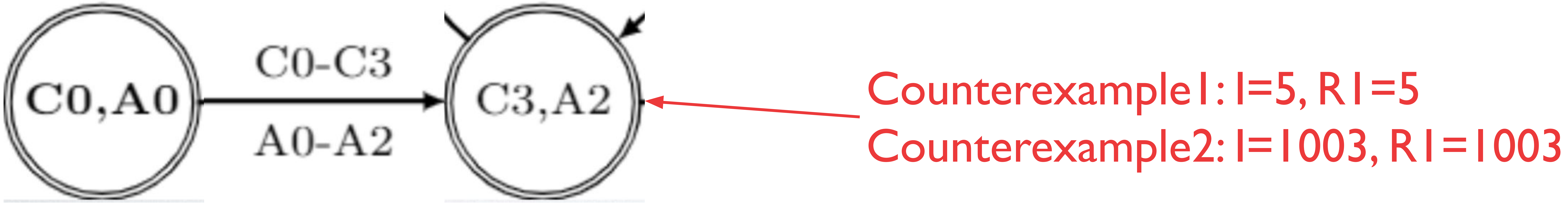
Counterexamples identified at (C3,A2) during invariant inference

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.

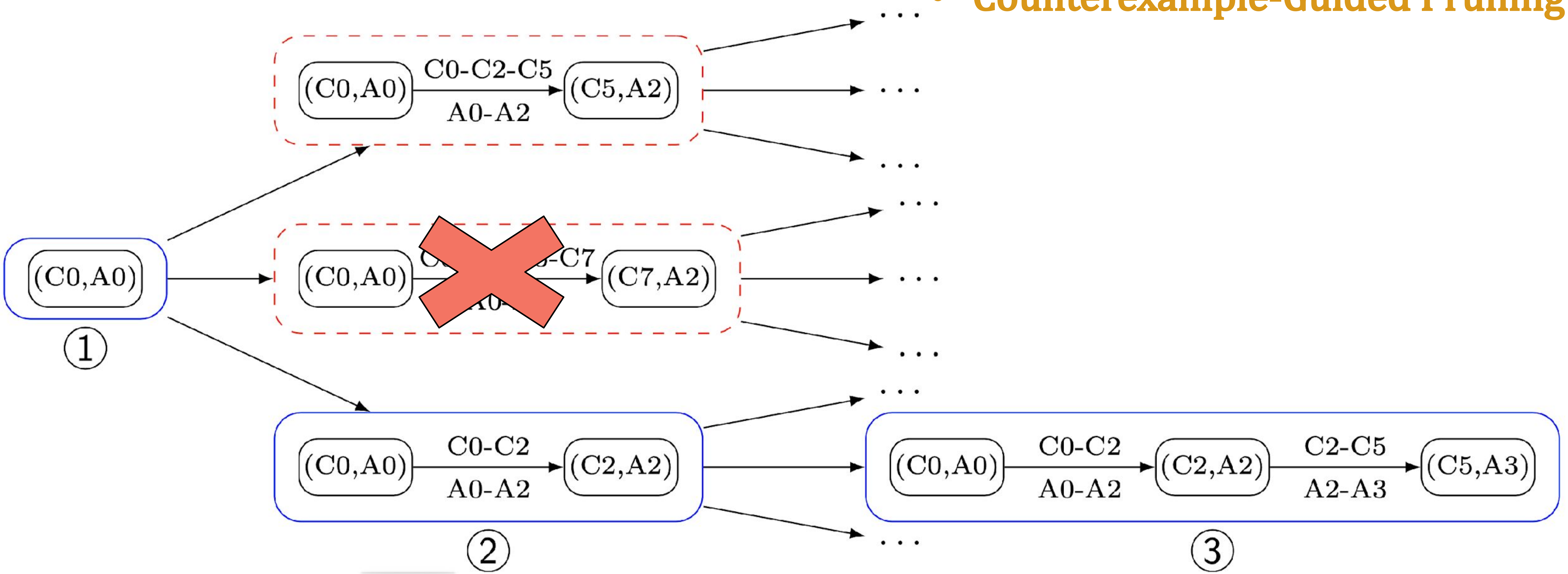
The concrete state would involve valuations for (related) variables of both C and A.



Infer Invariants at (C3,A2)

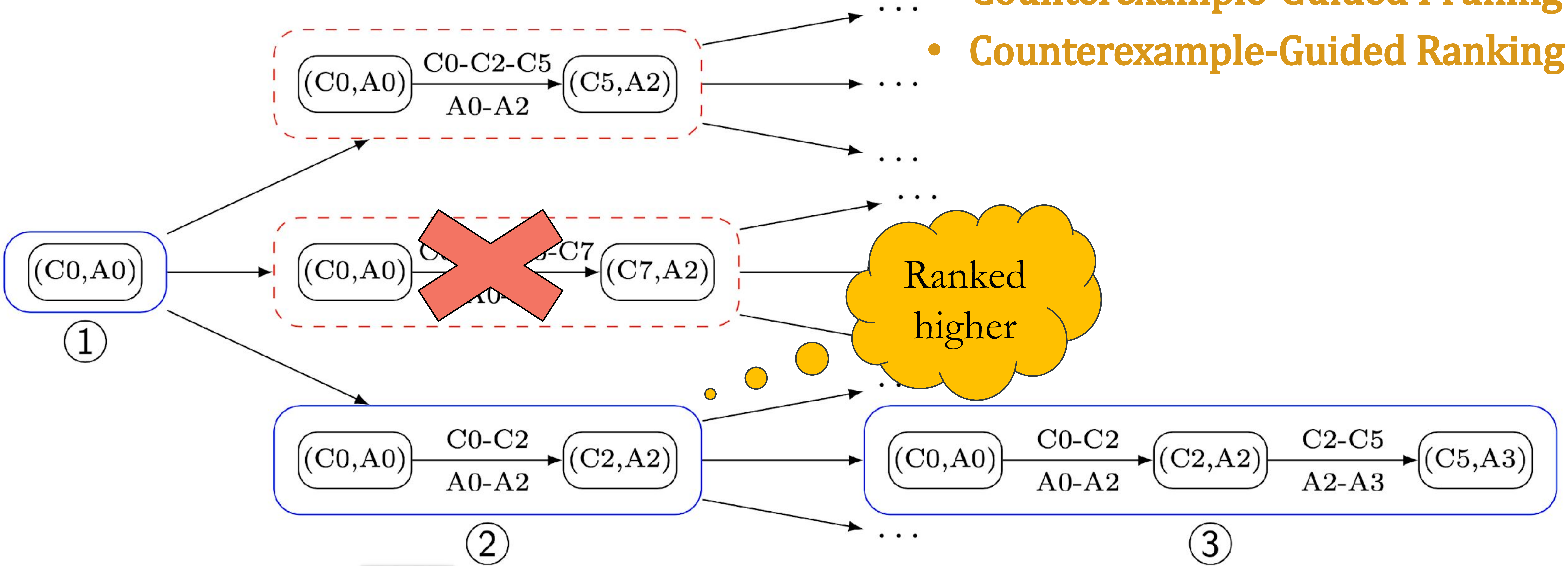
COUNTEREXAMPLE GUIDED BEST-FIRST SEARCH

- **Counterexample-Guided Pruning**



COUNTEREXAMPLE GUIDED BEST-FIRST SEARCH

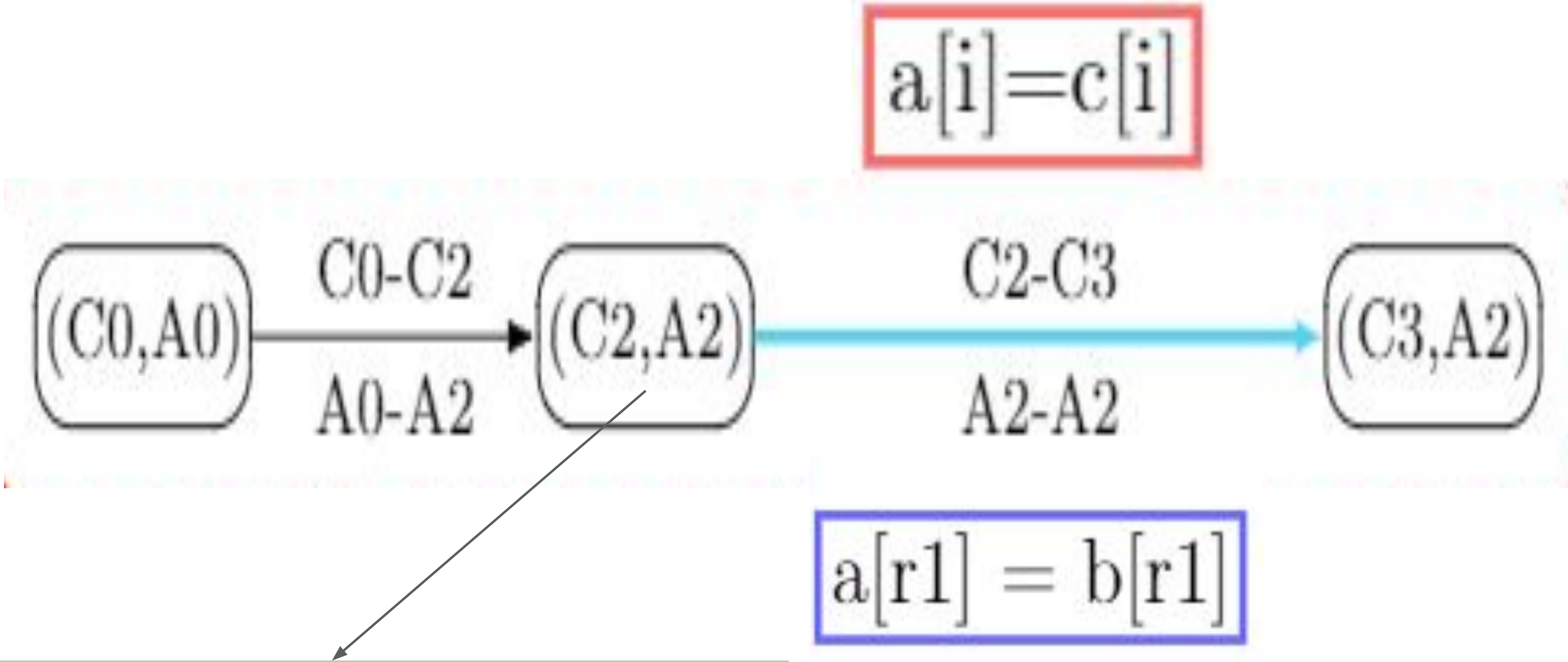
- **Counterexample-Guided Pruning**
- **Counterexample-Guided Ranking**



Counterexample Guided Best-First Search

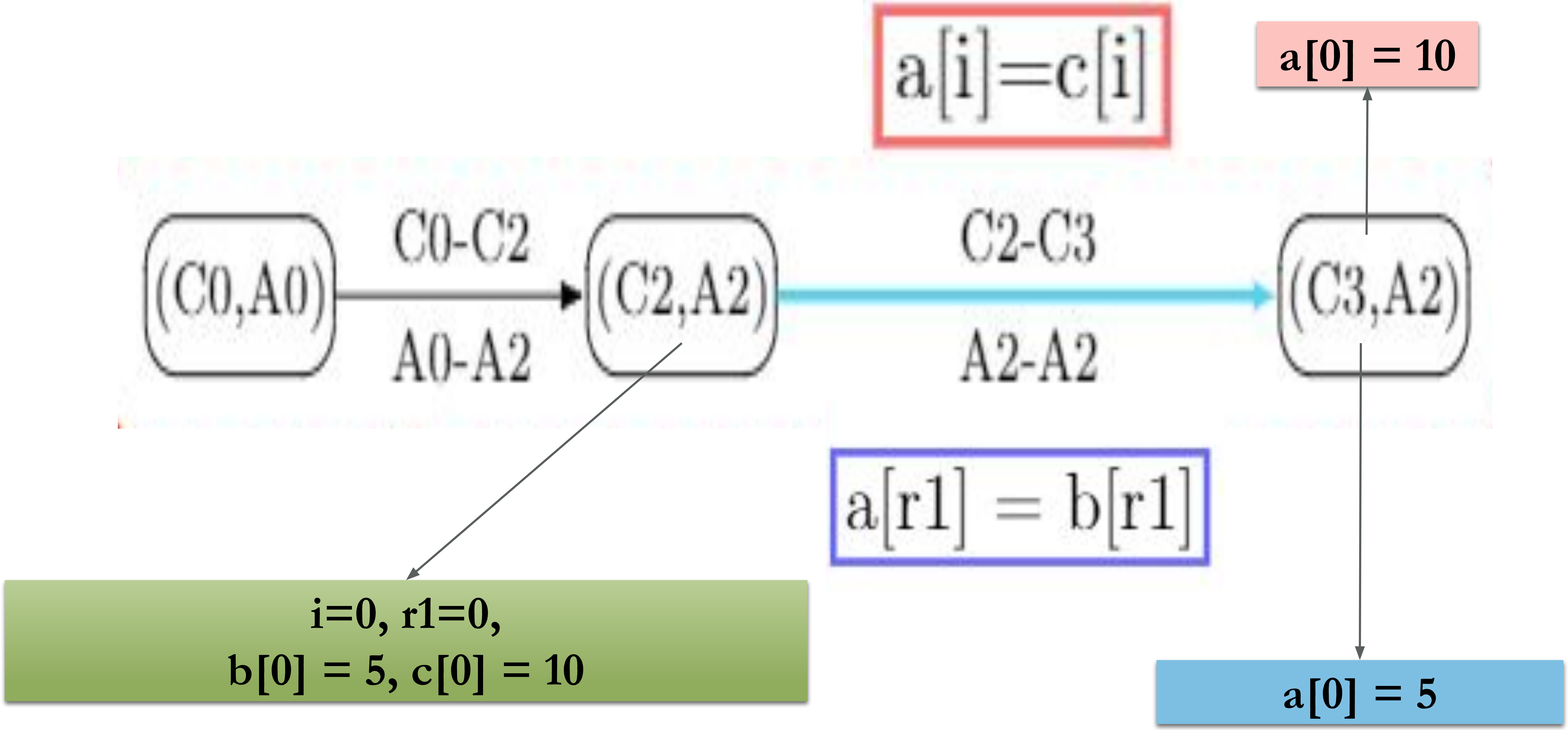
- Counterexample Guided Pruning

COUNTEREXAMPLE EXECUTION

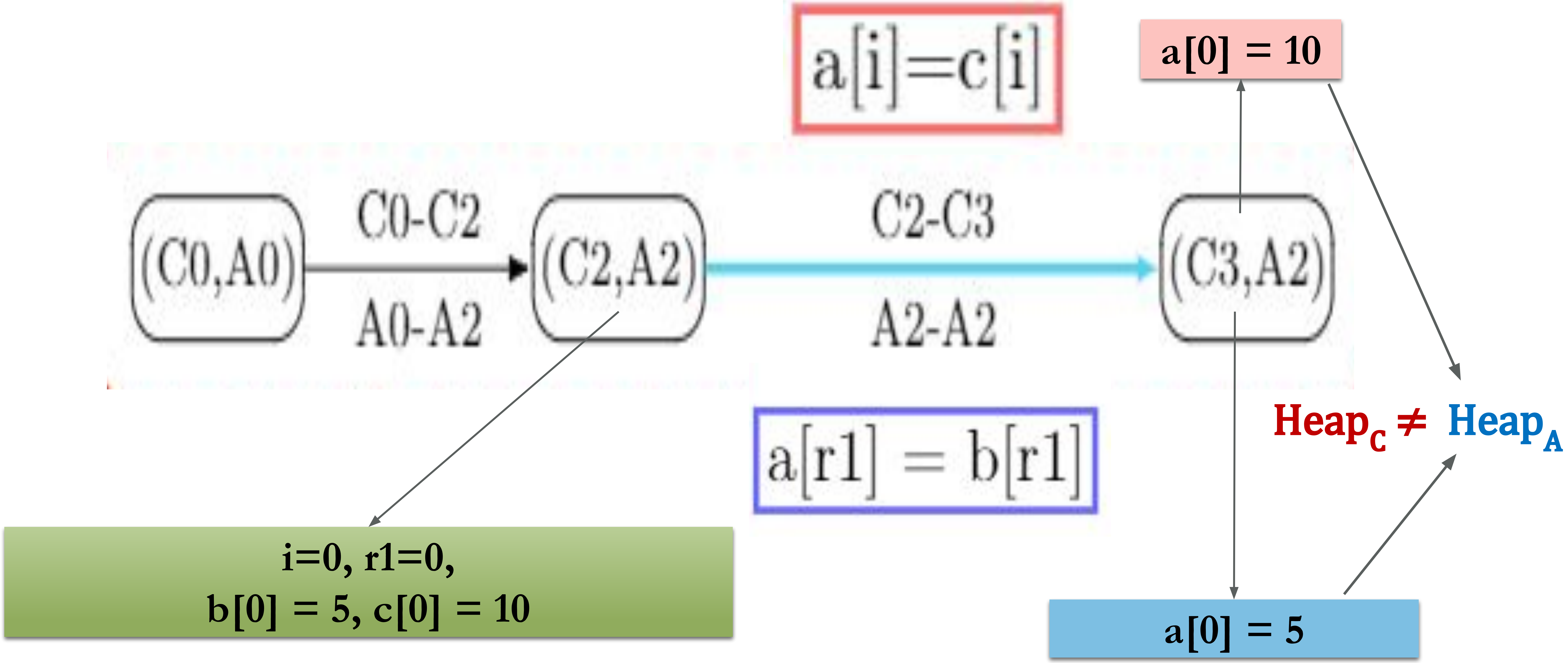


i=0, r1=0,
b[0] = 5, c[0] = 10

COUNTEREXAMPLE EXECUTION

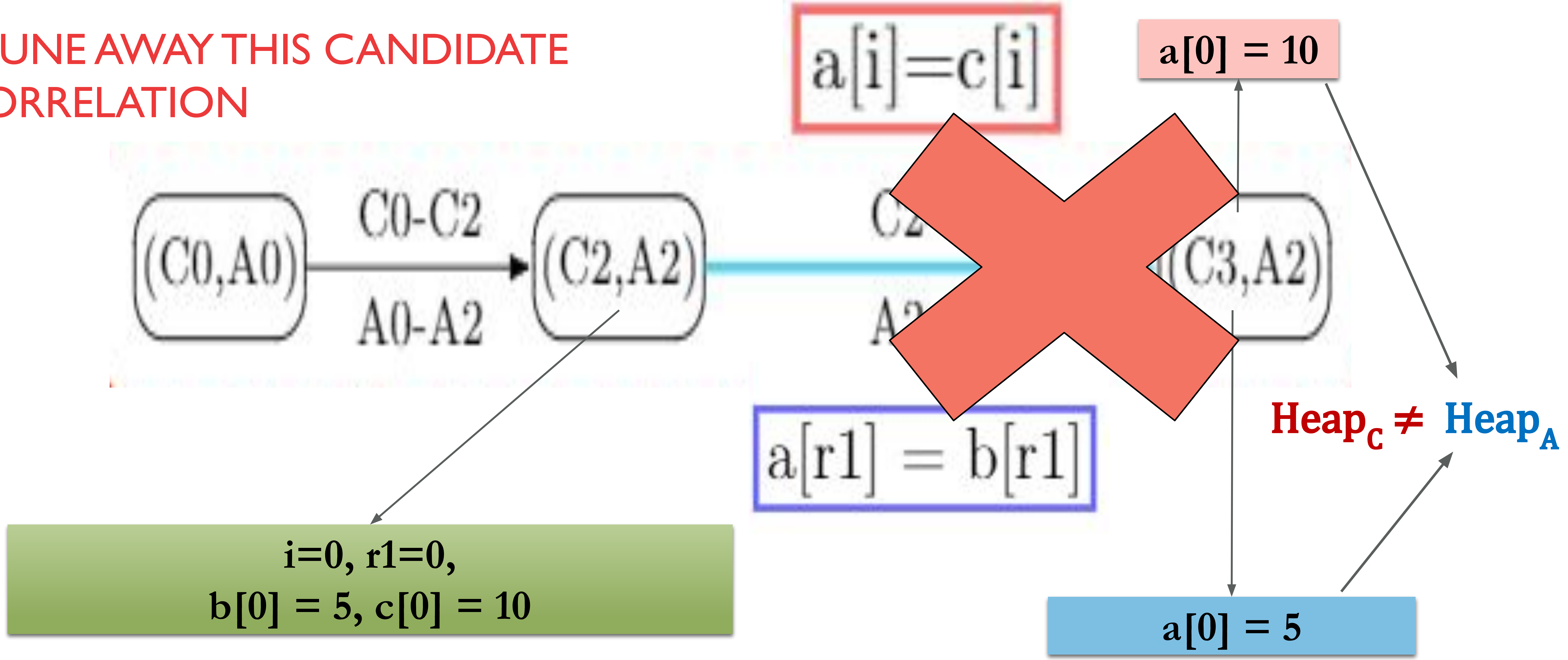


Counterexample Guided Pruning



Counterexample Guided Pruning

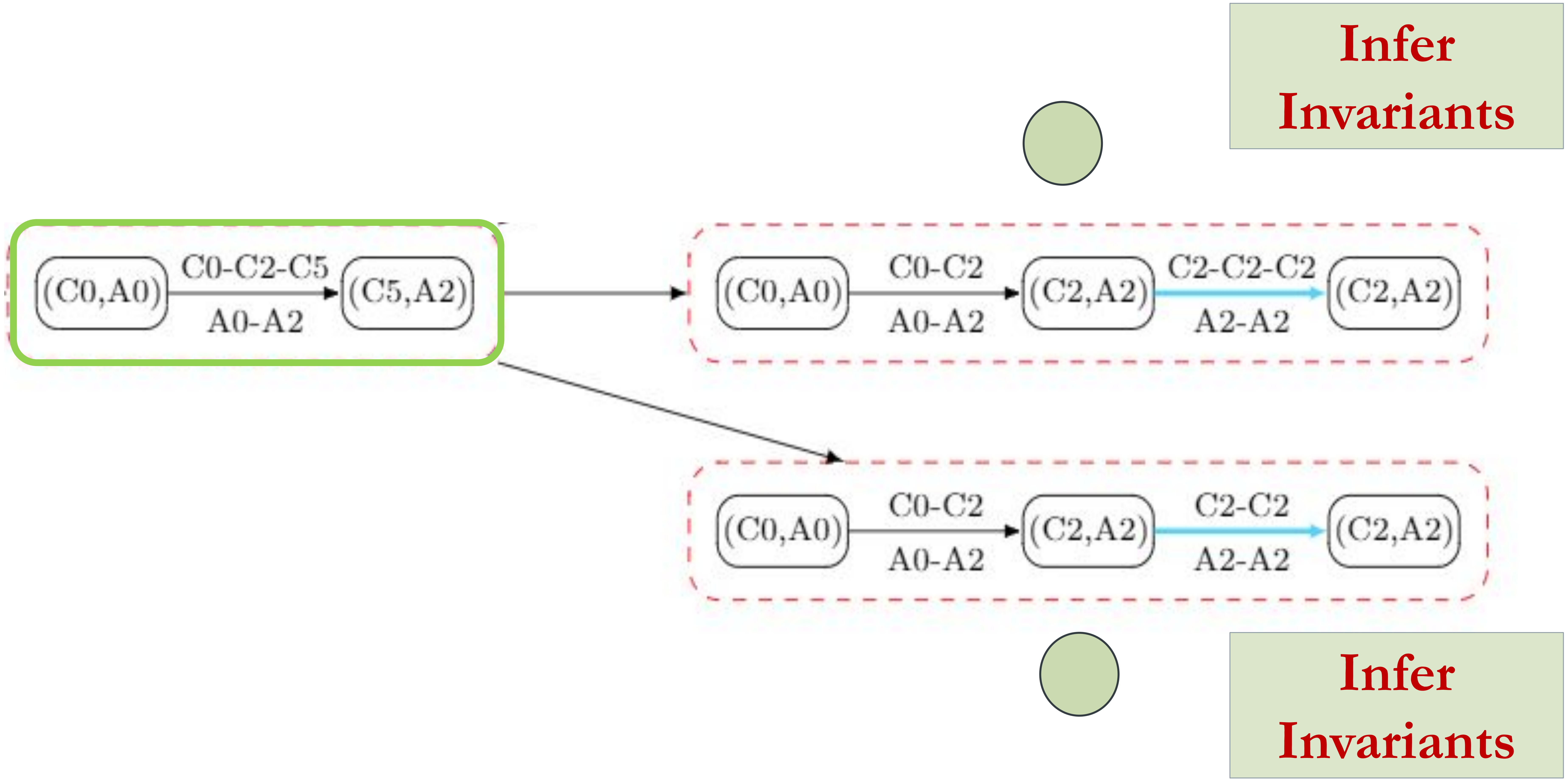
PRUNE AWAY THIS CANDIDATE CORRELATION



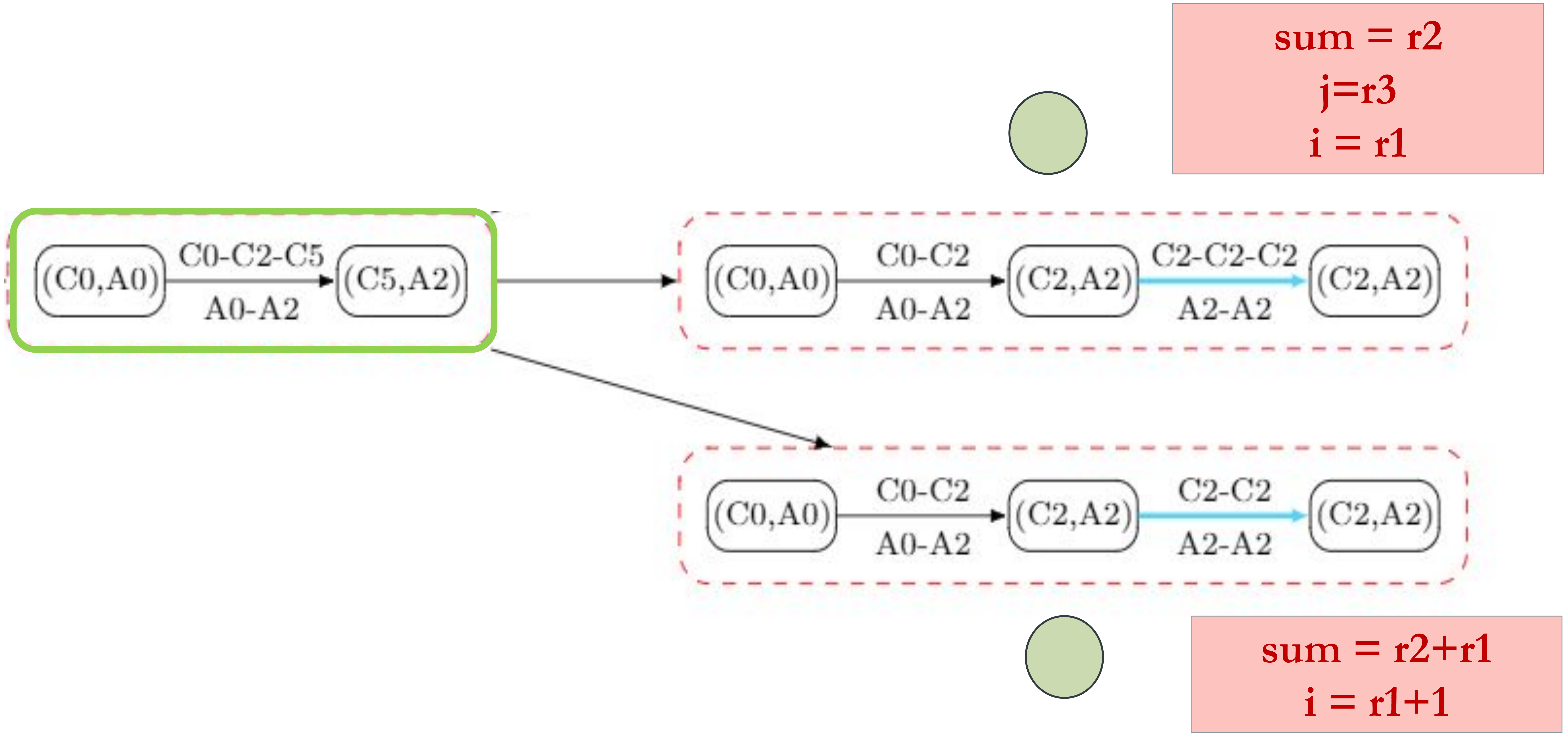
Counterexample Guided Best-First Search

- Counterexample Guided Pruning
- Counterexample Guided Ranking

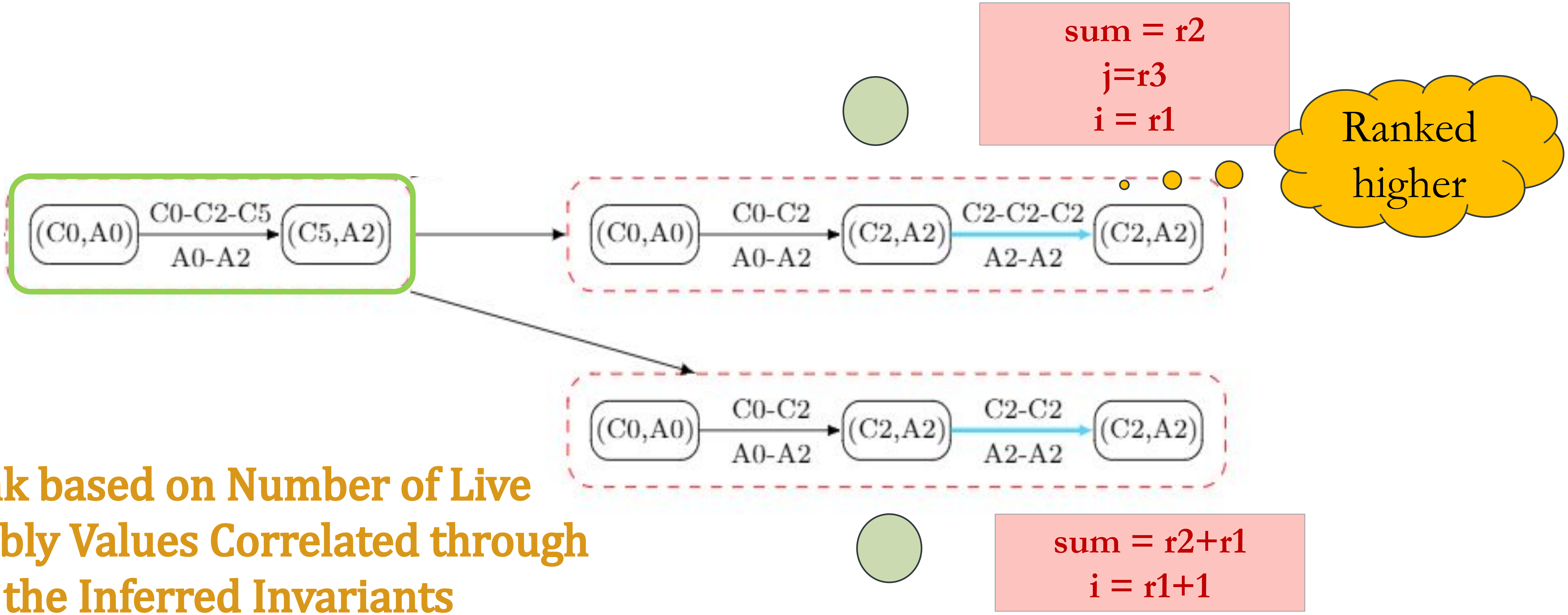
Infer Invariant Covers for Executed Counterexamples



Infer Invariant Covers for Executed Counterexamples

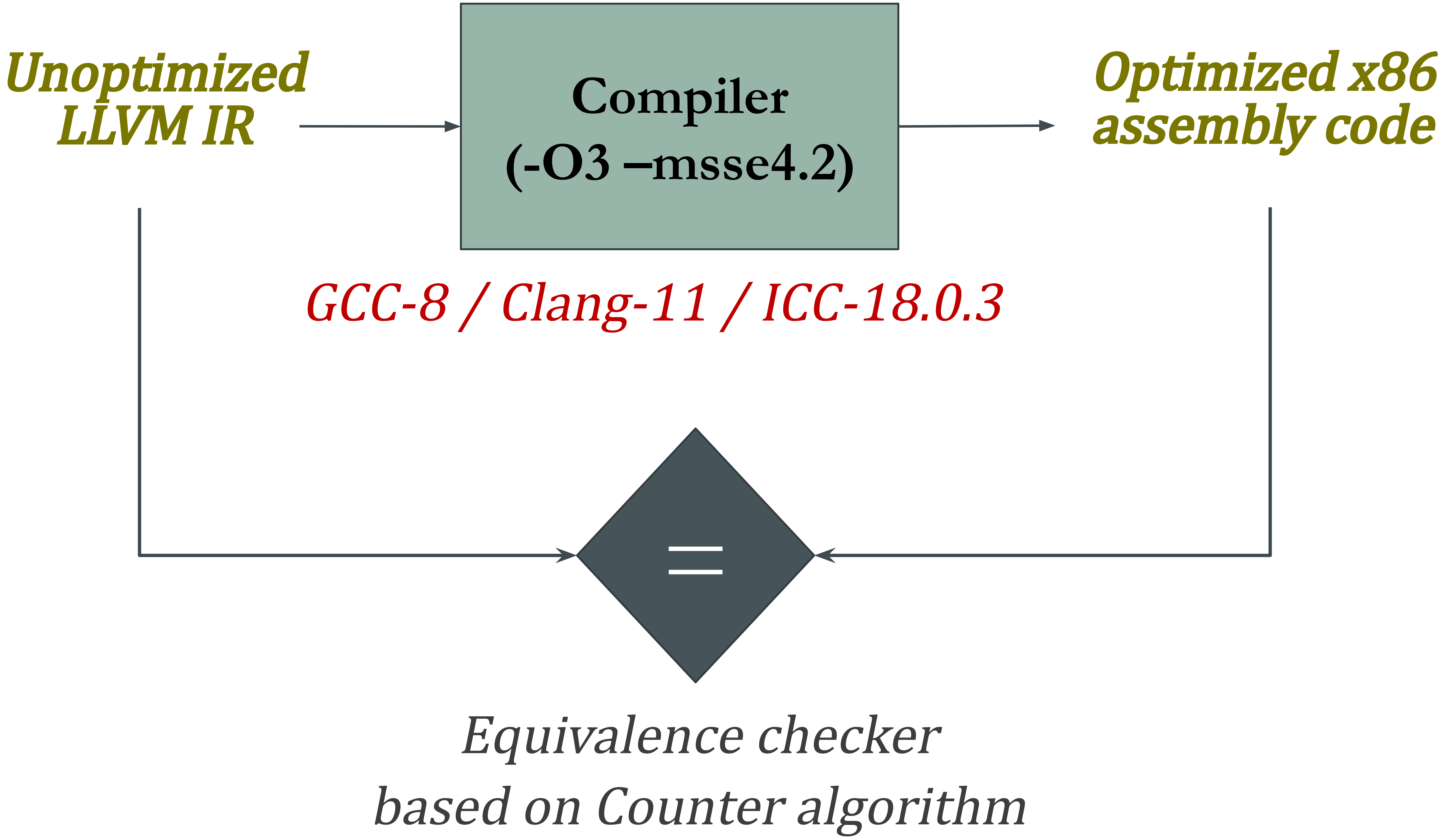


Infer Invariant Covers for Executed Counterexamples



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

Counter Evaluation



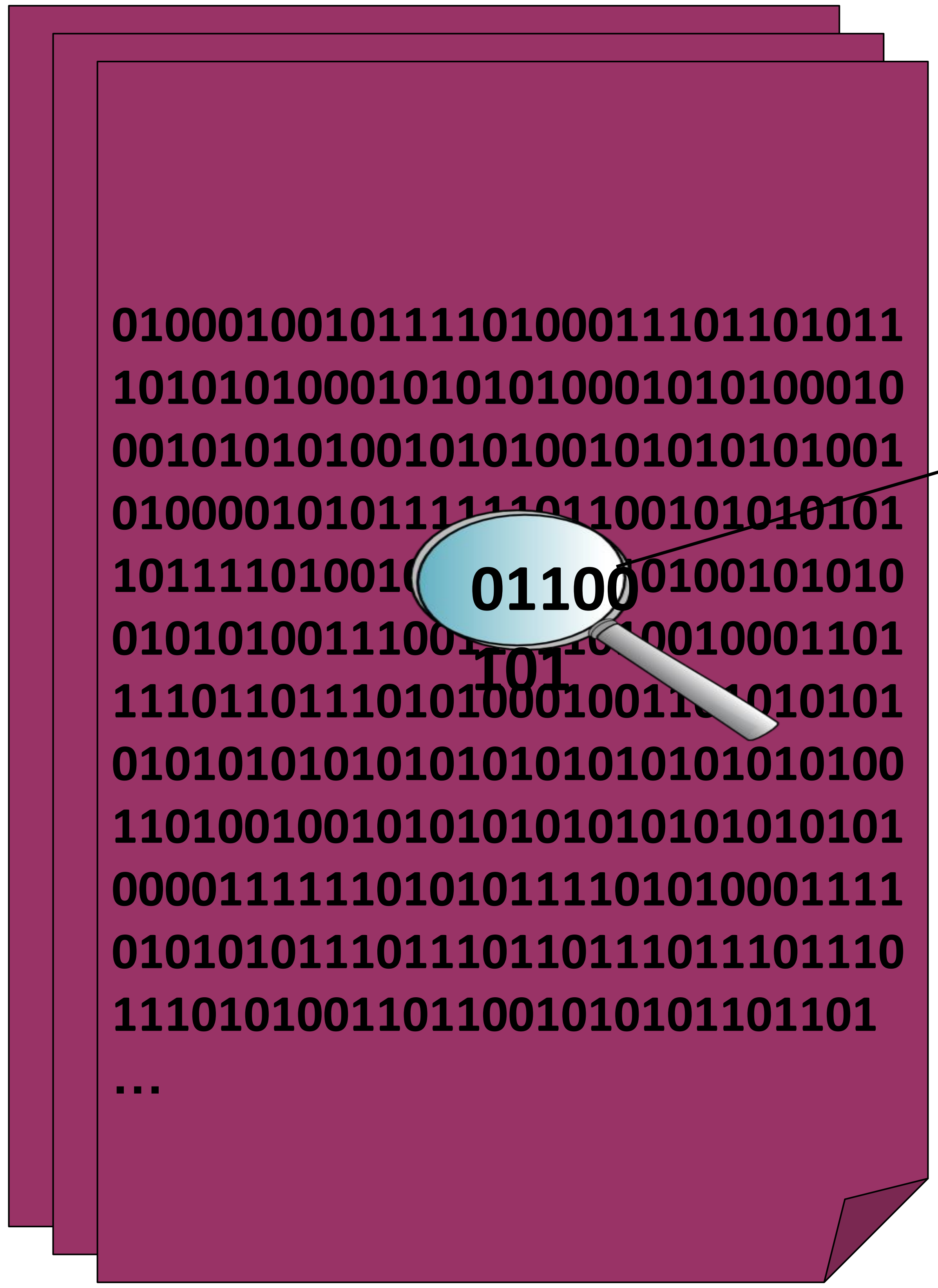
Counter Evaluation

- TSVC Benchmarks : TestSuite for Vectorizing Compilers
  - 208 function-compiler pairs tested
  - **175 function-compiler pairs pass**

Counter Evaluation

- TSVC Benchmarks : TestSuite for Vectorizing Compilers
  - 208 function-compiler pairs tested
  - **175 function-compiler pairs pass**
- LORE Repository for Loop Nests
  - **27 different vectorizable loop patterns, all pass**
  - 16 with multiple potentially-nested loops
  - 6 where multiple control flow paths in the loop body
  - 17 use multi-dimensional arrays

Peephole Superoptimization
Step 1



Harvest instruction sequences that can potentially be optimized. Canonicalize and store them.

```
mov %eax, %ecx
mov %ecx, %eax
```

```
sub $123, %eax
add $456, %eax
```

```
movl (%eax), %ecx
inc %ecx
movl %ecx, (%eax)
```

...

Target Sequences

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

Peephole Superoptimization
Step 2

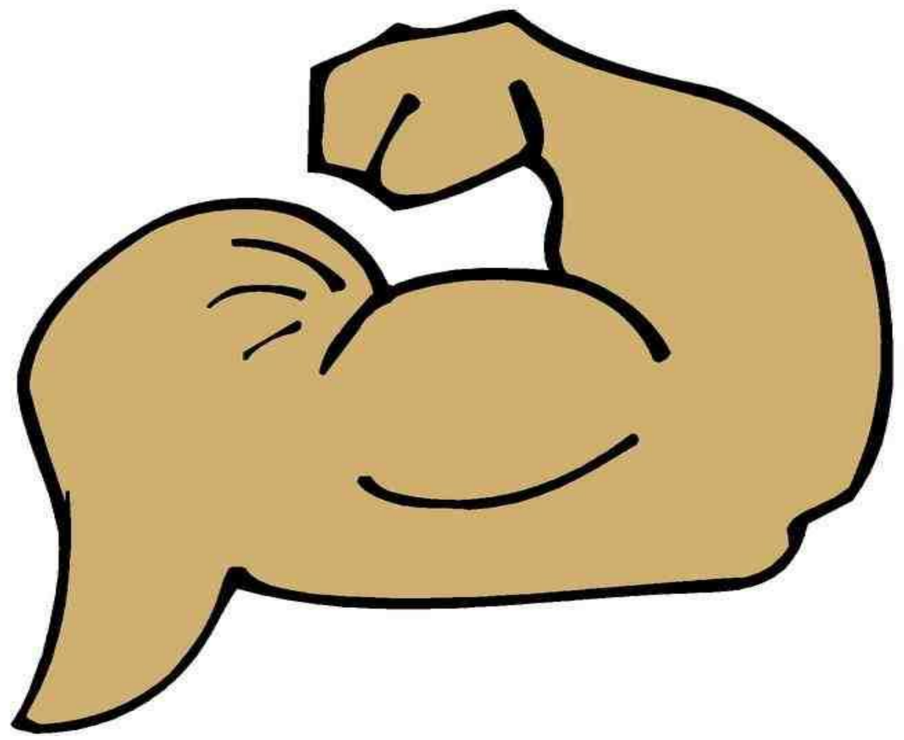
```
mov %eax, %ecx
mov %ecx, %eax

sub $123, %eax
add $456, %eax

movl (%eax), %ecx
inc %ecx
movl %ecx, (%eax)

...
```

Target Sequences



Superoptimization

```
mov %eax, %ecx

add $333, %eax

inc (%eax)

...
```

Optimized Sequences

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

Peephole Superoptimization
Step 3

mov %eax, %ecx mov %ecx, %eax	mov %eax, %ecx
sub $123, %eax add $456, %eax	add $333, %eax

movl (%eax), %ecx
inc %ecx
movl %ecx, (%eax)
	inc (%eax)

...	...

Table of Peephole Optimizations

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

Why is this nowhere near enough?

- Invariants are not captured, e.g., x=constant

pattern	replacement

//reg1 is known to be a constant C0
mov reg1, reg2
add $C1, reg2
	mov $(C0+C1), reg2

Why is this nowhere near enough?

- Local Memory Usage in the Pattern is *not* Modeled

pattern	replacement
char x[42]; … printf(x);	sub $42, %esp … printf(%esp)

Why is this nowhere near enough?

- Local Memory Usage in the Replacement is *not* Modeled

pattern

```
if (x == 0) {
  y = 42;
} else if (x == 1) {
  y = …;
} else if (x = …) {
  y = …
} …
```

replacement

```
y = PrecomputedArray[x];
```

[V. Kurhe et. al., Automatic Generation of Debug Headers, CGO 2022]

Why is this nowhere near enough?

- No preservation of debugging information

pattern	unoptimized	optimized

sum += a[i];
sum += a[i+1];
…
sum += a[i+7];
	addl a[i], reg
	addl a[i+1], reg
	…
	addl a[i+7], reg
		psubb %mm0, %mm0
		psadbw &a[i], %mm0
		movd %mm0, reg

Can correlate reg with sum after every instruction
Done by compiler developers

A superoptimized implementation erases this information.
Can reconstruct! [CGO22]

Serious?

- Aliasing information is not captured, e.g., heap access vs. stack access
- Invariants are not captured, e.g., x=constant
- Loops are not supported
- Local Memory Usage is not supported

Serious?

- Aliasing information is not captured, e.g., heap access vs. stack access
- Invariants are not captured, e.g., x=constant
- Loops are not supported
- Local Memory Usage is not supported

Invariant Sketching