Taming Pointers

A Symbolic Approach

Jianwen Zhu

jzhu@eecg.toronto.edu

Electrical and Computer Engineering

University of Toronto

Outline

- An Old Problem
- A New Strategy
- Taming Context Sensitivity
- Result and Conclusion

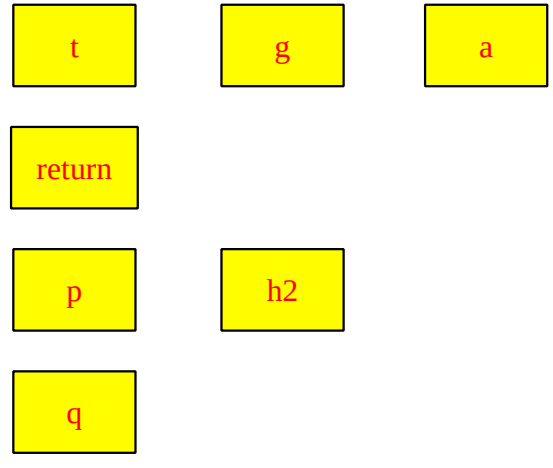
What is Pointer Analysis

```
char    *g, a, h1, h2;                    1
void main() {                             2
   char   *p, *q;                         3
S0: alloc( &p, &h1 );                     4
S1: p = getg( &q );                       5
S2: g = &a;                               6
   }                                      7
                                          8
char* getg( char** r ) {                  9
   char  **t;                            10
S0: t = &g;                              11
   if( g == NULL )                       12
S1:      alloc( t, &h2 );                13
S2: *r = *t;                             14
S3: return *r;                           15
   }                                     16
                                         17
void alloc( char** f, char* h ) {        18
S0: *f = h;                              19
   }                                     20
```

What is Pointer Analysis

```
char    *g, a, h1, h2;                1
void main() {                         2
   char   *p, *q;                     3
S0: alloc( &p, &h1 );                 4
S1: p = getg( &q );                   5
S2: g = &a;                           6
   }                                  7
                                      8
char* getg( char** r ) {              9
   char  **t;                        10
S0: t = &g;                          11
   if( g == NULL )                   12
S1:      alloc( t, &h2 );            13
S2: *r = *t;                         14
S3: return *r;                       15
   }                                 16
                                     17
void alloc( char** f, char* h ) {    18
S0: *f = h;                          19
   }                                 20
```

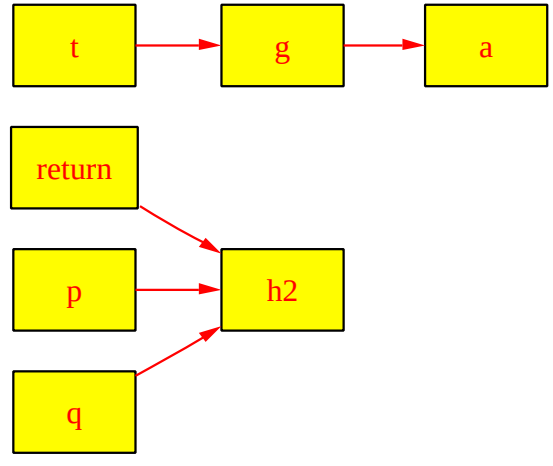
- Program variables
  - Named: globals/locals
  - Anonymous: return/malloc



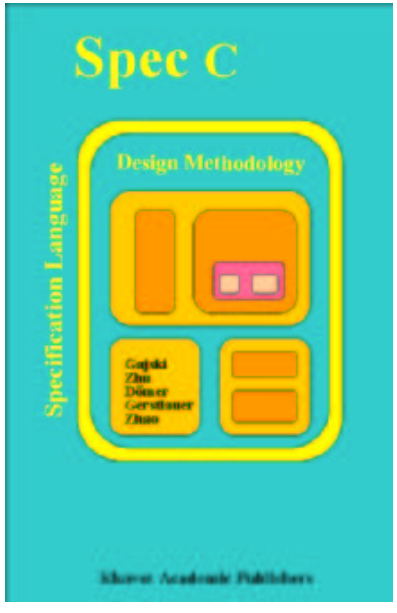
What is Pointer Analysis

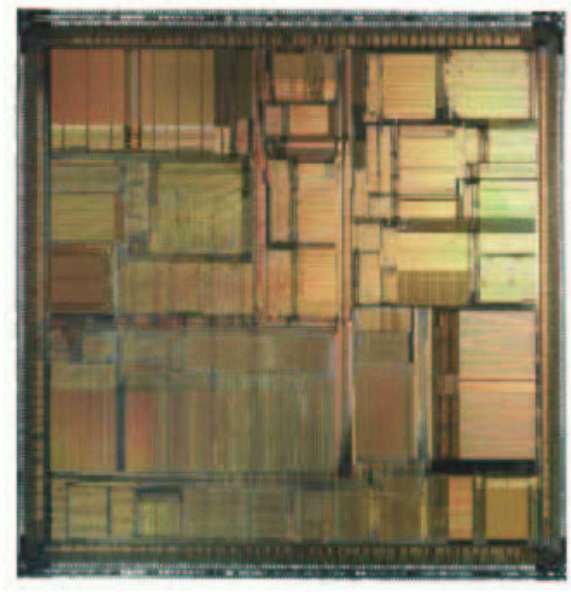
```
char    *g, a, h1, h2;                  1
void main() {                           2
   char   *p, *q;                       3
S0: alloc( &p, &h1 );                   4
S1: p = getg( &q );                     5
S2: g = &a;                             6
   }                                    7
                                        8
char* getg( char** r ) {                9
   char  **t;                           10
S0: t = &g;                             11
   if( g == NULL )                      12
S1:      alloc( t, &h2 );               13
S2: *r = *t;                            14
S3: return *r;                          15
   }                                    16
                                        17
void alloc( char** f, char* h ) {       18
S0: *f = h;                             19
   }                                    20
```

- Program variables
  - Named: globals/locals
  - Anonymous: return/malloc
- Program state: Point-to graph

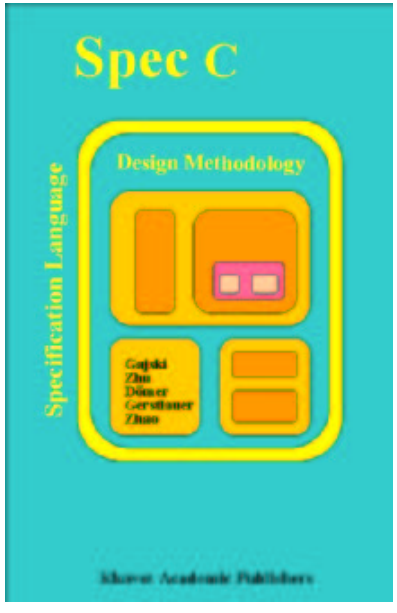


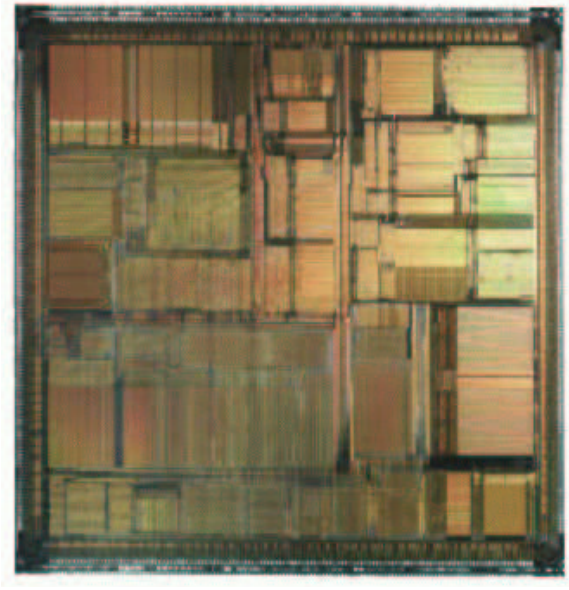
Why Do We Care: Silicon Compiler



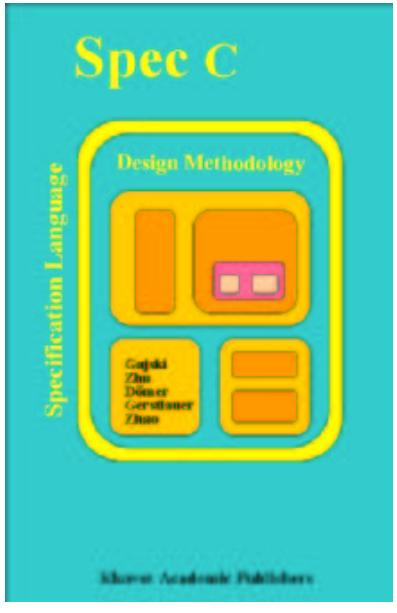


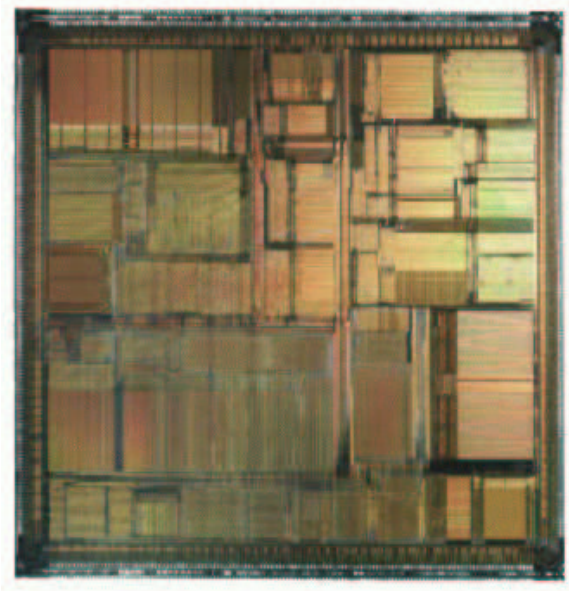
Why Do We Care: Silicon Compiler





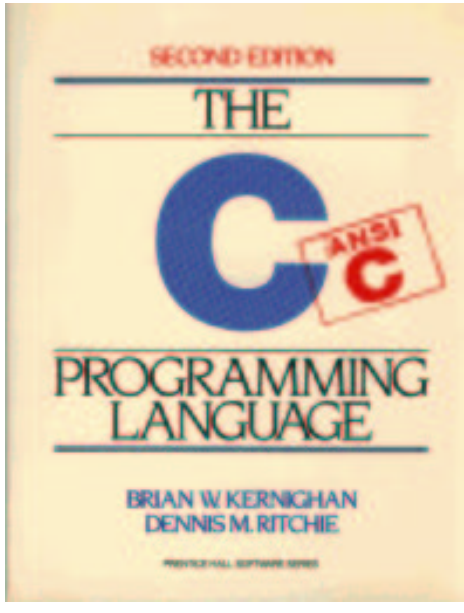
Why Do We Care: Silicon Compiler





- Fine-grained parallelism
  - Feed values to functional units
  - Dependency test
- Coarse-grained parallelism
  - Feed data structures to processors/cores
  - Disjoint data structure partition

Why Do We Care



- Fine-grained parallelism
  - Feed values to functional units
  - Dependency test
- Coarse-grained parallelism
  - Feed data structures to processors
  - Disjoint data structure partition

Precision Landscape

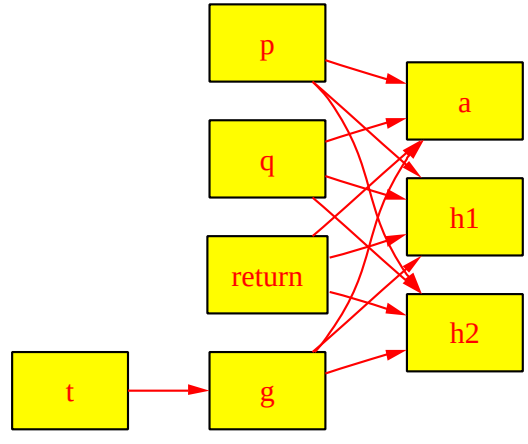
```
char    *g, a, h1, h2;                  1
void main() {                           2
   char   *p, *q;                       3
S0: alloc( &p, &h1 );                   4
S1: p = getg( &q );                     5
S2: g = &a;                             6
   }                                    7
                                        8
char* getg( char** r ) {                9
   char  **t;                           10
S0: t = &g;                             11
   if( g == NULL )                      12
S1:      alloc( t, &h2 );               13
S2: *r = *t;                            14
S3: return *r;                          15
   }                                    16
                                        17
void alloc( char** f, char* h ) {       18
S0: *f = h;                             19
   }                                    20
```

- Flow and Context Insensitive Analysis
- FICS Analysis
- FSCS Analysis

Precision Landscape

```
char    *g, a, h1, h2;                    1
void main() {                             2
   char   *p, *q;                         3
S0: alloc( &p, &h1 );                     4
S1: p = getg( &q );                       5
S2: g = &a;                               6
   }                                      7
                                          8
char* getg( char** r ) {                  9
   char  **t;                            10
S0: t = &g;                              11
   if( g == NULL )                       12
S1:      alloc( t, &h2 );                13
S2: *r = *t;                             14
S3: return *r;                           15
   }                                     16
                                         17
void alloc( char** f, char* h ) {        18
S0: *f = h;                              19
   }                                     20
```

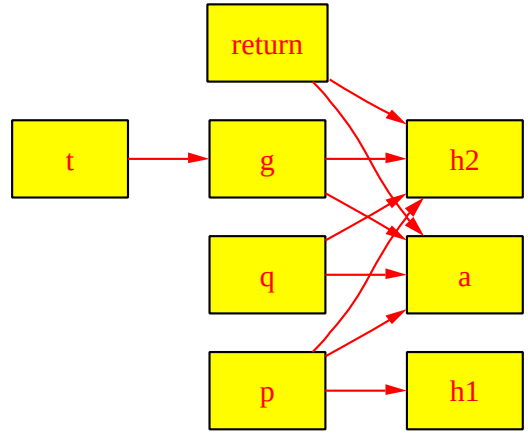
- Flow and Context Insensitive Analysis
- FICS Analysis
- FSCS Analysis



Precision Landscape

```
char    *g, a, h1, h2;                    1
void main() {                             2
   char   *p, *q;                         3
S0: alloc( &p, &h1 );                     4
S1: p = getg( &q );                       5
S2: g = &a;                               6
   }                                      7
                                          8
char* getg( char** r ) {                  9
   char  **t;                            10
S0: t = &g;                              11
   if( g == NULL )                       12
S1:      alloc( t, &h2 );                13
S2: *r = *t;                             14
S3: return *r;                           15
   }                                     16
                                         17
void alloc( char** f, char* h ) {        18
S0: *f = h;                              19
   }                                     20
```

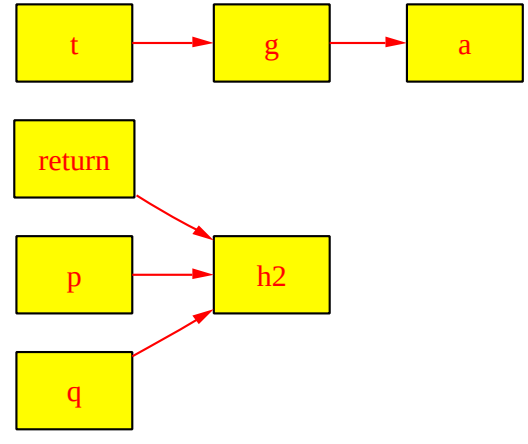
- Flow and Context Insensitive Analysis
- FICS Analysis
- FSCS Analysis



Precision Landscape

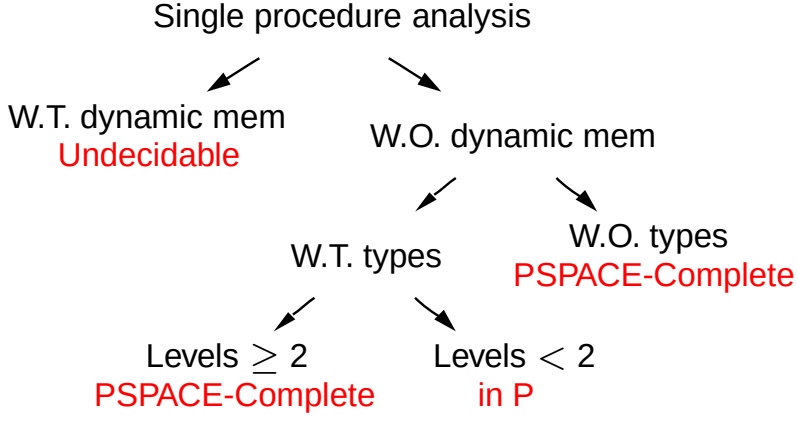
```
char    *g, a, h1, h2;                    1
void main() {                             2
   char   *p, *q;                         3
S0: alloc( &p, &h1 );                     4
S1: p = getg( &q );                       5
S2: g = &a;                               6
   }                                      7
                                          8
char* getg( char** r ) {                  9
   char  **t;                            10
S0: t = &g;                              11
   if( g == NULL )                       12
S1:      alloc( t, &h2 );                13
S2: *r = *t;                             14
S3: return *r;                           15
   }                                     16
                                         17
void alloc( char** f, char* h ) {        18
S0: *f = h;                              19
   }                                     20
```

- Flow and Context Insensitive Analysis
- FICS Analysis
- FSCS Analysis



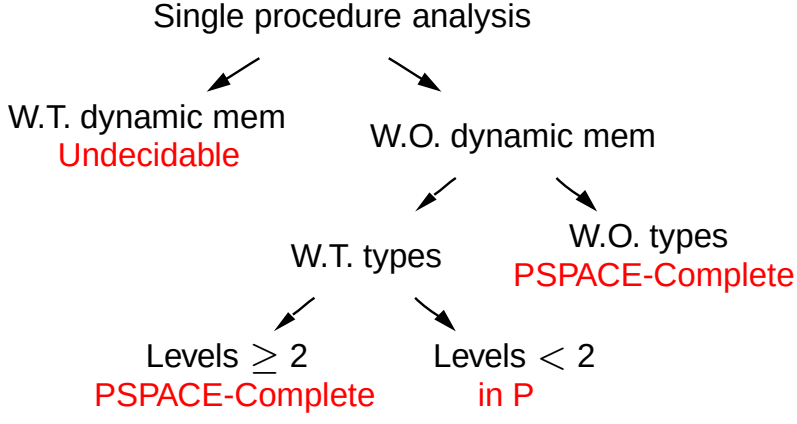
Why Pointer Analysis is Hard

- Flow-sensitive analysis
  - Theory: Chakaravarthy POPL'03
  - Practice: program state at every program point!
- Context-sensitive analysis
  - Evil of path



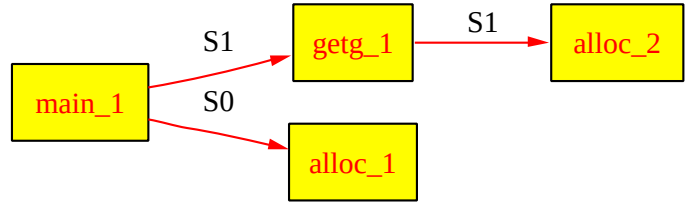
Why Pointer Analysis is Hard

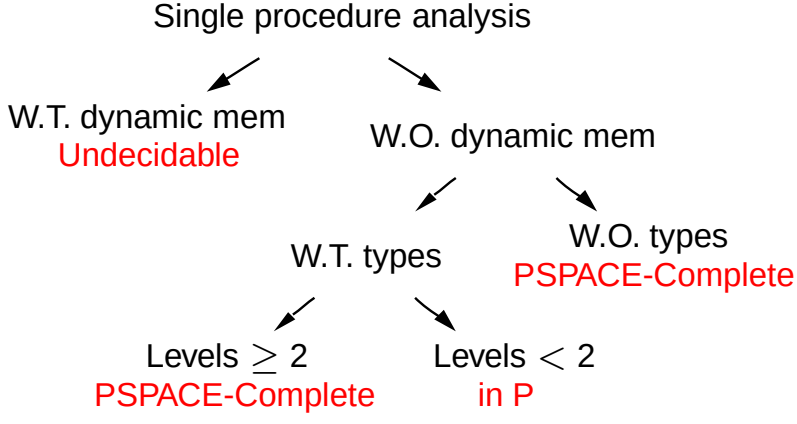
- Flow-sensitive analysis
  - Theory: Chakaravarthy POPL'03
  - Practice: program state at every program point!
- Context-sensitive analysis
  - Evil of path



Why Pointer Analysis is Hard

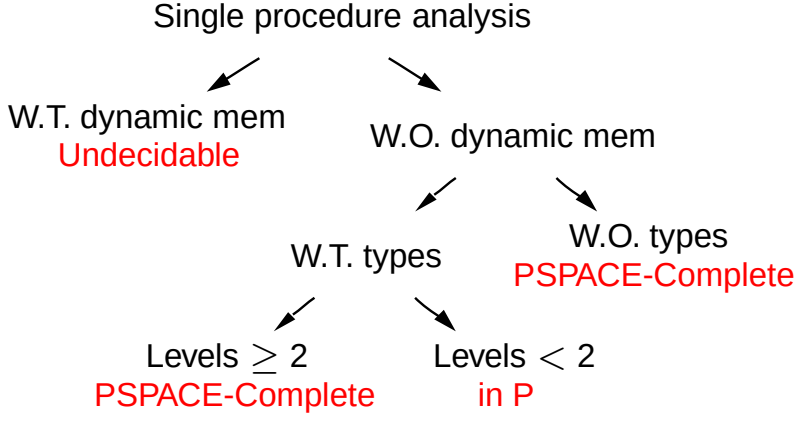
- Flow-sensitive analysis
  - Theory: Chakaravarthy POPL'03
  - Practice: program state at every program point!
- Context-sensitive analysis
  - Evil of path



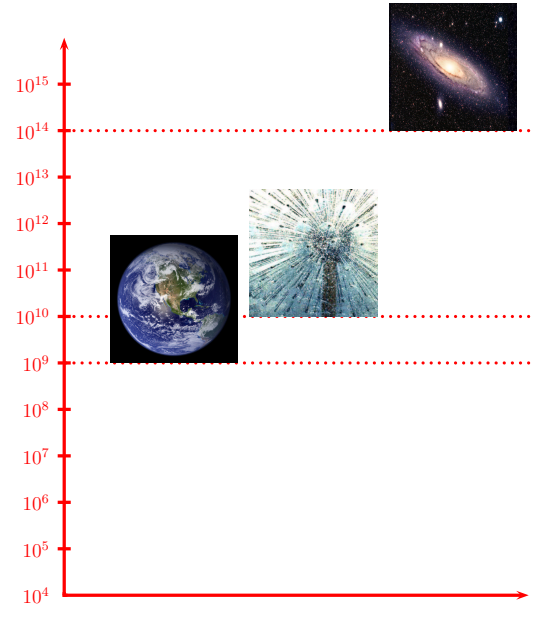


Why Pointer Analysis is Hard

- Flow-sensitive analysis
  - Theory: Chakaravarthy POPL'03
  - Practice: program state at every program point!



- Context-sensitive analysis
  - Evil of path



Outline

- An Old Problem
- A New Strategy
- Taming Context Sensitivity
- Result and Conclusion

Representing Set

- Discrete set isomorphic to integer set
- Given a finite discrete domain D
  - Sufficient to consider $D = [0, n-1]$
- Consider a set S in domain D: $S \subseteq D$
- Characteristic function $\lambda_S : D \mapsto \{0, 1\}$

$$\lambda_S(i) = \begin{cases} 0, & \text{if } i \notin S \\ 1, & \text{if } i \in S \end{cases}$$

Binary Decision Diagram (BDD)

x_2	x_1	x_0	λ_S
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

- $D = [0, 7]$
- $S = \{3, 5, 7\}$
- $\lambda_S = \overline{x_2} x_1 x_0 + x_2 \overline{x_1} x_0 + x_2 x_1 x_0$

Binary Decision Diagram (BDD)

x_2	x_1	x_0	λ_S
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

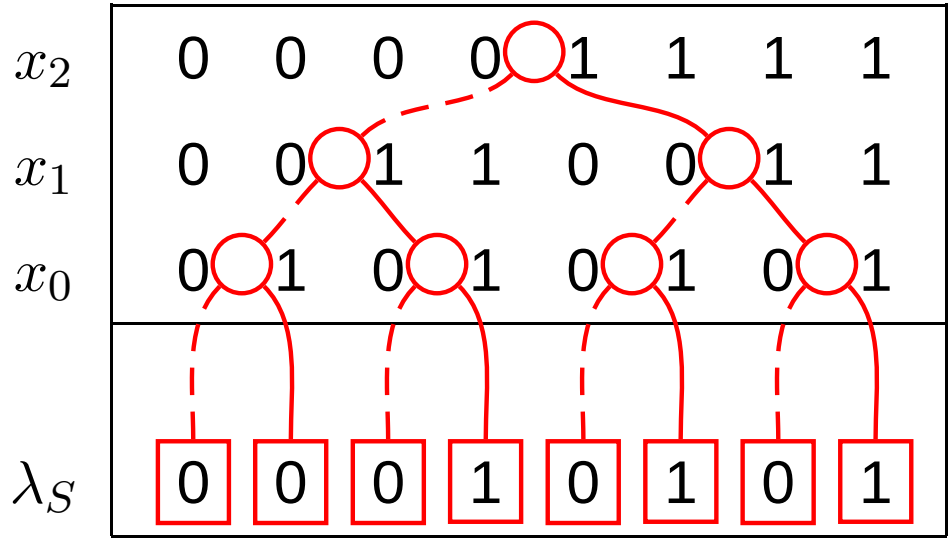
- $D = [0, 7]$
- $S = \{3, 5, 7\}$
- $\lambda_S = \overline{x_2} x_1 x_0 + x_2 \overline{x_1} x_0 + x_2 x_1 x_0$

x_2	0	0	0	0	1	1	1	1
x_1	0	0	1	1	0	0	1	1
x_0	0	1	0	1	0	1	0	1
λ_S	0	0	0	1	0	1	0	1

Binary Decision Diagram (BDD)

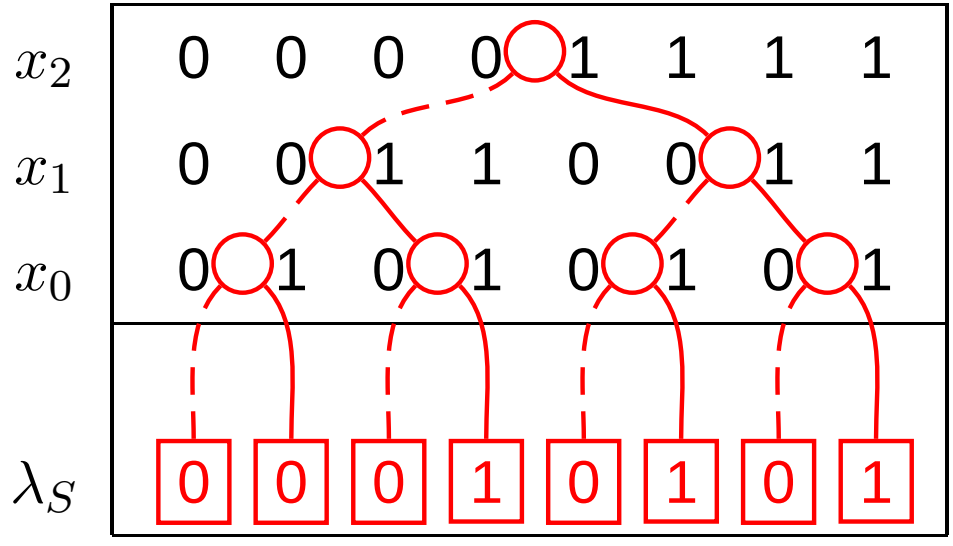
x_2	x_1	x_0	λ_S
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

- $D = [0, 7]$
- $S = \{3, 5, 7\}$
- $\lambda_S = \overline{x_2} x_1 x_0 + x_2 \overline{x_1} x_0 + x_2 x_1 x_0$



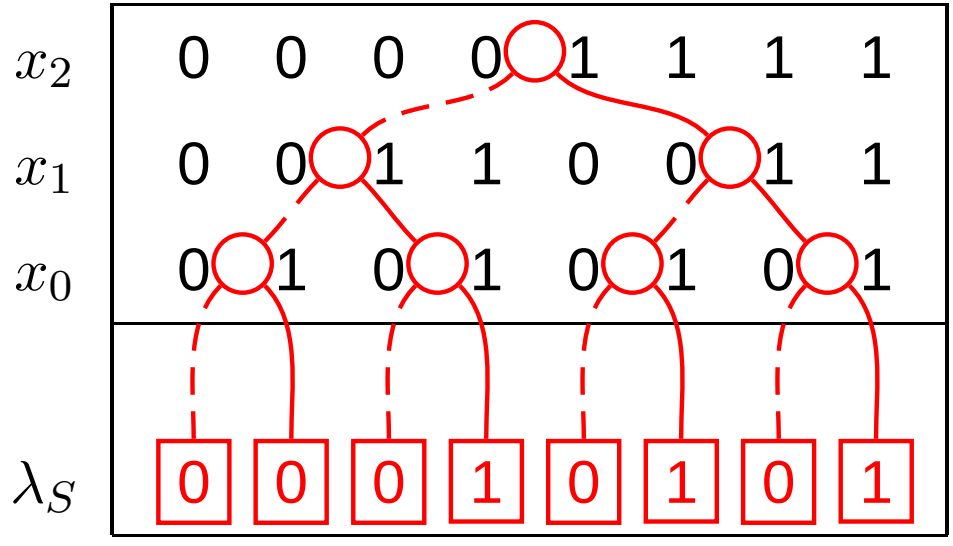
Reduced Ordered BDD (ROBDD)

- Reduction rules (Bryant'86)
  - var in fixed order



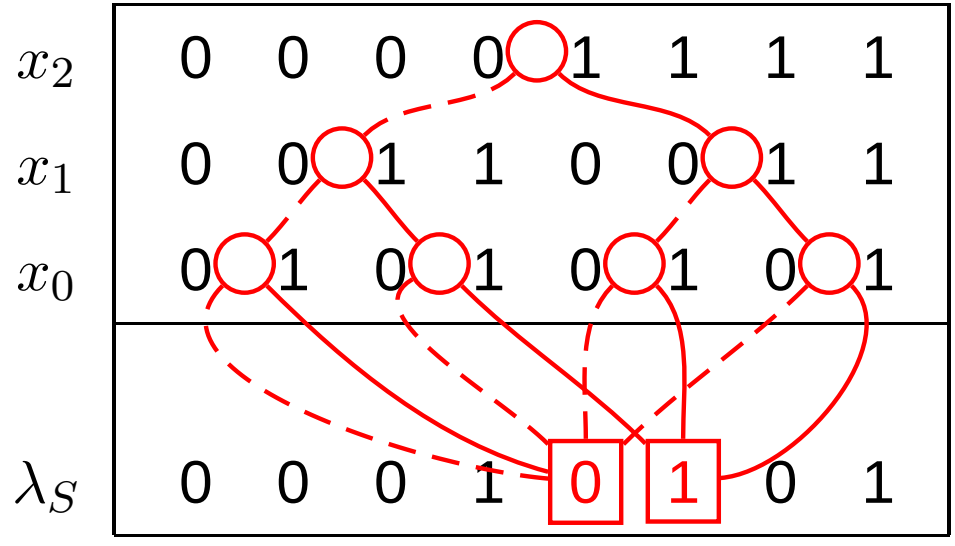
Reduced Ordered BDD (ROBDD)

- Reduction rules (Bryant'86)
  - var in fixed order
  - f.then = g.then, f.else = g.else



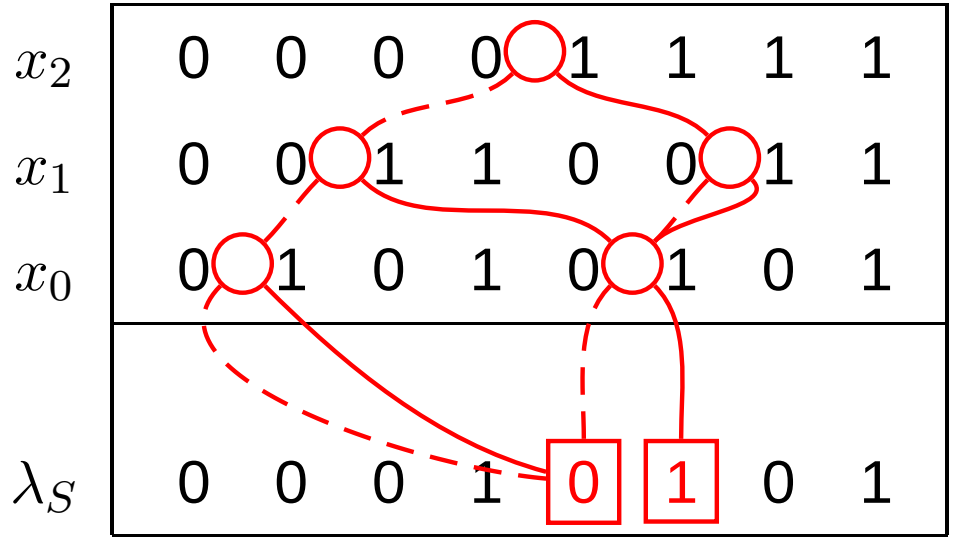
Reduced Ordered BDD (ROBDD)

- Reduction rules (Bryant'86)
  - var in fixed order
  - f.then = g.then, f.else = g.else



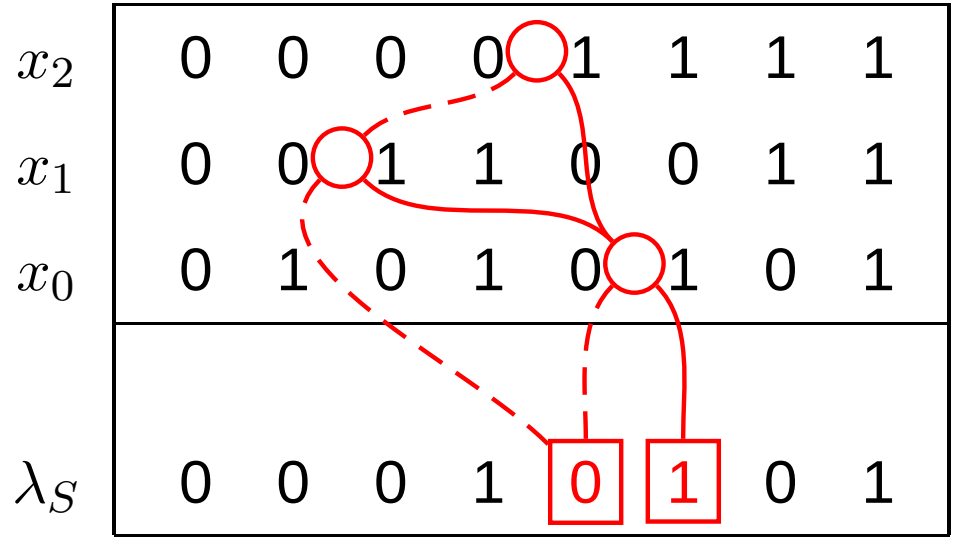
Reduced Ordered BDD (ROBDD)

- Reduction rules (Bryant'86)
  - var in fixed order
  - f.then = g.then, f.else = g.else
  - f.then = f.else



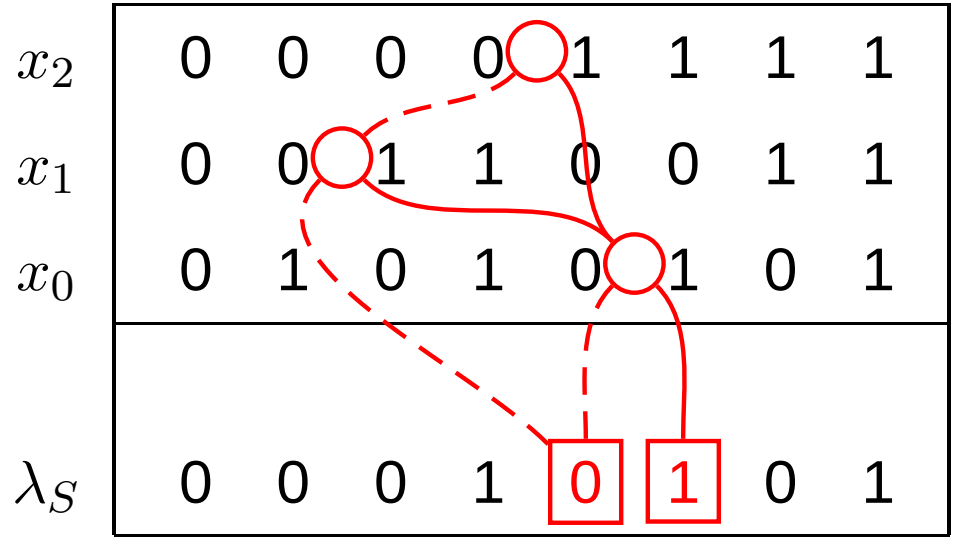
Reduced Ordered BDD (ROBDD)

- Reduction rules (Bryant'86)
  - var in fixed order
  - f.then = g.then, f.else = g.else
  - f.then = f.else



Reduced Ordered BDD (ROBDD)

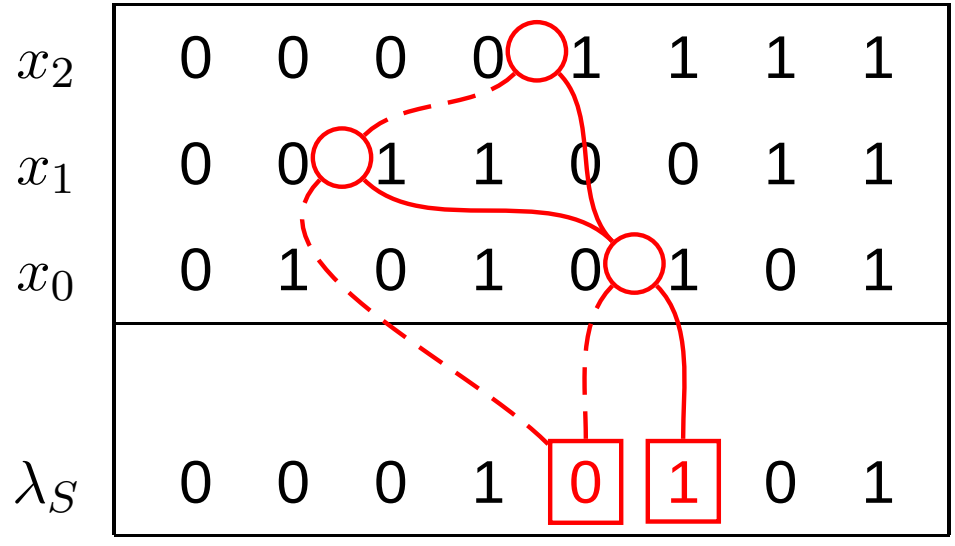
- Reduction rules (Bryant'86)
  - var in fixed order
  - f.then = g.then, f.else = g.else
  - f.then = f.else



- Counterintuitive implications

Reduced Ordered BDD (ROBDD)

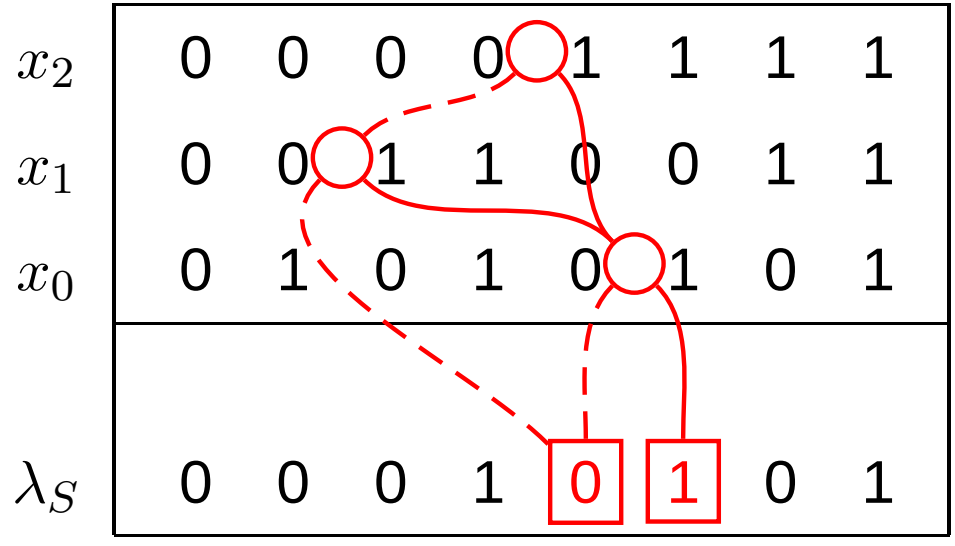
- Reduction rules (Bryant'86)
  - var in fixed order
  - f.then = g.then, f.else = g.else
  - f.then = f.else



- Counterintuitive implications
  - ~~Paths are evil!~~

Reduced Ordered BDD (ROBDD)

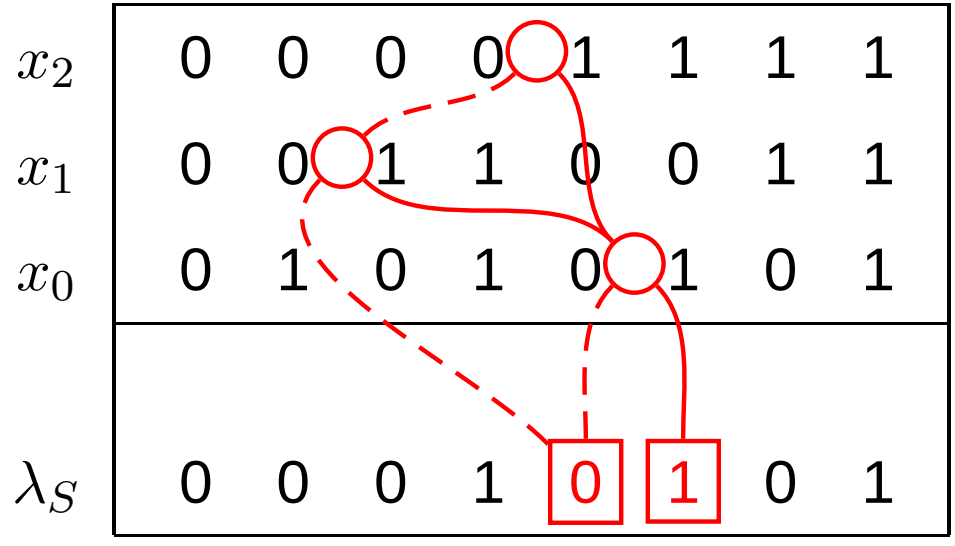
- Reduction rules (Bryant'86)
  - var in fixed order
  - f.then = g.then, f.else = g.else
  - f.then = f.else



- Counterintuitive implications
  - ~~Paths are evil!~~
  - ~~Size of a set $\propto$ number of its elements!~~

Reduced Ordered BDD (ROBDD)

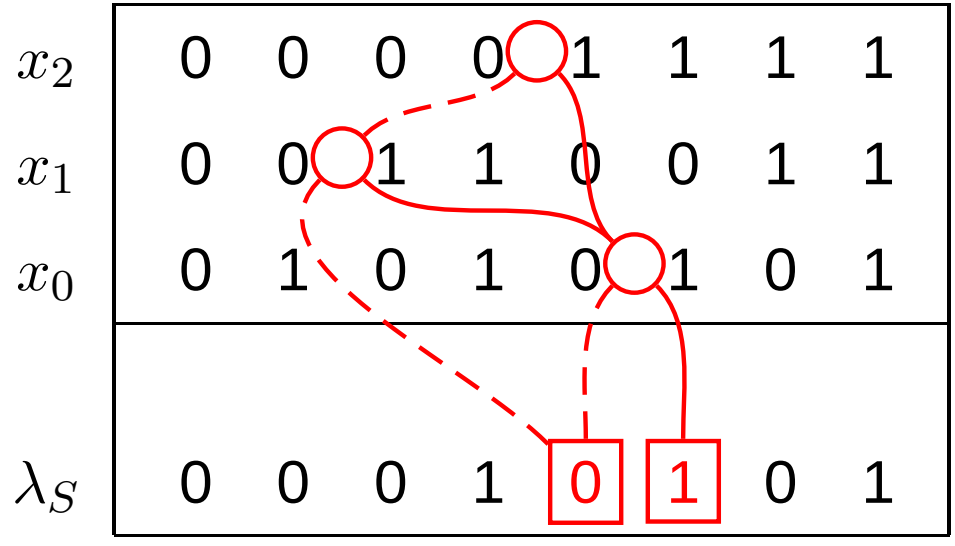
- Reduction rules (Bryant'86)
  - var in fixed order
  - f.then = g.then, f.else = g.else
  - f.then = f.else



- Counterintuitive implications
  - ~~Paths are evil!~~
  - ~~Size of a set $\propto$ number of its elements!~~

Reduced Ordered BDD (ROBDD)

- Reduction rules (Bryant'86)
  - var in fixed order
  - f.then = g.then, f.else = g.else
  - f.then = f.else



- Counterintuitive implications
  - ~~Paths are evil!~~
  - ~~Size of a set $\propto$ number of its elements!~~

Representing Relation

- Relation $R \subseteq A \times B$
  - Nothing but a set!
  - Characteristic function $\lambda_R : A \times B \mapsto \{0, 1\}$

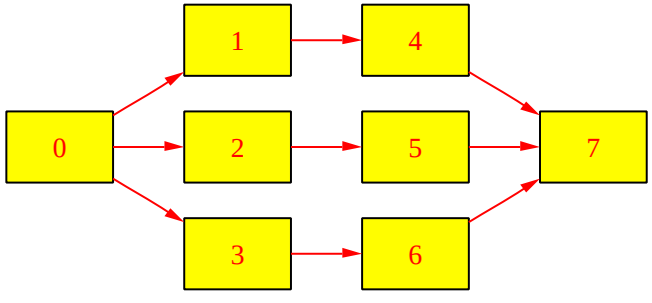
$$\lambda_R(i,j) = \begin{cases} 0, & \text{if } \langle i,j \rangle \notin R \\ 1, & \text{if } \langle i,j \rangle \in R \end{cases}$$

- Consider graph $\langle V, E \rangle$
  - Let V^0, V^1 be two Boolean spaces for V
  - Let V_i^0, V_j^1 be the ith, jth minterms of V^0, V^1
  - Product term $V_i^0 V_j^1$ represents an edge

Representing Relation

- Relation $R \subseteq A \times B$
  - Nothing but a set!
  - Characteristic function $\lambda_R : A \times B \mapsto \{0, 1\}$

$$\lambda_R(i,j) = \begin{cases} 0, & \text{if } \langle i,j \rangle \notin R \\ 1, & \text{if } \langle i,j \rangle \in R \end{cases}$$

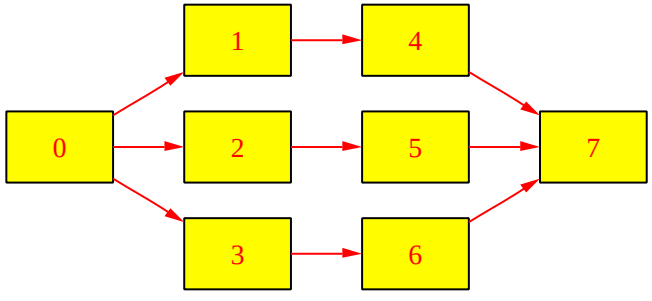


- Consider graph $\langle V, E \rangle$
  - Let V^0, V^1 be two Boolean spaces for V
  - Let V_i^0, V_j^1 be the ith, jth minterms of V^0, V^1
  - Product term $V_i^0 V_j^1$ represents an edge

Representing Relation

- Relation $R \subseteq A \times B$
  - Nothing but a set!
  - Characteristic function $\lambda_R : A \times B \mapsto \{0, 1\}$

$$\lambda_R(i,j) = \begin{cases} 0, & \text{if } \langle i,j \rangle \notin R \\ 1, & \text{if } \langle i,j \rangle \in R \end{cases}$$



- Consider graph $\langle V, E \rangle$
  - Let V^0, V^1 be two Boolean spaces for V
  - Let V_i^0, V_j^1 be the ith, jth minterms of V^0, V^1
  - Product term $V_i^0 V_j^1$ represents an edge

$$\begin{aligned} \lambda_E &= V_0^0 V_1^1 + V_0^0 V_2^1 + V_0^0 V_3^1 \\ &+ V_1^0 V_4^1 + V_2^0 V_5^1 + V_3^0 V_6^1 \\ &+ V_4^0 V_7^1 + V_5^0 V_7^1 + V_6^0 V_7^1 \end{aligned}$$

Symbolic Computation

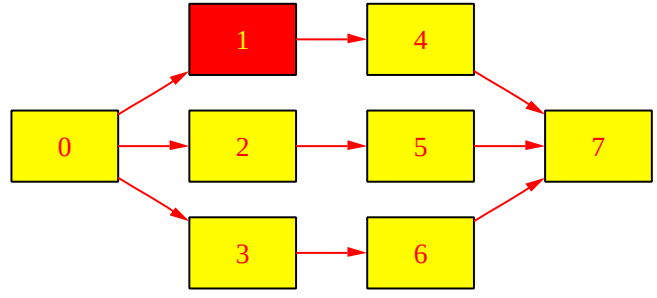
- First order logic in BDD
  - Negation: $\neg f$
  - Conjunction: $f \wedge g$
  - Disjunction: $f \vee g$
  - Existential abstraction: $\exists D^i . f$
  - Substitution: $f|_{D^i \mapsto D^j}$
  - Polynomial to BDD size
- Exercise: Finding successors

Symbolic Computation

- First order logic in BDD
  - Negation: $\neg f$
  - Conjunction: $f \wedge g$
  - Disjunction: $f \vee g$
  - Existential abstraction: $\exists D^i . f$
  - Substitution: $f|_{D^i \mapsto D^j}$
  - Polynomial to BDD size
- Exercise: Finding successors

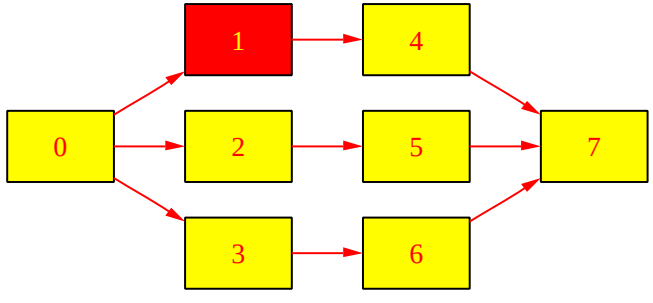
Symbolic Computation

- First order logic in BDD
  - Negation: $\neg f$
  - Conjunction: $f \wedge g$
  - Disjunction: $f \vee g$
  - Existential abstraction: $\exists D^i . f$
  - Substitution: $f|_{D^i \mapsto D^j}$
  - Polynomial to BDD size
- Exercise: Finding successors



Symbolic Computation

- First order logic in BDD
  - Negation: $\neg f$
  - Conjunction: $f \wedge g$
  - Disjunction: $f \vee g$
  - Existential abstraction: $\exists D^i . f$
  - Substitution: $f|_{D^i \mapsto D^j}$
  - Polynomial to BDD size

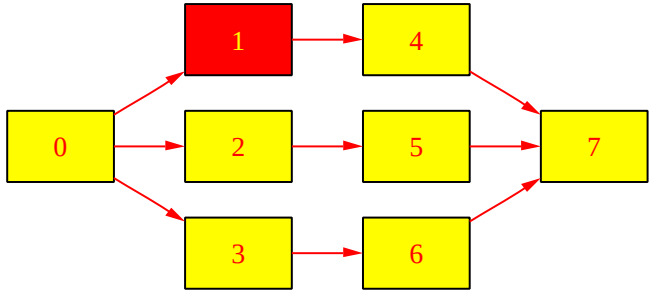


- Exercise: Finding successors
  - Given

$$
\begin{aligned}
r &= V_0^0 V_1^1 + V_0^0 V_2^1 + V_0^0 V_3^1 \\
&+ V_1^0 V_4^1 + V_2^0 V_5^1 + V_3^0 V_6^1 \\
&+ V_4^0 V_7^1 + V_5^0 V_7^1 + V_6^0 V_7^1 \\
s &= V_1^0
\end{aligned}
$$

Symbolic Computation

- First order logic in BDD
  - Negation: $\neg f$
  - Conjunction: $f \wedge g$
  - Disjunction: $f \vee g$
  - Existential abstraction: $\exists D^i . f$
  - Substitution: $f|_{D^i \mapsto D^j}$
  - Polynomial to BDD size



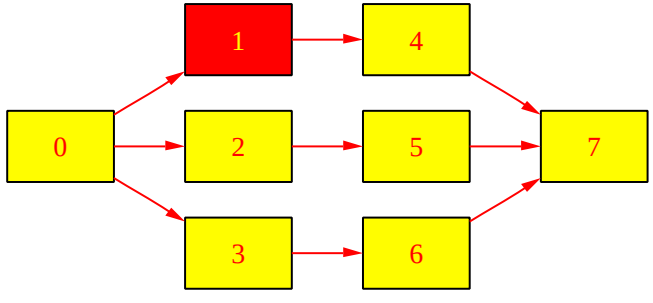
- Exercise: Finding successors
  - Given

$$\begin{aligned} r &= V_0^0 V_1^1 + V_0^0 V_2^1 + V_0^0 V_3^1 \\ &+ V_1^0 V_4^1 + V_2^0 V_5^1 + V_3^0 V_6^1 \\ &+ V_4^0 V_7^1 + V_5^0 V_7^1 + V_6^0 V_7^1 \\ s &= V_1^0 \end{aligned}$$

  - Step 1: $s \wedge r = V_1^0 V_4^1$

Symbolic Computation

- First order logic in BDD
  - Negation: $\neg f$
  - Conjunction: $f \wedge g$
  - Disjunction: $f \vee g$
  - Existential abstraction: $\exists D^i . f$
  - Substitution: $f|_{D^i \mapsto D^j}$
  - Polynomial to BDD size



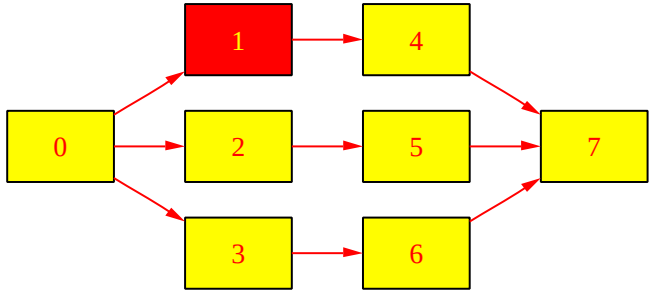
- Exercise: Finding successors
  - Given

$$\begin{aligned} r &= V_0^0 V_1^1 + V_0^0 V_2^1 + V_0^0 V_3^1 \\ &+ V_1^0 V_4^1 + V_2^0 V_5^1 + V_3^0 V_6^1 \\ &+ V_4^0 V_7^1 + V_5^0 V_7^1 + V_6^0 V_7^1 \\ s &= V_1^0 \end{aligned}$$

  - Step 1: $s \wedge r = V_1^0 V_4^1$
  - Step 2: $\exists V^0 . s \wedge r = V_4^1$

Symbolic Computation

- First order logic in BDD
  - Negation: $\neg f$
  - Conjunction: $f \wedge g$
  - Disjunction: $f \vee g$
  - Existential abstraction: $\exists D^i . f$
  - Substitution: $f|_{D^i \mapsto D^j}$
  - Polynomial to BDD size



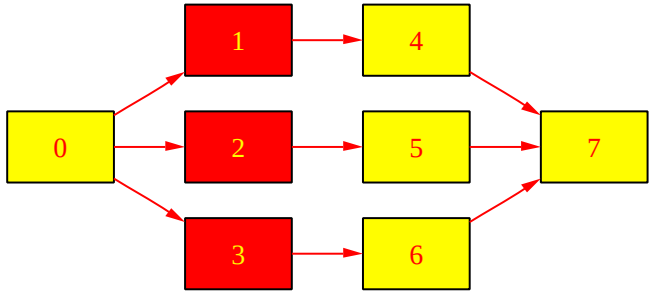
- Exercise: Finding successors
  - Given

$$\begin{aligned} r &= V_0^0 V_1^1 + V_0^0 V_2^1 + V_0^0 V_3^1 \\ &+ V_1^0 V_4^1 + V_2^0 V_5^1 + V_3^0 V_6^1 \\ &+ V_4^0 V_7^1 + V_5^0 V_7^1 + V_6^0 V_7^1 \\ s &= V_1^0 \end{aligned}$$

  - Step 1: $s \wedge r = V_1^0 V_4^1$
  - Step 2: $\exists V^0 . s \wedge r = V_4^1$
  - Step 3: $[\exists V^0 . s \wedge r]|_{V^1 \mapsto V^0} = V_4^0$

Image Computation

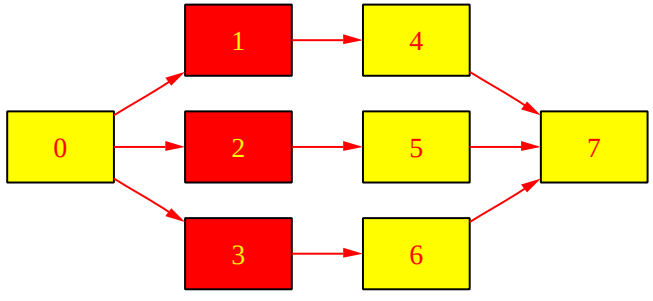
- s can be just a set
- Image (output) of a set (input) under a function



- Exercise again

Image Computation

- s can be just a set
- Image (output) of a set (input) under a function

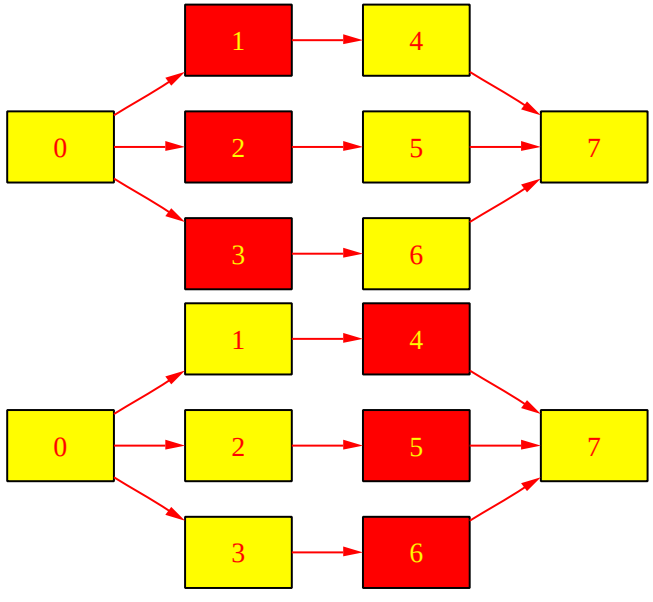


- Exercise again
  - Given

$$
\begin{aligned}
r &= V_0^0 V_1^1 + V_0^0 V_2^1 + V_0^0 V_3^1 \\
&+ V_1^0 V_4^1 + V_2^0 V_5^1 + V_3^0 V_6^1 \\
&+ V_4^0 V_7^1 + V_5^0 V_7^1 + V_6^0 V_7^1 \\
s &= V_1^0 + V_2^0 + V_3^0
\end{aligned}
$$

Image Computation

- s can be just a set
- Image (output) of a set (input) under a function



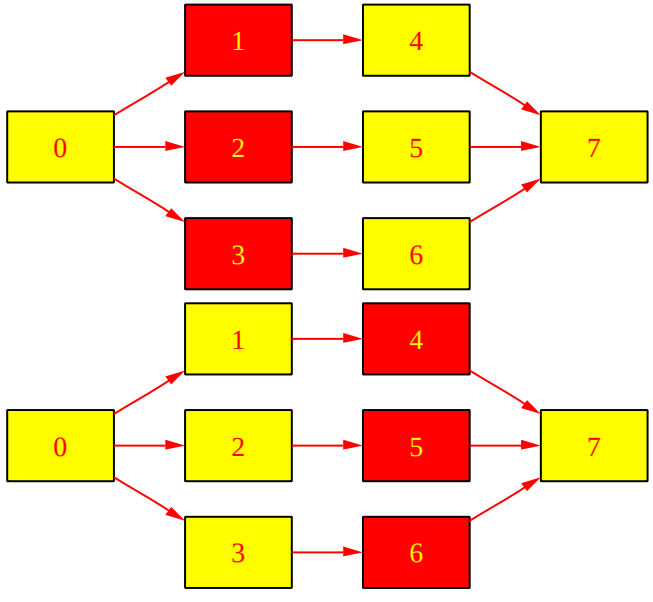
- Exercise again
  - Given

$$\begin{aligned} r &= V_0^0 V_1^1 + V_0^0 V_2^1 + V_0^0 V_3^1 \\ &+ V_1^0 V_4^1 + V_2^0 V_5^1 + V_3^0 V_6^1 \\ &+ V_4^0 V_7^1 + V_5^0 V_7^1 + V_6^0 V_7^1 \\ s &= V_1^0 + V_2^0 + V_3^0 \end{aligned}$$

  - Step 1: $s \wedge r = V_1^0 V_4^1 + V_2^0 V_5^1 + V_3^0 V_6^1$
  - Step 2: $\exists V^0 . s \wedge r = V_4^1 + V_5^1 + V_6^1$
  - Step 3: $[\exists V^0 . s \wedge r]|_{V^1 \mapsto V^0} = V_4^0 + V_5^0 + V_6^0$

Image Computation

- s can be just a set
- Image (output) of a set (input) under a function



- Does not have to be slower!

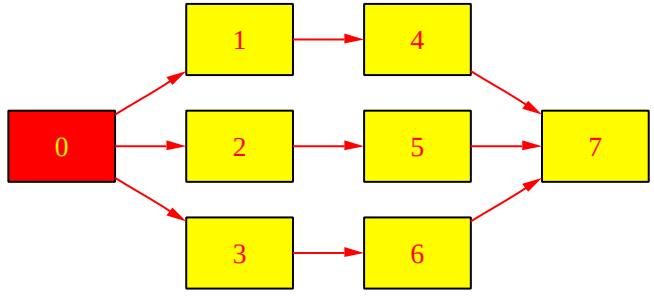
- Exercise again
  - Given

$$\begin{aligned} r &= V_0^0V_1^1 + V_0^0V_2^1 + V_0^0V_3^1 \\ &+ V_1^0V_4^1 + V_2^0V_5^1 + V_3^0V_6^1 \\ &+ V_4^0V_7^1 + V_5^0V_7^1 + V_6^0V_7^1 \\ s &= V_1^0 + V_2^0 + V_3^0 \end{aligned}$$

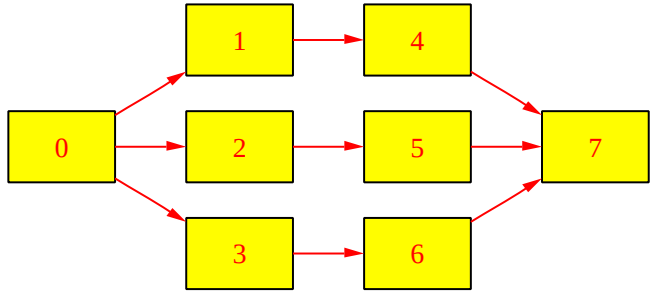
  - Step 1: $s \wedge r = V_1^0V_4^1 + V_2^0V_5^1 + V_3^0V_6^1$
  - Step 2: $\exists V^0.s \wedge r = V_4^1 + V_5^1 + V_6^1$
  - Step 3: $[\exists V^0.s \wedge r]|_{V^1 \mapsto V^0} = V_4^0 + V_5^0 + V_6^0$

Graph Reachability

- image



- reached

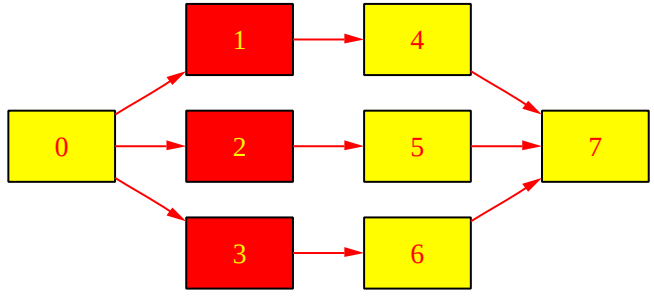


- Key for success of model checking
- Algorithm

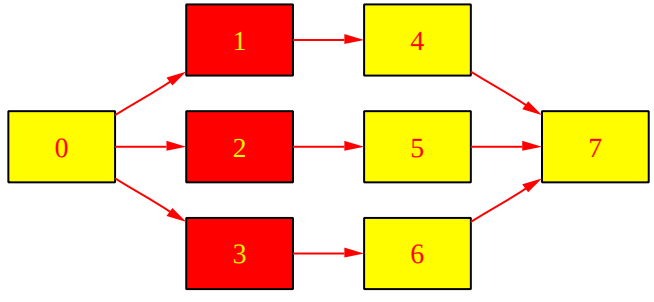
```
reachable( s : V^0, g : V^0 × V^1 )                    1
  : V^0 {                                              2
  var image, reached : V^0;                            3
  image = s;                                           4
  reached = 0;                                         5
  while( image ≠ 0 ) {                                 6
    image = [∃V^0.image ∧ g]|_{V^1 ↦ V^0};             7
    reached = reached ∨ image;                         8
    image = image ∧ ¬reached;                          9
    }                                                  10
  return reached ;                                     11
 }                                                     12
```

Graph Reachability

- image



- reached

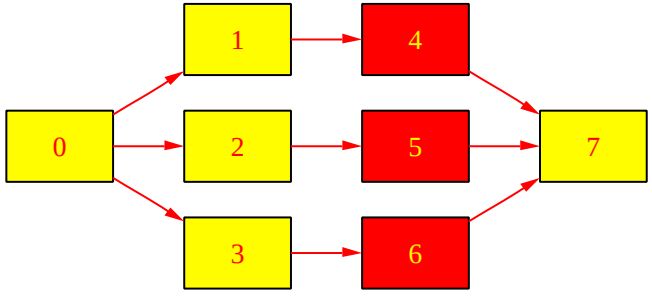


- Key for success of model checking
- Algorithm

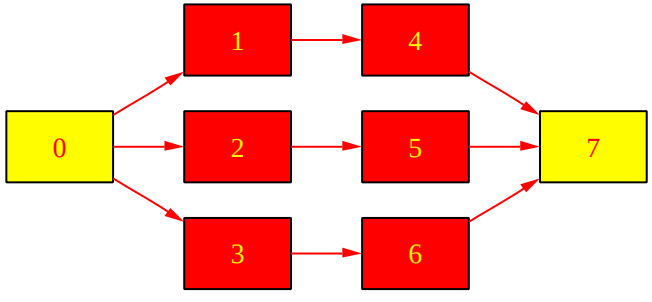
```
reachable( s : V^0, g : V^0 × V^1 )                    1
  : V^0 {                                              2
  var image, reached : V^0;                            3
  image = s;                                           4
  reached = 0;                                         5
  while( image ≠ 0 ) {                                 6
    image = [∃V^0.image ∧ g]|_{V^1 ↦ V^0};             7
    reached = reached ∨ image;                         8
    image = image ∧ ¬reached;                          9
    }                                                  10
  return reached ;                                     11
 }                                                     12
```

Graph Reachability

- image



- reached

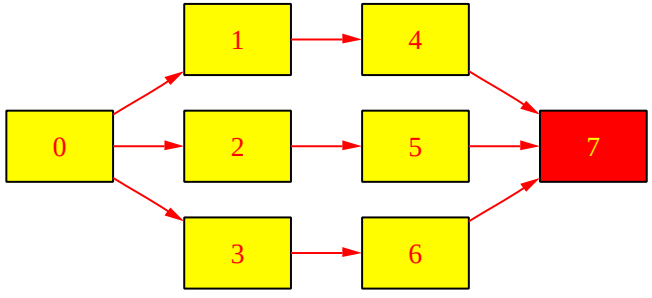


- Key for success of model checking
- Algorithm

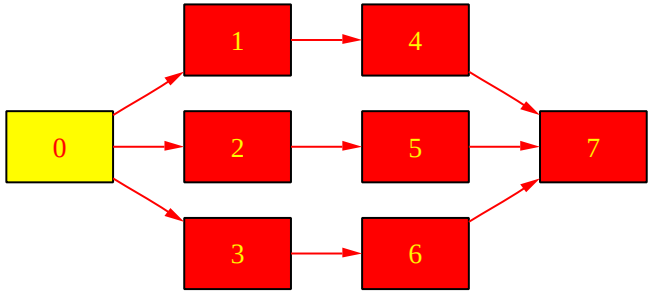
```
reachable( s : V^0, g : V^0 × V^1 )                    1
  : V^0 {                                              2
  var image, reached : V^0;                            3
  image = s;                                           4
  reached = 0;                                         5
  while( image ≠ 0 ) {                                 6
    image = [∃V^0.image ∧ g]|_{V^1 ↦ V^0};             7
    reached = reached ∨ image;                         8
    image = image ∧ ¬reached;                          9
    }                                                  10
  return reached ;                                     11
  }                                                    12
```

Graph Reachability

- image



- reached



- Key for success of model checking
- Algorithm

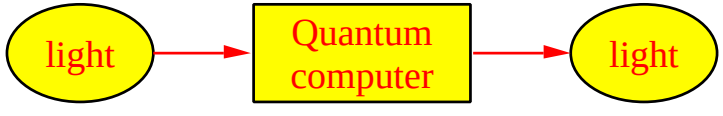
```
reachable( s : V^0, g : V^0 × V^1 )                    1
  : V^0 {                                              2
  var image, reached : V^0;                            3
  image = s;                                           4
  reached = 0;                                         5
  while( image ≠ 0 ) {                                 6
    image = [∃V^0.image ∧ g]|_{V^1 ↦ V^0};             7
    reached = reached ∨ image;                         8
    image = image ∧ ¬reached;                          9
    }                                                  10
  return reached ;                                     11
  }                                                    12
```

Superposed Computation

- Divide-and-Conquer paradigm
  - task $\mapsto$ many problem instances
- Parallel computing
  - One processor per instance
  - Exponential many instances?

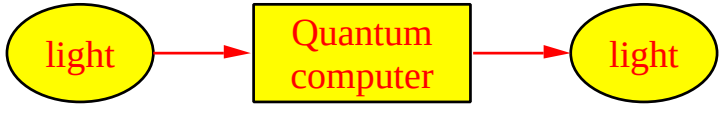
Superposed Computation

- Divide-and-Conquer paradigm
  - task $\mapsto$ many problem instances
- Parallel computing
  - One processor per instance
  - Exponential many instances?
- Quantum computing



Superposed Computation

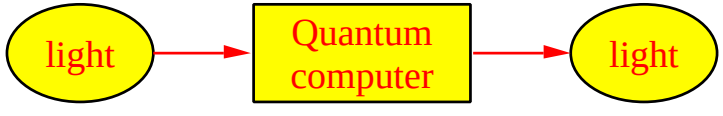
- Divide-and-Conquer paradigm
  - task $\mapsto$ many problem instances
- Parallel computing
  - One processor per instance
  - Exponential many instances?
- Quantum computing



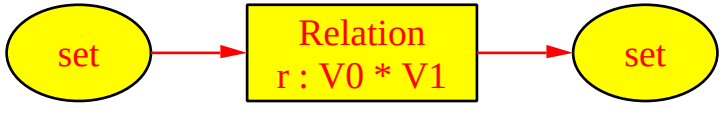
- Superposed computing

Superposed Computation

- Divide-and-Conquer paradigm
  - task $\mapsto$ many problem instances
- Parallel computing
  - One processor per instance
  - Exponential many instances?
- Quantum computing

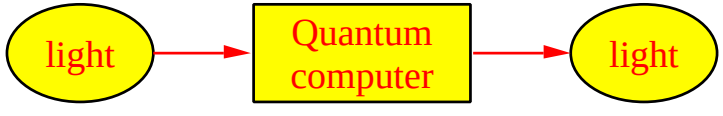


- Superposed computing

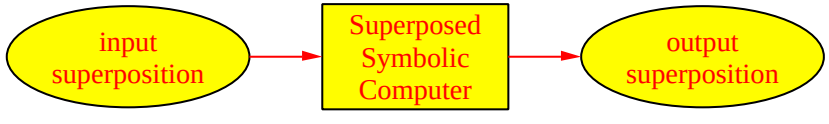


Superposed Computation

- Divide-and-Conquer paradigm
  - task $\mapsto$ many problem instances
- Parallel computing
  - One processor per instance
  - Exponential many instances?
- Quantum computing



- Superposed computing



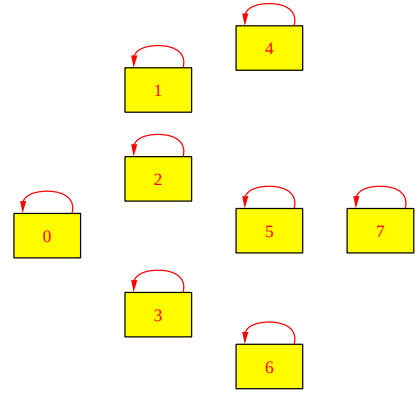
- Introduce domain of problem instances I
- Graph superposition: $I \times V^0 \times V^1$
- Input superposition: $I \times V^0$
- Superposed image computation
  - $s : I \times V^0, r : I \times V^0 \times V^1$
  - $s \wedge r : I \times V^0 \times V^1$
  - $\exists V^0 . s \wedge r : I \times V^1$
  - $[\exists V^0 . s \wedge r]|_{V^1 \mapsto V^0} : I \times V^0$

Transitive Closure

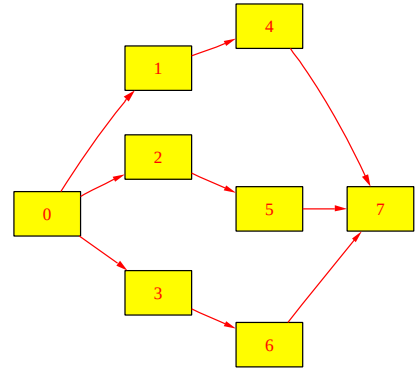
- Reachability problem for each node
- Algorithm

```
closure( g : V^0 × V^1 )                                   1
  : V^0 × V^1 {                                            2
  var image, reached : V^2 × V^0;                          3
  image = V^2 == V^0;                                      4
  reached = 0;                                             5
  while( image ≠ 0 ) {                                     6
    image = [∃V^0.image ∧ g]|_{V^1↦V^0};                   7
    reached = reached ∨ image;                             8
    image = image ∧ ¬reached;                              9
    }                                                      10
  return reached|_{V^2↦V^0, V^1↦V^1} ;                     11
}                                                          12
```

- image



- reached

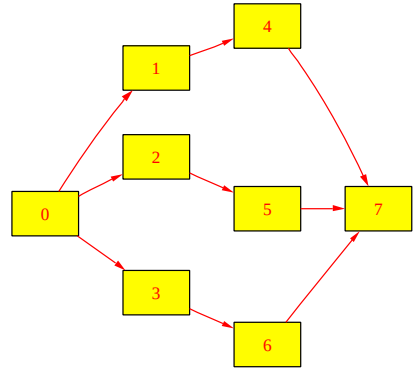


Transitive Closure

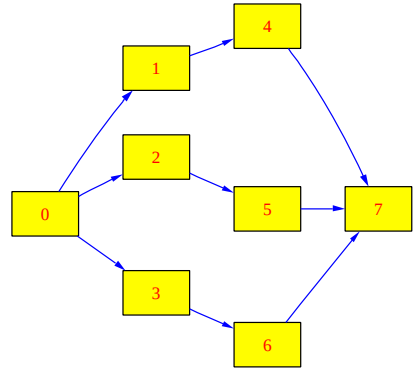
- Reachability problem for each node
- Algorithm

```
closure( g : V^0 × V^1 )                                   1
  : V^0 × V^1 {                                            2
  var image, reached : V^2 × V^0;                          3
  image = V^2 == V^0;                                      4
  reached = 0;                                             5
  while( image ≠ 0 ) {                                     6
    image = [∃V^0.image ∧ g]|_{V^1↦V^0};                   7
    reached = reached ∨ image;                             8
    image = image ∧ ¬reached;                              9
    }                                                      10
  return reached|_{V^2↦V^0, V^1↦V^1} ;                     11
}                                                          12
```

- image



- reached

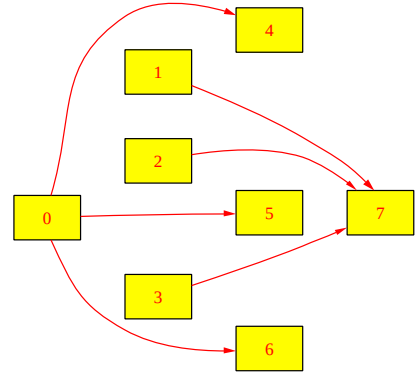


Transitive Closure

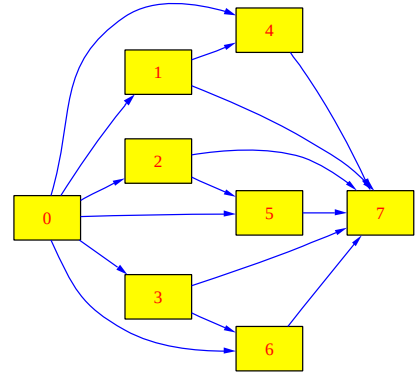
- Reachability problem for each node
- Algorithm

```
closure( g : V^0 × V^1 )                                   1
  : V^0 × V^1 {                                            2
  var image, reached : V^2 × V^0;                          3
  image = V^2 == V^0;                                      4
  reached = 0;                                             5
  while( image ≠ 0 ) {                                     6
    image = [∃V^0.image ∧ g]|_{V^1↦V^0};                   7
    reached = reached ∨ image;                             8
    image = image ∧ ¬reached;                              9
    }                                                      10
  return reached|_{V^2↦V^0, V^1↦V^1} ;                     11
}                                                          12
```

- image



- reached

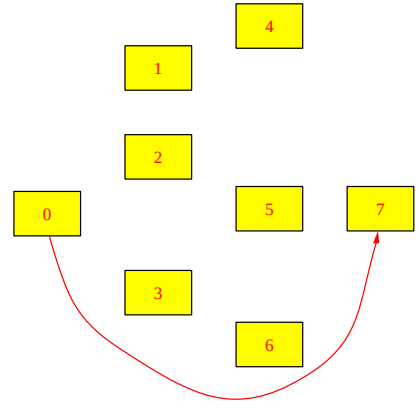


Transitive Closure

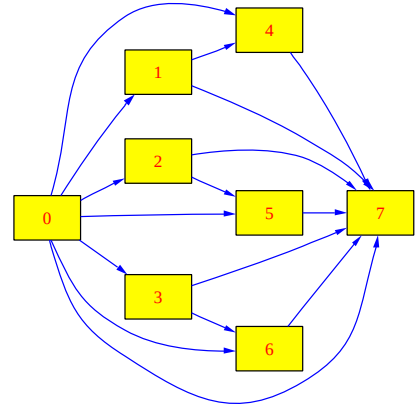
- Reachability problem for each node
- Algorithm

```
closure( g : V^0 × V^1 )                                   1
  : V^0 × V^1 {                                            2
  var image, reached : V^2 × V^0;                          3
  image = V^2 == V^0;                                      4
  reached = 0;                                             5
  while( image ≠ 0 ) {                                     6
    image = [∃V^0.image ∧ g]|_{V^1↦V^0};                   7
    reached = reached ∨ image;                             8
    image = image ∧ ¬reached;                              9
    }                                                      10
  return reached|_{V^2↦V^0, V^1↦V^1} ;                     11
}                                                          12
```

- image



- reached



Outline

- An Old Problem
- A New Strategy

Outline

- An Old Problem
- A New Strategy
  - Any structured information (set, graph, relation) maybe be captured compactly by Boolean functions.

Outline

- An Old Problem
- A New Strategy
  - Any structured information (set, graph, relation) maybe be captured compactly by Boolean functions.
  - Different problem instances can be superposed and solved collectively by first order logic operators, which may lead to exponential reduction of runtime.

Outline

- An Old Problem
- A New Strategy
  - Any structured information (set, graph, relation) maybe be captured compactly by Boolean functions.
  - Different problem instances can be superposed and solved collectively by first order logic operators, which may lead to exponential reduction of runtime.
- Taming Context Sensitivity
- Result and Conclusion

Symbolic Framework (FICS)

- Given

$call$	:	$(M^0 \times S^0 \times C^0) \times (M^1 \times C^1)$
tr	:	$M^0 \times (B^2 \times P^0) \times (B^3 \times P^1)$
$path$	:	$P^0 \times F^0 \times P^1$
$inmap$	:	$M^0 \times S^0 \times (B^2 \times P^0) \times (B^3 \times P^1)$
$outmap$	:	$M^0 \times S^0 \times (B^2 \times P^0) \times (B^3 \times P^1)$

Symbolic Framework (FICS)

- Given

$$
\begin{array}{rcl}
call & : & (M^0 \times S^0 \times C^0) \times (M^1 \times C^1) \\
tr & : & M^0 \times (B^2 \times P^0) \times (B^3 \times P^1) \\
path & : & P^0 \times F^0 \times P^1 \\
inmap & : & M^0 \times S^0 \times (B^2 \times P^0) \times (B^3 \times P^1) \\
outmap & : & M^0 \times S^0 \times (B^2 \times P^0) \times (B^3 \times P^1)
\end{array}
$$

- Find

$$
\begin{array}{rcl}
s & : & B^0 \times F^0 \times B^1 \\
q & : & M^0 \times C^0 \times (B^2 \times P^0) \times B^0
\end{array}
$$

Symbolic Framework (FICS)

- Given

$$
\begin{array}{rcl}
call & : & (M^0 \times S^0 \times C^0) \times (M^1 \times C^1) \\
tr & : & M^0 \times (B^2 \times P^0) \times (B^3 \times P^1) \\
path & : & P^0 \times F^0 \times P^1 \\
inmap & : & M^0 \times S^0 \times (B^2 \times P^0) \times (B^3 \times P^1) \\
outmap & : & M^0 \times S^0 \times (B^2 \times P^0) \times (B^3 \times P^1)
\end{array}
$$

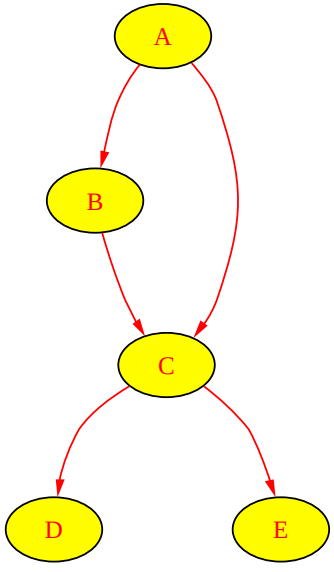
- Find

$$
\begin{array}{rcl}
s & : & B^0 \times F^0 \times B^1 \\
q & : & M^0 \times C^0 \times (B^2 \times P^0) \times B^0
\end{array}
$$

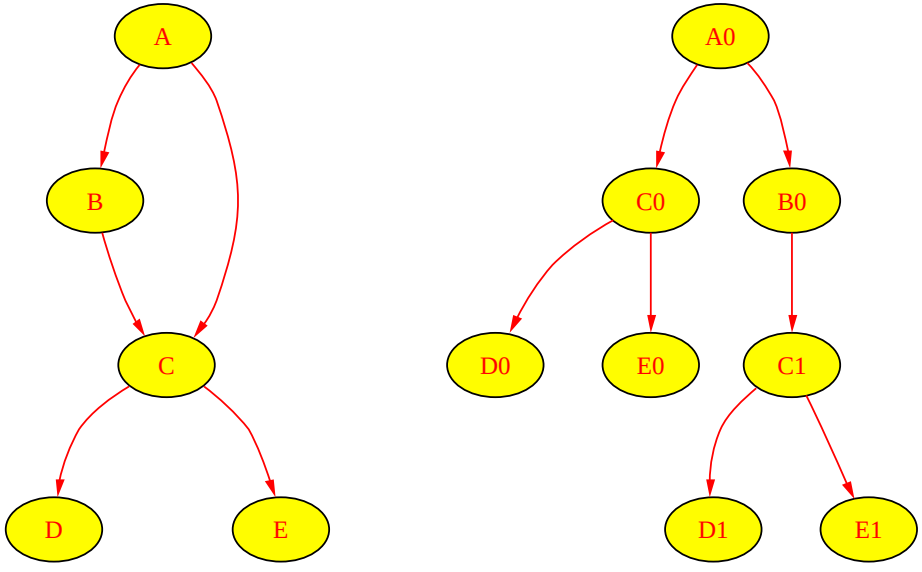
- By Recurrence equations

$$
\begin{array}{rcl}
s & = & \text{apply}(tr, q) \\
q & = & \text{bind}(\text{query}(s, path), inmap, outmap, call)
\end{array}
$$

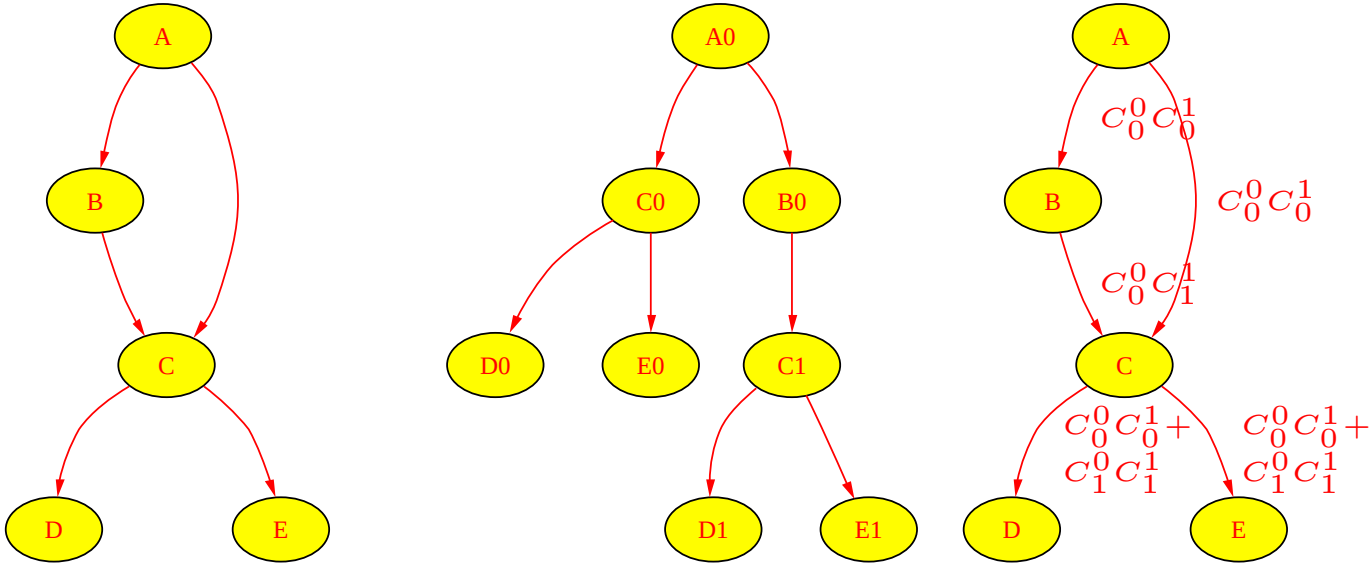
Symbolic Invocation Graph (SIG)



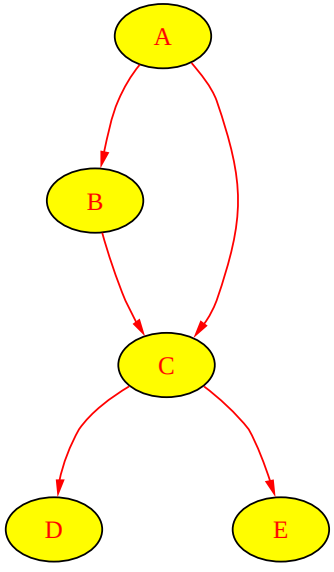
Symbolic Invocation Graph (SIG)

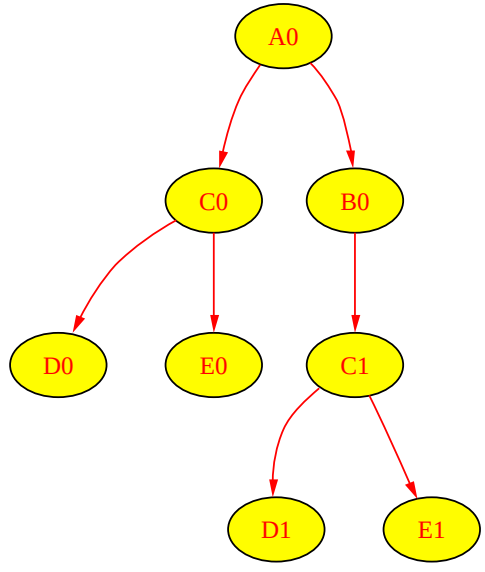


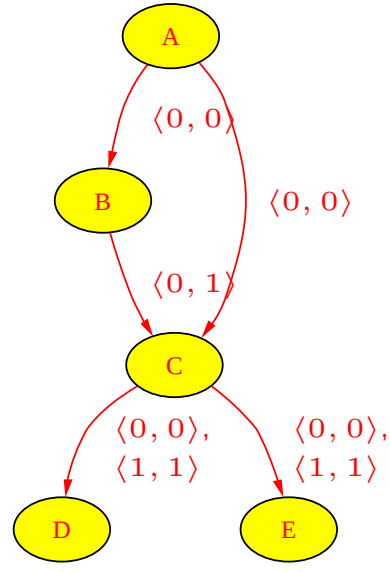
Symbolic Invocation Graph (SIG)



Symbolic Invocation Graph (SIG)

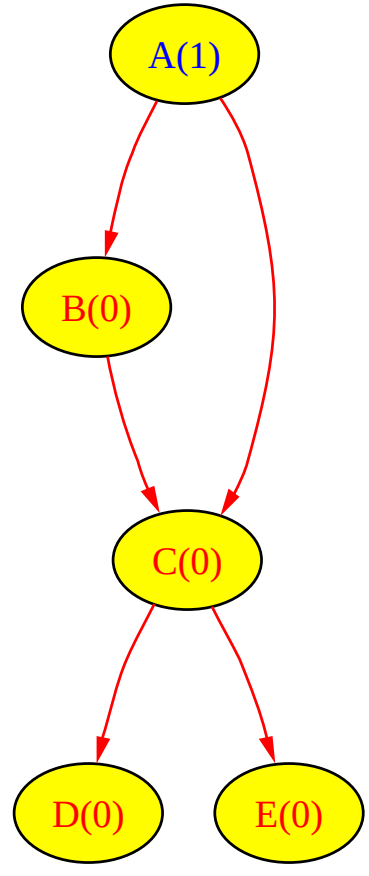






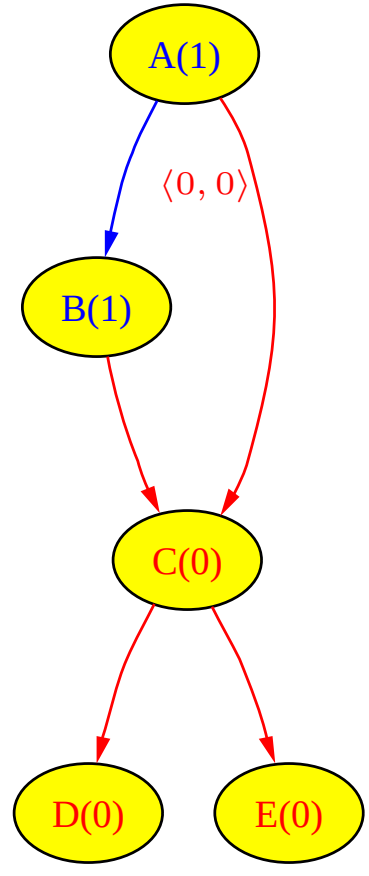
SIG Construction

- Step 1: Acyclic graph reduction
  - Presence of recursion
  - Identify Strongly connected components (SCCs)
  - Merge SCCs
- Step 2: Acyclic graph expansion
  - Top-down topological traversal
  - Annotate each call graph edge



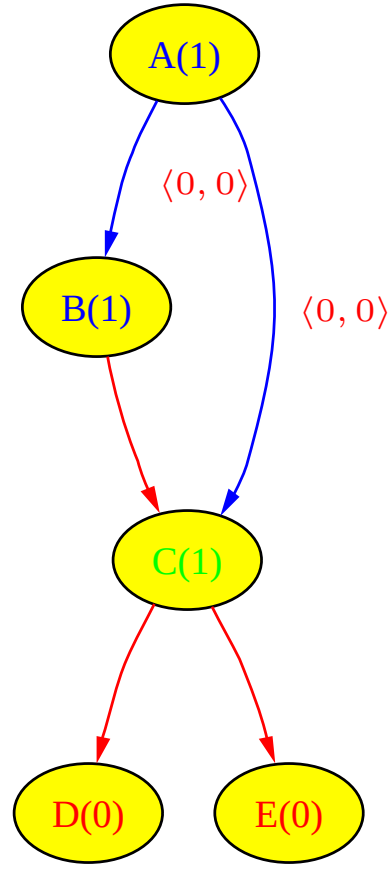
SIG Construction

- Step 1: Acyclic graph reduction
  - Presence of recursion
  - Identify Strongly connected components (SCCs)
  - Merge SCCs
- Step 2: Acyclic graph expansion
  - Top-down topological traversal
  - Annotate each call graph edge



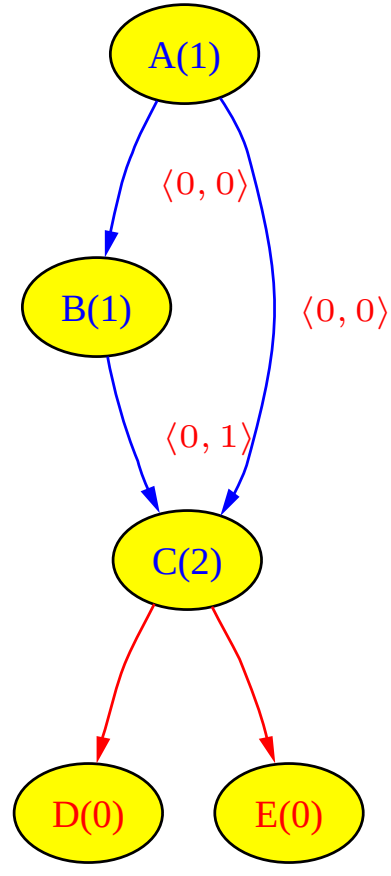
SIG Construction

- Step 1: Acyclic graph reduction
  - Presence of recursion
  - Identify Strongly connected components (SCCs)
  - Merge SCCs
- Step 2: Acyclic graph expansion
  - Top-down topological traversal
  - Annotate each call graph edge



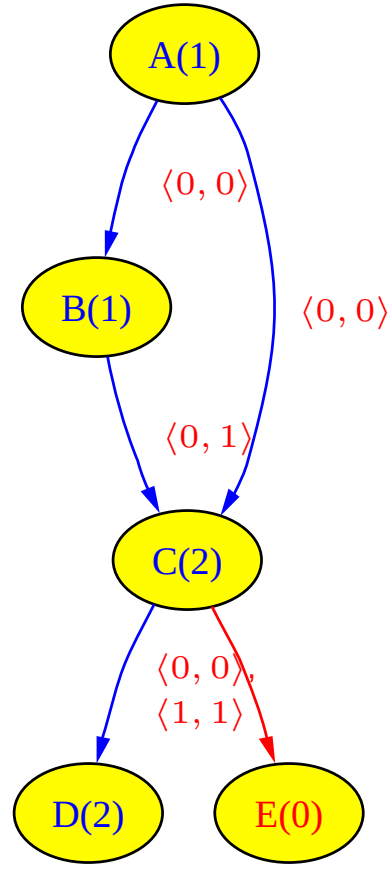
SIG Construction

- Step 1: Acyclic graph reduction
  - Presence of recursion
  - Identify Strongly connected components (SCCs)
  - Merge SCCs
- Step 2: Acyclic graph expansion
  - Top-down topological traversal
  - Annotate each call graph edge



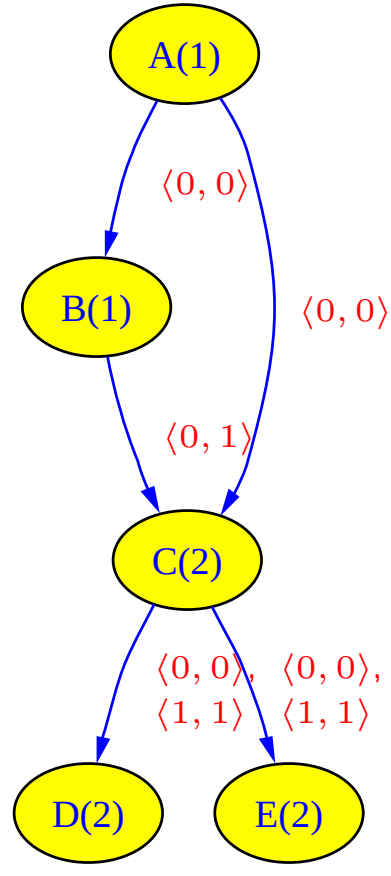
SIG Construction

- Step 1: Acyclic graph reduction
  - Presence of recursion
  - Identify Strongly connected components (SCCs)
  - Merge SCCs
- Step 2: Acyclic graph expansion
  - Top-down topological traversal
  - Annotate each call graph edge



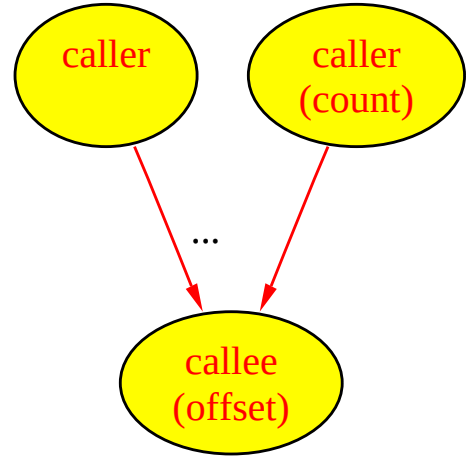
SIG Construction

- Step 1: Acyclic graph reduction
  - Presence of recursion
  - Identify Strongly connected components (SCCs)
  - Merge SCCs
- Step 2: Acyclic graph expansion
  - Top-down topological traversal
  - Annotate each call graph edge



SIG Edge Construction

- Contexts for each procedure numbered continuously
- SIG edges for each CG edge
  - $\langle 0, \text{offset} \rangle$, $\langle 1, \text{offset} + 1 \rangle$, ...
    $\langle \text{count} - 1, \text{offset} + \text{count} - 1 \rangle$
  - Explicit construction is expensive



SIG Edge Construction

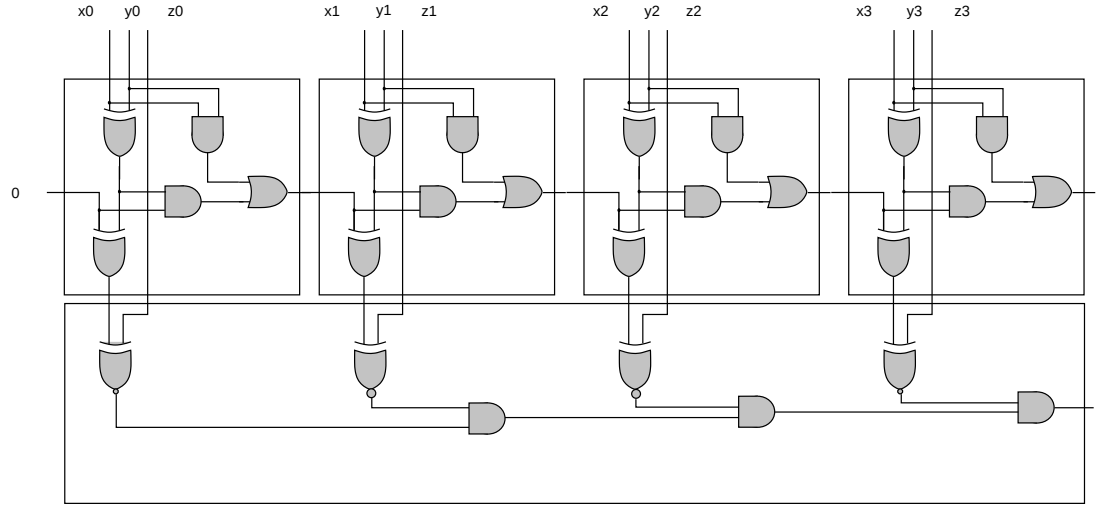
- $\{\langle x, y\rangle\}$ satisfy two conditions
  - $y = x +$ offset
  - $x <$ count

SIG Edge Construction

- $\{\langle x, y\rangle\}$ satisfy two conditions
  - $y = x +$ offset
  - $x <$ count
- Built by constructing circuits

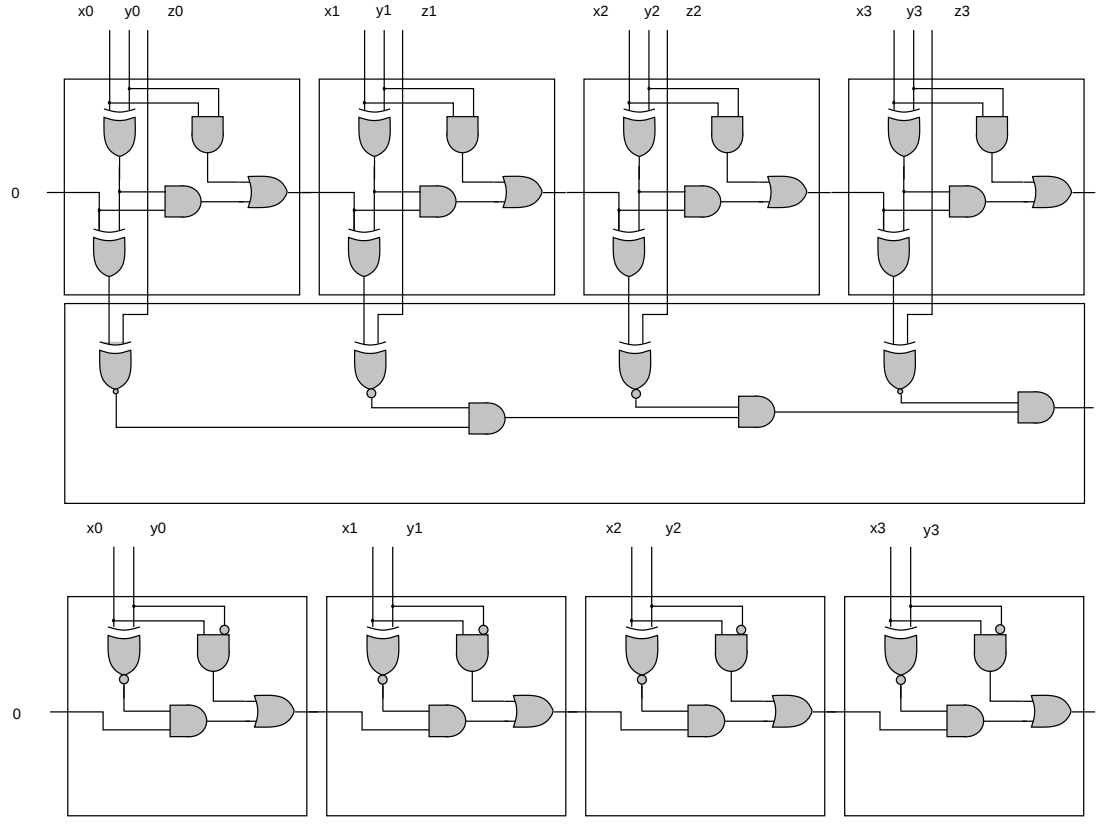
SIG Edge Construction

- $\{\langle x, y\rangle\}$ satisfy two conditions
  - $y = x +$ offset
  - $x <$ count
- Built by constructing circuits
  - Adder



SIG Edge Construction

- $\{\langle x, y\rangle\}$ satisfy two conditions
  - $y = x +$ offset
  - $x <$ count
- Built by constructing circuits
  - Adder
  - Comparator



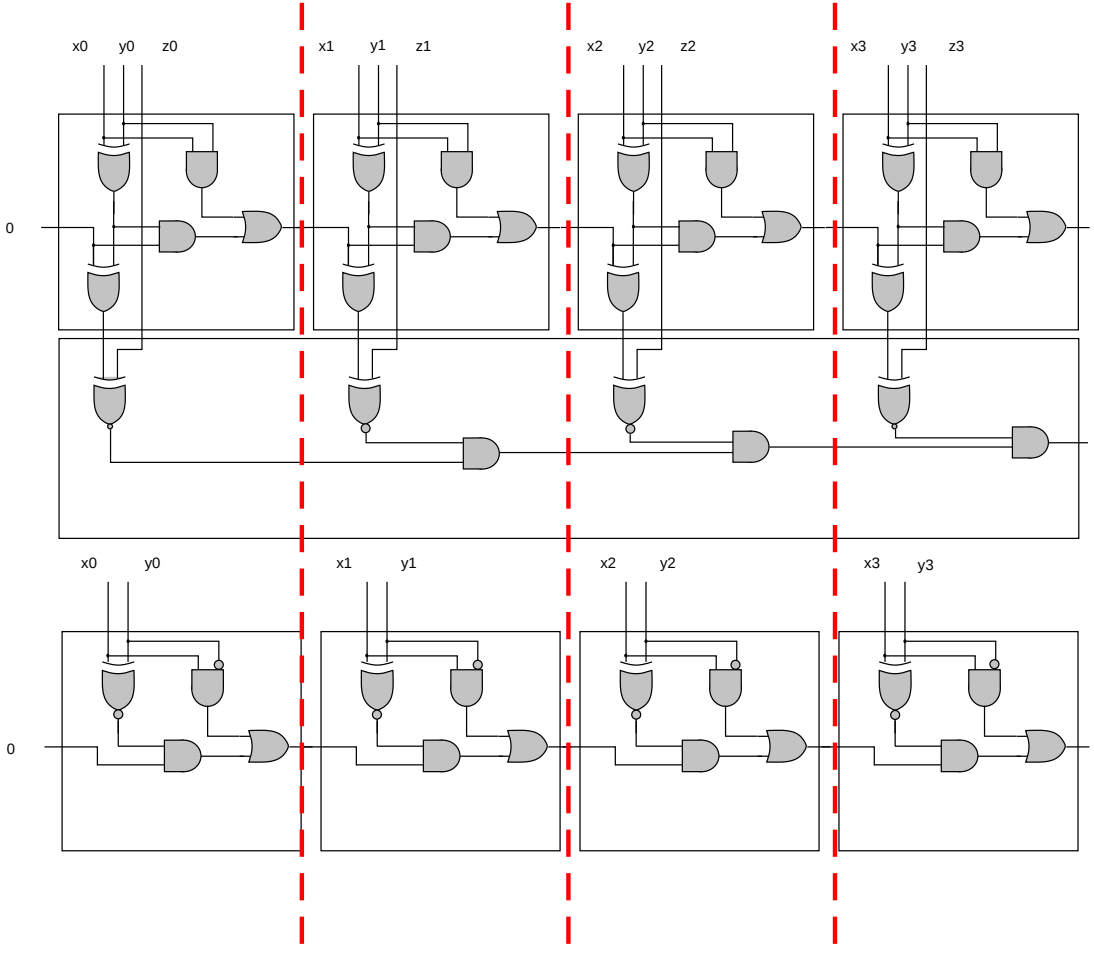
Insight in Complexity

- Algorithm polynomial to BDD size
  - BDD size in general is exponential
- Question
  Can we bound BDD size of SIG?
- Berman [TCAD'01]
  - Establish relation between circuit complexity and BDD complexity
  - BDD size $\propto 2^{\text{circuit cutwidth}} m$
  - m is number of BDD variables

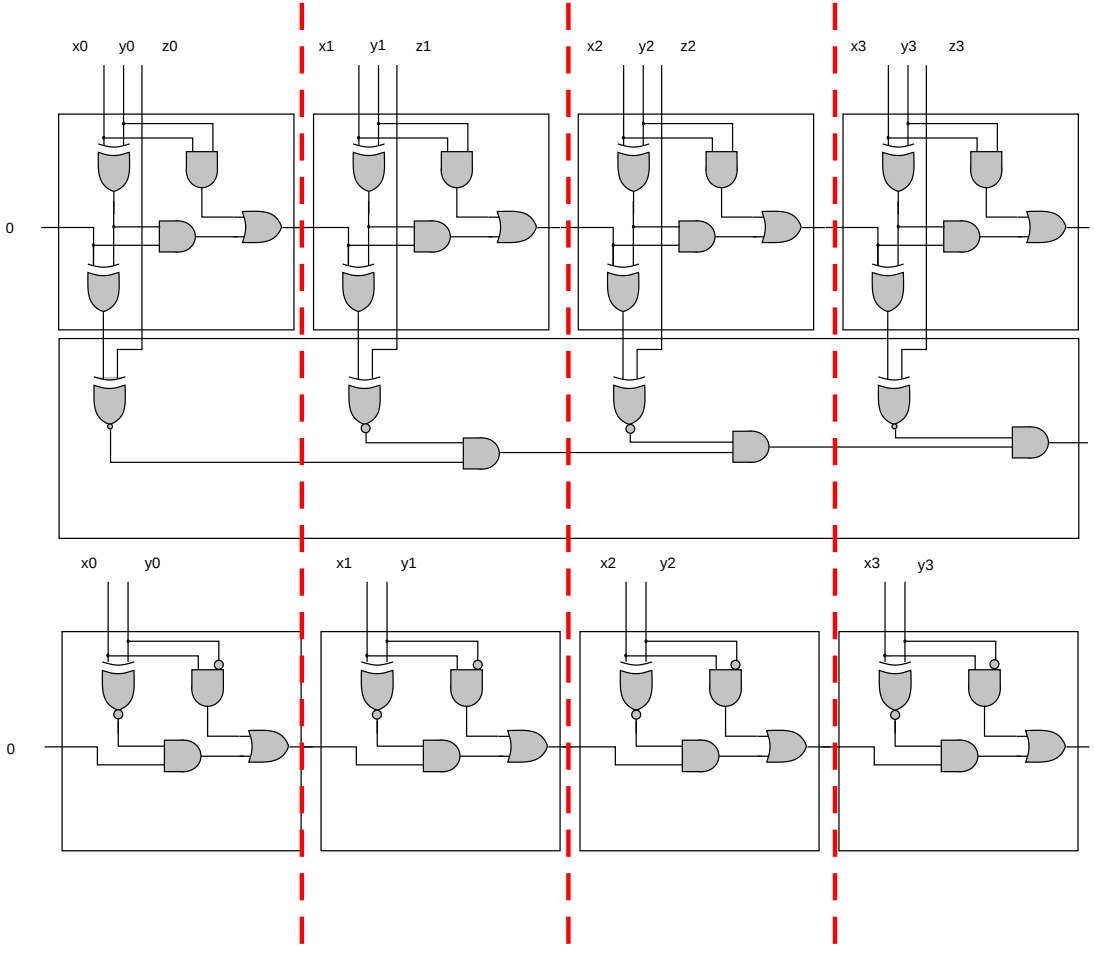
Insight in Complexity

- Algorithm polynomial to BDD size
  - BDD size in general is exponential
- Question
  Can we bound BDD size of SIG?
- Berman [TCAD'01]
  - Establish relation between circuit complexity and BDD complexity
  - BDD size $\propto 2^{\text{circuit cutwidth}} m$
  - m is number of BDD variables

Insight in Complexity



Insight in Complexity



- Cutwidth is constant
- BDD size of adder, comparator linear!
- SIG construction polynomial

Outline

- An Old Problem
- A New Strategy
- Taming Context Sensitivity

Outline

- An Old Problem
- A New Strategy
- Taming Context Sensitivity
  - Evil of paths (IG) overcome by bless of paths (BDD)

Outline

- An Old Problem
- A New Strategy
- Taming Context Sensitivity
  - Evil of paths (IG) overcome by bless of paths (BDD)
  - Circuit inspired solution for combinatorial problems

Outline

- An Old Problem
- A New Strategy
- Taming Context Sensitivity
  - Evil of paths (IG) overcome by bless of paths (BDD)
  - Circuit inspired solution for combinatorial problems
- Result and Conclusion

Experiment Setup

- Sun Blade 150, 550MHz CPU, 128MB RAM
- Benchmarks
  - prolangs: pointer analysis community
  - MediaBench: embedded systems community
  - SPEC2K: computer architecture community

prolangs				MediaBench				SPEC2000			
name	#lines	#contexts	#blocks	name	#lines	#contexts	#blocks	name	#lines	#contexts	#blocks
315	1411	49	136	gsm	5473	267	1124	vortex	67211	$9.217*10^{10}$	18433
TWMC	24032	6522	4613	pegwit	5503	1968	1121	bzip2	4665	495	995
simulator	3558	8953	1316	pgp	28065	199551	5265	vpr	16984	179905	4318
larn	9933	1750823	6180	mpeg2dec	9823	44979	2748	crafty	19478	317378	5282
moria	25002	318675286	9446	mpeg2enc	7605	1955	2997	twolf	19756	5538	4231

Experiment Goal

- Goal: evaluate scalability
- Comparative study
  - FICS: flow-insensitive, context-sensitive analysis
  - FSCI: flow-sensitive, context-insensitive analysis
  - FSCS: flow-sensitive, context-sensitive analysis
  - FSCS*: flow-sensitive, context-sensitive, field-sensitive analysis

Experimental Result: prolangs

Benchmarks			Flow Time (s)	Solver Time (s)	Total Time (s)
prolangs	315	CS	0	0.01	0.01
		FS	0.01	0	0.01
		FSCS	0.04	0.01	0.05
		FSCS*	0.04	0.01	0.05
	T-W-MC	CS	0	3.72	3.72
		FS	4.43	0.59	5.02
		FSCS	8.71	1.76	10.47
		FSCS*	8.86	1.68	10.54
	larn	CS	0	0.83	0.83
		FS	0.67	0.19	0.86
		FSCS	21.83	1.64	23.47
		FSCS*	21.94	1.68	23.62
	moria	CS	0	1.47	1.47
		FS	2.24	0.47	2.71
		FSCS	40.60	4.4	45.00
		FSCS*	40.87	4.47	45.34
	simulator	CS	0	0.10	0.10
		FS	0.06	0.04	0.1
		FSCS	0.76	0.10	0.86
		FSCS*	0.79	0.14	0.93

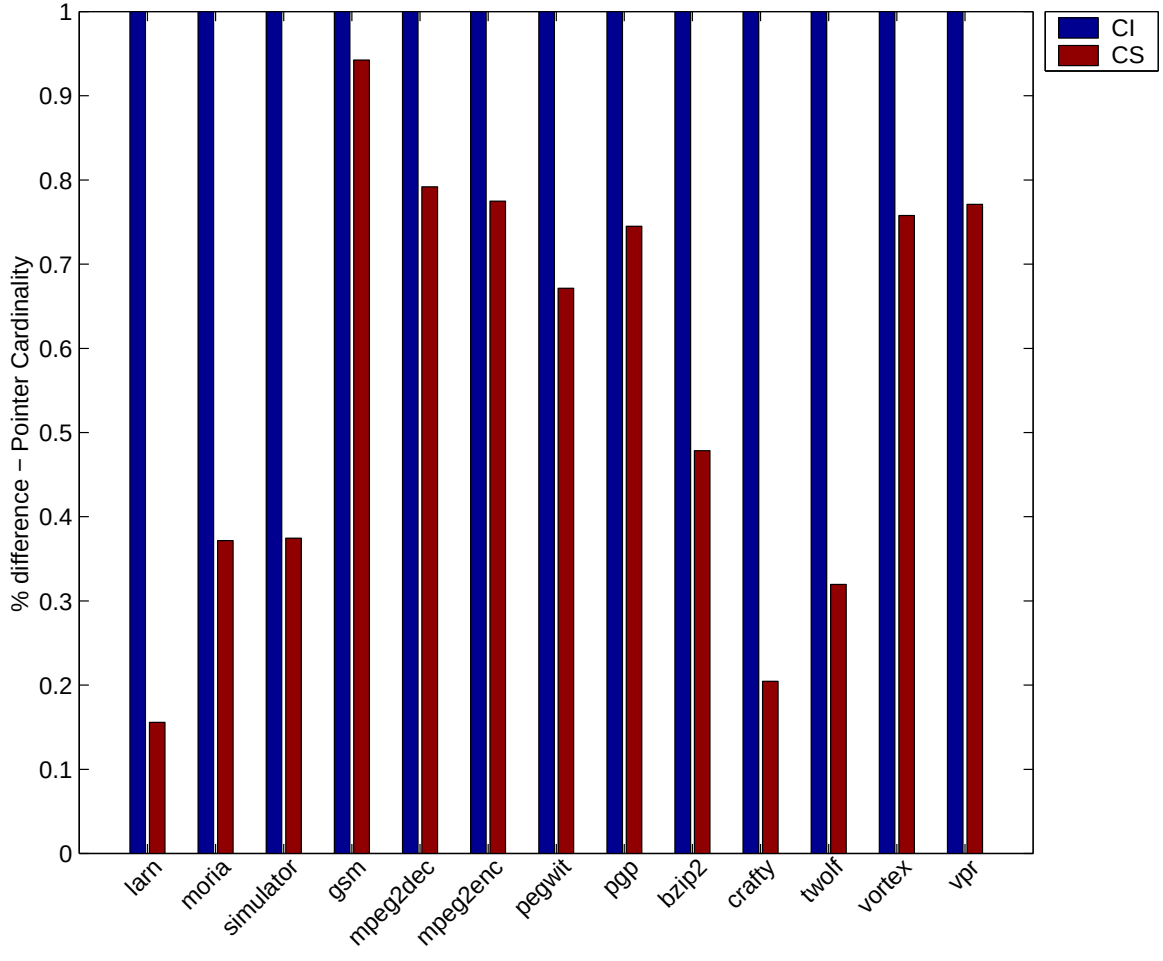
Experimental Result: MediaBench

Benchmarks			Flow Time (s)	Solver Time (s)	Total Time (s)
MediaBench	gsm	CS	0	0.05	0.05
		FS	0.19	0.05	0.24
		FSCS	0.55	0.1	0.65
		FSCS*	0.59	0.08	0.67
	mpeg2dec	CS	0	0.20	0.20
		FS	0.29	0.23	0.52
		FSCS	3.15	0.54	3.69
		FSCS*	3.23	0.48	2.19
	mpeg2enc	CS	0	0.19	0.19
		FS	0.95	0.14	1.09
		FSCS	1.44	0.27	1.71
		FSCS*	1.52	0.23	1.75
	pegwit	CS	0	0.12	0.12
		FS	0.09	0.09	0.18
		FSCS	0.97	0.26	1.23
		FSCS*	0.96	0.25	1.21
	pgp	CS	0	1.05	1.05
		FS	2.32	0.73	3.05
		FSCS	26.28	6.03	32.31
		FSCS*	26.19	5.90	32.09

Experimental Result: SPEC2K

Benchmarks			Flow Time (s)	Solver Time (s)	Total Time (s)
SPEC2000	255.vortex	CS	0	4.32	4.32
		FS	10.53	1.31	11.84
		FSCS	135.59	7.49	143.08
		FSCS*	136.68	7.71	144.39
	186.crafty	CS	0	0.53	0.53
		FS	3.1	0.44	3.54
		FSCS	12.12	2.06	14.18
		FSCS*	12.17	2.09	14.26
	256.bzip2	CS	0	0.02	0.02
		FS	0.06	0.02	0.08
		FSCS	0.3	0.06	0.36
		FSCS*	0.32	0.06	0.38
	300.twolf	CS	0	0.09	0.09
		FS	5.25	0.09	5.34
		FSCS	9.19	0.26	9.45
		FSCS*	9.21	0.20	9.41
	175.vpr	CS	0	0.23	0.23
		FS	1.37	0.1	1.47
		FSCS	5.96	0.4	6.36
		FSCS*	5.84	0.34	6.18

Experimental Result: Precision



Conclusion

- Extends success of formal verification to general combinatorial problems
- Promotes a new way of thinking
- Leads to new milestones in solving the pointer analysis problem
- Lessons learned
  - Elegance at the expense of efficiency

Conclusion

- Extends success of formal verification to general combinatorial problems
- Promotes a new way of thinking
- Leads to new milestones in solving the pointer analysis problem
- Lessons learned
  - ~~Elegance at the expense of efficiency~~

Conclusion

- Extends success of formal verification to general combinatorial problems
- Promotes a new way of thinking
- Leads to new milestones in solving the pointer analysis problem
- Lessons learned
  - ~~Elegance at the expense of efficiency~~
  - BDD is a panacea

Conclusion

- Extends success of formal verification to general combinatorial problems
- Promotes a new way of thinking
- Leads to new milestones in solving the pointer analysis problem
- Lessons learned
  - ~~Elegance at the expense of efficiency~~
  - ~~BDD is a panacea~~

Further Readings

References

[Zhu(2002)] Jianwen Zhu. Symbolic pointer analysis. In Proceedings of the International Conference on Computer-Aided Design (ICCAD), San Jose, November 2002.

[Berndl et al.(2003)Berndl, Lhoták, Qian, Hendren, and Umanee] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Point-to analysis using BDD. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), San Diego, June 2003.

[Zhu and Calman(2004)] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2004.

[Lhoták and Hendren(2004)] Ondřej Lhoták and Laurie Hendren. Jedd: A BDD-based relational extension of Java. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2004.

[Whaley and Lam(2004)] John Whaley and Monica Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2004.